Gen AI Builder SDK

Release 1.3

EnterpriseDB Corporation

Oct 31, 2025

Contents

1 Why Consider a Framework? 2

2 Prerequisites 3
- 2.1 1. OpenAI API Key . . . 3
- 2.2 2. Install Gen AI Builder . . . 3
- 2.3 Using Uv . . . 3
- 2.4 Using Pip . . . 4
- 2.5 Installing Optional Dependencies . . . 4

3 Prompt Task 5

4 Changing Models 6

5 Templating 7

6 Rules 8

7 Structured Output 11

8 Conversation Memory 13

9 Tools 15

10 Structures 17
- 10.1 Directly Using PromptSummaryEngine . . . 18
- 10.2 TextSummaryTask . . . 20

11 Conclusion and Next Steps 23
- 11.1 Data abstractions . . . 23
  - 11.1.1 Overview . . . 23
    - Artifacts . . . 24
    - Chunkers . . . 25
    - Loaders . . . 26
- 11.2 Prompt . . . 29
  - 11.2.1 Overview . . . 29
  - 11.2.2 Structured Output . . . 30
    - Prompt Task . . . 31
  - 11.2.3 Prompt Drivers . . . 32

OpenAI Chat . . . 32
OpenAI Compatible . . . 33
Azure OpenAI Chat . . . 34
Gen AI Builder . . . 34
Cohere . . . 35
Anthropic . . . 35
Google . . . 36
Amazon Bedrock . . . 36
Ollama . . . 37
Hugging Face Hub . . . 37
Text Generation Interface . . . 38
Hugging Face Pipeline . . . 38
Amazon SageMaker Jumpstart . . . 39
Grok . . . 39
Perplexity . . . 39
11.3 RAG Engine . . . 85
11.3.1 RAG Engines . . . 85
RAG Stages . . . 86
RAG Modules . . . 86
Query Stage . . . 86
Query Stage Modules . . . 86
Retrieval Stage . . . 86
Retrieval Stage Modules . . . 86
Response Stage . . . 86
Response Stage Modules . . . 86
RAG Context . . . 86
Simple Example . . . 86
Advanced Example . . . 87
RAG Tool . . . 90
11.4 Events . . . 93
11.4.1 Overview . . . 93
11.4.2 Specific Event Types . . . 93
11.4.3 All Event Types . . . 94
11.4.4 Stream Iterator . . . 95
11.4.5 Context Managers . . . 95
11.4.6 Streaming . . . 95
11.4.7 Counting Tokens . . . 98
11.4.8 Inspecting Payloads . . . 98
11.4.9 `EventListenerDriver.on_event` Return Value Behavior . . . 98
`EventListener.on_event` is None . . . 98
Return `BaseEvent` or `dict` . . . 98
Return `None` . . . 98
11.5 SDK reference . . . 102
11.5.1 artifacts . . . 102
Attributes . . . 102
to_text() . . . 103
Attributes . . . 103
attrs_post_init() . . . 104
make_name() . . . 104
to_bytes(**kwargs) . . . 105
to_text() . . . 105
Attributes . . . 105
bool() . . . 108
len() . . . 108

str() . . . 108
to_bytes(**kwargs) . . . 109
to_text()abstractmethod . . . 109
Attributes . . . 109
to_bytes(**kwargs) . . . 110
to_text() . . . 110
Attributes . . . 110
add(other) . . . 112
eq(value) . . . 112
parse_bool(value)classmethod . . . 112
to_text() . . . 112
Attributes . . . 112
to_text() . . . 114
Attributes . . . 114
to_text() . . . 114
Attributes . . . 115
attrs_post_init() . . . 116
make_name() . . . 117
to_bytes(**kwargs) . . . 117
to_text() . . . 117
Attributes . . . 117
Attributes . . . 118
to_text() . . . 118
Attributes . . . 119
to_text() . . . 119
value_to_json(value)classmethod . . . 120
add(other) . . . 121
bool() . . . 121
getitem(key) . . . 121
iter() . . . 121
has_items() . . . 121
is_type(target_type) . . . 122
to_text() . . . 122
validate*value(*, value) . . . 122
Attributes . . . 122
to_text() . . . 123
add(other) . . . 124
bool() . . . 124
generate_embedding(driver) . . . 124
to_text() . . . 124
token_count(tokenizer) . . . 124
Attributes . . . 125
to_bytes(*, headers=None, **kwargs) . . . 126
Parameters . . . 126
Returns . . . 126
to_text() . . . 126
11.5.2 chunkers . . . 153
__find_midpoint_index(separator, subchunks, half_token_count) . . . 155
__get_subchunks(separator, subchunks, balance_index) . . . 156
_chunk_recursively(chunk, current_separator=None) . . . 156
chunk(text) . . . 157
validate*max_tokens(*, max_tokens) . . . 157
init(value, is_prefix=False) . . . 158
11.5.3 common . . . 166

str() . . . 166
from_deltas(deltas)classmethod . . . 168
from_deltas(deltas)classmethod . . . 169
Attributes . . . 169
from_deltas(deltas)classmethod . . . 172
Usage . . . 174
is_assistant() . . . 174
is_system() . . . 174
is_user() . . . 175
bool() . . . 175
len() . . . 175
str() . . . 176
from_deltas(deltas)abstractmethodclassmethod . . . 176
to_text() . . . 176
from_deltas(deltas)classmethod . . . 177
from_deltas(deltas)classmethod . . . 177
init(content, **kwargs) . . . 178
str() . . . 179
get_content_type(content_type) . . . 179
has_all_content_type(content_type) . . . 179
has_any_content_type(content_type) . . . 179
is_text() . . . 179
to_artifact(meta=None) . . . 179
to_text() . . . 180
Call . . . 180
__to_message_content(artifact) . . . 183
add_assistant_message(artifact) . . . 183
add_message(artifact, role) . . . 183
add_system_message(artifact) . . . 184
add_user_message(artifact) . . . 184
from_artifact(artifact)classmethod . . . 184
to_output_json_schema(schema_id='Output Format') . . . 184
11.5.4 True}) . . . 185
from_deltas(deltas)classmethod . . . 185
Attributes . . . 186
str() . . . 188
from_native_tool_name(native_tool_name)classmethod . . . 188
to_native_tool_name() . . . 189
11.5.5 configs . . . 215
base_config . . . 215
defaults_config . . . 215
drivers . . . 216
logging . . . 253
11.5.6 drivers . . . 258
Attributes . . . 259
client() . . . 262
try_embed_chunk(chunk, **kwargs) . . . 263
Attributes . . . 263
_make_request(request) . . . 268
client() . . . 269
try_image_inpainting(prompts, image, mask, negative_prompts=None) . . . 269
try_image_outpainting(prompts, image, mask, negative_prompts=None) . . . 269
try_image_variation(prompts, image, negative_prompts=None) . . . 270
try_text_to_image(prompts, negative_prompts=None) . . . 271

__to_bedrock_message_content(content) . . . 276
__to_bedrock_messages(messages) . . . 277
__to_bedrock_role(message) . . . 278
__to_bedrock_tool_use_content(artifact) . . . 278
__to_bedrock_tools(tools) . . . 278
__to_prompt_stack_delta_message_content(event) . . . 278
__to_prompt_stack_message_content(content) . . . 279
_base_params(prompt_stack) . . . 280
client() . . . 280
try_run(prompt_stack) . . . 280
try_stream(prompt_stack) . . . 281
validate*structured_output_strategy(*, value) . . . 282
Attributes . . . 282
_invoke_model(payload) . . . 285
client() . . . 285
try_embed_artifact(artifact, **kwargs) . . . 285
try_embed_chunk(chunk, **kwargs) . . . 285
_get_key() . . . 287
load() . . . 287
store(runs, metadata) . . . 287
table() . . . 288
Attributes . . . 288
upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs) . . . 290
_post_process(meta, records)classmethod . . . 293
_process_cells_from_rows_and_columns(columns, rows)classmethod . . . 294
_process_columns_from_column_metadata(meta)classmethod . . . 294
_process_rows_from_records(records)classmethod . . . 294
client() . . . 294
execute_query(query) . . . 294
execute_query_raw(query) . . . 294
get_table_schema(table_name, schema=None) . . . 295
validate*params(*, workgroup_name) . . . 296
Attributes . . . 296
_is_a_directory(full_key) . . . 300
_list_files_and_dirs(full_key, **kwargs) . . . 301
_normpath(path) . . . 301
_to_dir_full_key(path) . . . 302
_to_full_key(path) . . . 302
client() . . . 302
try_list_files(path) . . . 302
try_load_file(path) . . . 302
try_save_file(path, value) . . . 303
client() . . . 304
try_embed_chunk(chunk, **kwargs) . . . 304
__prompt_stack_to_tokens(prompt_stack) . . . 308
_base_params(prompt_stack) . . . 308
_prompt_stack_to_messages(prompt_stack) . . . 309
client() . . . 309
prompt_stack_to_string(prompt_stack) . . . 309
try_run(prompt_stack) . . . 309
try_stream(prompt_stack) . . . 310
validate*stream(*, stream) . . . 310
validate*structured_output_strategy(*, value) . . . 310
client() . . . 311

try_publish_event_payload(event_payload) . . . 311
try_publish_event_payload_batch(event_payload_batch) . . . 312
Attributes . . . 312
__to_anthropic_content(message) . . . 318
__to_anthropic_message_content(content) . . . 318
__to_anthropic_messages(messages) . . . 319
__to_anthropic_role(message) . . . 319
__to_anthropic_tool_result_content(artifact) . . . 319
__to_anthropic_tools(tools) . . . 320
__to_prompt_stack_delta_message_content(event) . . . 320
__to_prompt_stack_message_content(content) . . . 321
_base_params(prompt_stack) . . . 321
client() . . . 322
try_run(prompt_stack) . . . 322
try_stream(prompt_stack) . . . 322
validate*structured_output_strategy(*, value) . . . 323
Attributes . . . 323
client() . . . 330
collection() . . . 330
delete_vector(vector_id) . . . 330
Parameters . . . 330
load_entries(*, namespace=None) . . . 331
Parameters . . . 331
Returns . . . 331
load_entry(vector_id, *, namespace=None) . . . 332
Parameters . . . 332
Returns . . . 332
query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs) . . . 333
Parameters . . . 333
Returns . . . 333
upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs) . . . 335
Parameters . . . 335
Returns . . . 335
client() . . . 337
try_publish_event_payload(event_payload) . . . 337
try_publish_event_payload_batch(event_payload_batch) . . . 337
query_vector(vector, *, count=None, namespace=None, include_vectors=False, offset=None, **kwargs) . . . 339
Attributes . . . 340
_base_params(prompt_stack) . . . 344
client() . . . 344
Attributes . . . 344
client() . . . 349
Attributes . . . 349
client() . . . 352
Attributes . . . 353
client() . . . 356
run(*args) . . . 357
try_run(*args)abstractmethod . . . 357
after_run() . . . 358
before_run() . . . 358
run(audio, prompts=None) . . . 358
try_run(audio, prompts=None)abstractmethod . . . 358
_from_params_dict(params_dict) . . . 359

_to_params_dict(runs, metadata) . . . 359
load()abstractmethod . . . 359
store(runs, metadata)abstractmethod . . . 359
make_additional_params(negative_prompts, device)abstractmethod . . . 360
make_image_param(image)abstractmethod . . . 360
prepare_pipeline(model, device)abstractmethod . . . 360
Attributes . . . 360
attrs_post_init() . . . 363
_embed_long_string(string, *, vector_operation=None) . . . 363
embed(value, *, vector_operation=None) . . . 364
embed_string(string, *, vector_operation=None) . . . 364
embed_text_artifact(artifact, *, vector_operation=None) . . . 365
try_embed_artifact(artifact, *, vector_operation=None) . . . 365
try_embed_chunk(chunk, *, vector_operation=None)abstractmethod . . . 365
_safe_publish_event_payload(event_payload) . . . 367
_safe_publish_event_payload_batch(event_payload_batch) . . . 367
flush_events() . . . 367
publish_event(event) . . . 368
try_publish_event_payload(event_payload)abstractmethod . . . 368
try_publish_event_payload_batch(event_payload_batch)abstractmethod . . . 368
Attributes . . . 368
list_files(path) . . . 370
load_artifact(path) . . . 370
load_file(path) . . . 370
save_artifact(path, artifact) . . . 371
save_file(path, value) . . . 371
try_list_files(path)abstractmethod . . . 371
try_load_file(path)abstractmethod . . . 371
try_save_file(path, value)abstractmethod . . . 371
after_run() . . . 374
before_run(prompts, negative_prompts=None) . . . 374
run_image_inpainting(prompts, image, mask, negative_prompts=None) . . . 374
run_image_outpainting(prompts, image, mask, negative_prompts=None) . . . 374
run_image_variation(prompts, image, negative_prompts=None) . . . 375
run_text_to_image(prompts, negative_prompts=None) . . . 375
try_image_inpainting(prompts, image, mask, negative_prompts=None)abstractmethod . . . 375
try_image_outpainting(prompts, image, mask, negative_prompts=None)abstractmethod . . . 376
try_image_variation(prompts, image, negative_prompts=None)abstractmethod . . . 376
try_text_to_image(prompts, negative_prompts=None)abstractmethod . . . 376
get_generated_image(response)abstractmethod . . . 377
image_inpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)abstractmethod . . . 377
image_outpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)abstractmethod . . . 378
image_variation_request_parameters(prompts, image, negative_prompts=None, seed=None)abstractmethod . . . 378
text_to_image_request_parameters(prompts, image_width, image_height, negative_prompts=None, seed=None)abstractmethod . . . 378
Attributes . . . 379
enter() . . . 380
exit(exc_type, exc_value, exc_traceback) . . . 380
get_span_id()abstractmethod . . . 380
observe(call)abstractmethod . . . 381
Attributes . . . 381

__build_message(delta_contents, usage) . . . 387
__process_run(prompt_stack) . . . 388
__process_stream(prompt_stack) . . . 388
_init_structured_output(prompt_stack) . . . 389
after_run(result) . . . 390
before_run(prompt_stack) . . . 390
prompt_stack_to_string(prompt_stack) . . . 390
Parameters . . . 390
Returns . . . 390
run(prompt_input) . . . 391
try_run(prompt_stack)abstractmethod . . . 392
try_stream(prompt_stack)abstractmethod . . . 392
run(query, artifacts)abstractmethod . . . 392
Attributes . . . 392
_from_ruleset_dict(params_dict) . . . 394
_get_rule(value, meta) . . . 394
load(ruleset_name)abstractmethod . . . 394
RowResultdataclass . . . 395
execute_query(query)abstractmethod . . . 395
execute_query_raw(query)abstractmethod . . . 395
get_table_schema(table_name, schema=None)abstractmethod . . . 395
Attributes . . . 395
run(*args) . . . 396
try_run(*args)abstractmethod . . . 396
after_run() . . . 397
before_run(prompts) . . . 397
run_text_to_audio(prompts) . . . 397
try_text_to_audio(prompts)abstractmethod . . . 398
Entry . . . 402
_get_default_vector_id(value) . . . 403
delete_vector(vector_id)abstractmethod . . . 403
does_entry_exist(vector_id, *, namespace=None) . . . 403
load_artifacts(*, namespace=None) . . . 403
load_entries(*, namespace=None)abstractmethod . . . 404
load_entry(vector_id, *, namespace=None)abstractmethod . . . 404
query(query, *, count=None, namespace=None, include_vectors=False, **kwargs) . . . 404
query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs) . . . 404
upsert(value, *, namespace=None, meta=None, vector_id=None, **kwargs) . . . 405
upsert_collection(artifacts, *, meta=None, **kwargs) . . . 405
upsert_text(string, *, namespace=None, meta=None, vector_id=None, **kwargs) . . . 406
upsert_text_artifact(artifact, *, namespace=None, meta=None, vector_id=None, **kwargs) . . . 406
upsert_text_artifacts(artifacts, *, meta=None, **kwargs) . . . 407
upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)abstractmethod . . . 407
extract_page(page)abstractmethod . . . 408
fetch_url(url)abstractmethod . . . 408
scrape_url(url) . . . 408
search(query, **kwargs)abstractmethod . . . 409
Attributes . . . 409
_request_parameters(prompts, width=None, height=None, image=None, mask=None, negative_prompts=None, seed=None, mask_source=None) . . . 415
get_generated_image(response) . . . 416
image_inpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None) . . . 417

image_outpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None) . . . 417
image_variation_request_parameters(prompts, image, negative_prompts=None, seed=None) . 418
text_to_image_request_parameters(prompts, image_width, image_height, negative_prompts=None, seed=None) . . . 418
Attributes . . . 418
_add_common_params(request, width, height, seed=None) . . . 422
get_generated_image(response) . . . 423
image_inpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None) . . . 423
image_outpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None) . . . 424
image_variation_request_parameters(prompts, image, negative_prompts=None, seed=None) . 424
text_to_image_request_parameters(prompts, image_width, image_height, negative_prompts=None, seed=None) . . . 425
Attributes . . . 426
client() . . . 428
try_embed_chunk(chunk, **kwargs) . . . 428
Attributes . . . 429
__to_cohere_message_content(content) . . . 434
__to_cohere_messages(messages) . . . 435
__to_cohere_role(message, message_content=None) . . . 436
__to_cohere_tools(tools) . . . 436
__to_prompt_stack_delta_message_content(event) . . . 437
__to_prompt_stack_message_content(response) . . . 437
_base_params(prompt_stack) . . . 438
client() . . . 438
try_run(prompt_stack) . . . 439
try_stream(prompt_stack) . . . 439
run(query, artifacts) . . . 440
client() . . . 442
search(query, **kwargs) . . . 442
try_run(audio, prompts=None) . . . 443
try_embed_chunk(chunk, **kwargs) . . . 443
try_image_inpainting(prompts, image, mask, negative_prompts=None) . . . 444
try_image_outpainting(prompts, image, mask, negative_prompts=None) . . . 445
try_image_variation(prompts, image, negative_prompts=None) . . . 445
try_text_to_image(prompts, negative_prompts=None) . . . 445
try_run(prompt_stack) . . . 446
try_stream(prompt_stack) . . . 446
try_text_to_audio(prompts) . . . 446
delete_vector(vector_id) . . . 447
load_entries(*, namespace=None) . . . 448
load_entry(vector_id, *, namespace=None) . . . 448
query(query, *, count=None, namespace=None, include_vectors=False, **kwargs) . . . 448
query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs) . . 448
upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs) . . . 448
client() . . . 450
try_text_to_audio(prompts) . . . 450
client() . . . 451
search(query, **kwargs) . . . 451
Attributes . . . 452
11.5.7 Optional title for the content. Only works with `retrieval_document` task type. . . . 453
_params(chunk) . . . 453

try_embed_chunk(chunk, **kwargs) . . . 454
Attributes . . . 454
__to_google_message_content(content) . . . 461
__to_google_messages(prompt_stack) . . . 462
__to_google_role(message) . . . 462
__to_google_tools(tools) . . . 462
__to_prompt_stack_delta_message_content(content) . . . 463
__to_prompt_stack_message_content(content) . . . 463
_base_params(prompt_stack) . . . 464
client() . . . 465
try_run(prompt_stack) . . . 465
try_stream(prompt_stack) . . . 465
validate*structured_output_strategy(*, value) . . . 466
_search_google(query, **kwargs) . . . 467
search(query, **kwargs) . . . 468
_create_or_find_thread(thread_alias=None) . . . 471
_create_run(*args) . . . 471
_create_thread(thread_alias=None) . . . 472
_find_thread_by_alias(thread_alias) . . . 472
_get_run_events(assistant_run_id) . . . 473
_get_run_result(assistant_run_id) . . . 473
try_run(*args) . . . 473
Attributes . . . 474
Raises . . . 475
_call_api(method, path, json=None, *, raise_for_status=True) . . . 479
_get_url(path) . . . 479
load() . . . 479
store(runs, metadata) . . . 480
validate*api_key(*, value) . . . 480
Attributes . . . 480
_get_event_request(event_payload) . . . 484
_post_event(json) . . . 484
publish_event(event) . . . 484
try_publish_event_payload(event_payload) . . . 485
try_publish_event_payload_batch(event_payload_batch) . . . 485
validate*api_key(*, api_key) . . . 485
validate*run_id(*, structure_run_id) . . . 485
Attributes . . . 486
Raises . . . 487
attrs_post_init() . . . 490
_call_api(method, path, json=None, params=None, *, raise_for_status=True) . . . 490
_get_asset_url(full_key) . . . 491
_get_url(path) . . . 491
_is_a_directory(path) . . . 491
_to_full_key(path) . . . 491
try_list_files(path, postfix='') . . . 491
try_load_file(path) . . . 492
try_save_file(path, value) . . . 492
validate*bucket_id(*, value) . . . 493
try_image_inpainting(prompts, image, mask, negative_prompts=None) . . . 495
try_image_outpainting(prompts, image, mask, negative_prompts=None) . . . 495
try_image_variation(prompts, image, negative_prompts=None) . . . 495
try_text_to_image(prompts, negative_prompts=None) . . . 495
build_span_exporter(base_url, api_key, headers, structure_run_id)staticmethod . . . 499

format_span_id(span_id)staticmethod . . . 500
format_trace_id(trace_id)staticmethod . . . 500
get_span_id() . . . 500
validate*run_id(*, structure_run_id) . . . 500
__to_griptape_tools(tools) . . . 503
_base_params(prompt_stack) . . . 503
try_run(prompt_stack) . . . 504
try_stream(prompt_stack) . . . 504
Attributes . . . 504
Raises . . . 505
_call_api(method, path, *, raise_for_status=True) . . . 507
_get_url(path) . . . 508
load(ruleset_name) . . . 508
validate*api_key(*, value) . . . 509
_create_run(*args) . . . 511
_get_run_events(structure_run_id) . . . 511
_get_run_result(structure_run_id) . . . 511
try_run(*args) . . . 512
Attributes . . . 512
delete_vector(vector_id) . . . 517
load_artifacts(*, namespace=None) . . . 517
load_entries(*, namespace=None) . . . 517
load_entry(vector_id, *, namespace=None) . . . 517
query(query, *, count=None, namespace=None, include_vectors=None, distance_metric=None, filter=None, **kwargs) . . . 517
upsert_text(string, vector_id=None, namespace=None, meta=None, **kwargs) . . . 518
upsert_text_artifact(artifact, namespace=None, meta=None, vector_id=None, **kwargs) . . . 518
upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs) . . . 519
Attributes . . . 519
client() . . . 521
try_embed_chunk(chunk, **kwargs) . . . 521
Attributes . . . 521
__prompt_stack_to_tokens(prompt_stack) . . . 526
_base_params(prompt_stack) . . . 526
_prompt_stack_to_messages(prompt_stack) . . . 527
client() . . . 527
prompt_stack_to_string(prompt_stack) . . . 527
try_run(prompt_stack) . . . 528
try_stream(prompt_stack) . . . 528
validate*structured_output_strategy(*, value) . . . 529
Attributes . . . 529
try_image_inpainting(prompts, image, mask, negative_prompts=None) . . . 532
try_image_outpainting(prompts, image, mask, negative_prompts=None) . . . 533
try_image_variation(prompts, image, negative_prompts=None) . . . 533
try_text_to_image(prompts, negative_prompts=None) . . . 534
Attributes . . . 534
__prompt_stack_to_tokens(prompt_stack) . . . 537
_base_params(prompt_stack) . . . 537
_prompt_stack_to_messages(prompt_stack) . . . 537
pipeline() . . . 538
prompt_stack_to_string(prompt_stack) . . . 538
try_run(prompt_stack) . . . 538
try_stream(prompt_stack) . . . 539
validate*structured_output_strategy(*, value) . . . 539

Attributes . . . 539
_create_generation(prompts, negative_prompts, init_image_id=None) . . . 548
_download_image(url) . . . 548
_get_image_url(generation_id) . . . 549
_make_api_request(endpoint, request, method='POST') . . . 549
_upload_init_image(image) . . . 549
try_image_inpainting(prompts, image, mask, negative_prompts=None) . . . 550
try_image_outpainting(prompts, image, mask, negative_prompts=None) . . . 550
try_image_variation(prompts, image, negative_prompts=None) . . . 550
try_text_to_image(prompts, negative_prompts=None) . . . 551
load() . . . 552
store(runs, metadata) . . . 552
Attributes . . . 552
_full_path(path) . . . 554
_is_dir(full_path) . . . 554
try_list_files(path) . . . 554
try_load_file(path) . . . 555
try_save_file(path, value) . . . 555
run(query, artifacts) . . . 556
load(ruleset_name) . . . 557
try_run(*args) . . . 558
attrs_post_init() . . . 561
__namespaced_vector_id(vector_id, *, namespace) . . . 561
__save_entries_to_file(json_file) . . . 561
delete_vector(vector_id) . . . 562
load_entries(*, namespace=None) . . . 562
load_entries_from_file(json_file) . . . 562
load_entry(vector_id, *, namespace=None) . . . 562
query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs) . . . 562
upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs) . . . 563
Attributes . . . 564
extract_page(page) . . . 568
fetch_url(url) . . . 569
Attributes . . . 569
__process_results(results, *, include_vectors) . . . 576
client() . . . 576
delete_index(name) . . . 577
Parameters . . . 577
delete_vector(vector_id) . . . 577
get_indexes() . . . 577
Returns . . . 577
load_entries(*, namespace=None) . . . 578
Parameters . . . 578
Returns . . . 578
load_entry(vector_id, *, namespace=None) . . . 579
Parameters . . . 579
Returns . . . 579
query(query, *, count=None, namespace=None, include_vectors=False, include_metadata=True, **kwargs) . . . 580
Parameters . . . 580
Returns . . . 581
query_vector(vector, *, count=None, namespace=None, include_vectors=False, include_metadata=True, **kwargs) . . . 581
Parameters . . . 582

Returns . . . 582
upsert(value, *, namespace=None, meta=None, vector_id=None, **kwargs) . . . 583
Parameters . . . 583
Returns . . . 584
upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs) . . . 585
Parameters . . . 585
Raises . . . 585
Returns . . . 585
Attributes . . . 586
client() . . . 592
delete_vector(vector_id) . . . 592
get_collection() . . . 592
load_entries(*, namespace=None) . . . 592
load_entry(vector_id, *, namespace=None) . . . 593
Returns . . . 593
query_vector(vector, *, count=None, namespace=None, include_vectors=False, offset=None, **kwargs) . . . 594
upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs) . . . 595
get_span_id() . . . 596
observe(call) . . . 596
Attributes . . . 596
client() . . . 597
try_embed_chunk(chunk, **kwargs) . . . 598
Attributes . . . 598
__to_ollama_message_content(content) . . . 603
__to_ollama_role(message, message_content=None) . . . 604
__to_ollama_tools(tools) . . . 604
__to_prompt_stack_delta_message_content(content_delta) . . . 605
__to_prompt_stack_message_content(response) . . . 605
_base_params(prompt_stack) . . . 606
_prompt_stack_to_messages(prompt_stack) . . . 606
client() . . . 607
try_run(prompt_stack) . . . 607
try_stream(prompt_stack) . . . 608
EventHandler . . . 610
_create_run(thread_id, *args) . . . 611
client() . . . 611
try_run(*args) . . . 612
client() . . . 613
try_run(audio, prompts=None) . . . 613
Attributes . . . 614
__to_openai_message_content(content) . . . 627
__to_openai_messages(messages) . . . 629
__to_openai_role(message, message_content=None) . . . 630
__to_openai_tools(tools) . . . 630
__to_prompt_stack_delta_message_content(content_delta) . . . 630
__to_prompt_stack_message_content(response) . . . 631
_base_params(prompt_stack) . . . 632
_to_delta_message_stream(result) . . . 633
_to_message(result) . . . 634
client() . . . 634
try_run(prompt_stack) . . . 634
try_stream(prompt_stack) . . . 635
Attributes . . . 635

_params(chunk) . . . 639
client() . . . 639
try_embed_chunk(chunk, **kwargs) . . . 639
Attributes . . . 639
_build_model_params(values) . . . 650
Parameters . . . 650
_parse_image_response(response, prompt) . . . 651
client() . . . 652
try_image_inpainting(prompts, image, mask, negative_prompts=None) . . . 652
try_image_outpainting(prompts, image, mask, negative_prompts=None) . . . 652
try_image_variation(prompts, image, negative_prompts=None) . . . 653
try_text_to_image(prompts, negative_prompts=None) . . . 653
validate_image_size(attribute, value) . . . 654
client() . . . 656
try_text_to_audio(prompts) . . . 656
Attributes . . . 656
client() . . . 662
delete_vector(vector_id) . . . 662
load_entries(*, namespace=None) . . . 662
Returns . . . 662
load_entry(vector_id, *, namespace=None) . . . 663
Returns . . . 663
query_vector(vector, *, count=None, namespace=None, include_vectors=False, include_metadata=True, field_name='vector', **kwargs) . . . 664
Returns . . . 664
upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs) . . . 665
attrs_post_init() . . . 668
enter() . . . 668
exit(exc_type, exc_value, exc_traceback) . . . 668
_trace_provider_factory() . . . 669
get_span_id() . . . 669
observe(call) . . . 670
_base_params(prompt_stack) . . . 671
_to_message(result) . . . 671
prompt_driver() . . . 672
search(query, **kwargs) . . . 672
delete_vector(vector_id) . . . 674
engine() . . . 675
load_artifacts(*, namespace=None) . . . 675
load_entries(*, namespace=None) . . . 675
load_entry(vector_id, *, namespace=None) . . . 675
query(query, *, count=BaseVectorStoreDriver.DEFAULT_QUERY_COUNT, **kwargs) . . . 675
upsert_text(string, vector_id=None, namespace=None, meta=None, **kwargs) . . . 676
upsert_text_artifact(artifact, namespace=None, meta=None, vector_id=None, **kwargs) . . . 676
upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs) . . . 676
Attributes . . . 677
default_vector_model() . . . 683
delete_vector(vector_id) . . . 683
engine() . . . 683
load_entries(*, namespace=None) . . . 683
load_entry(vector_id, *, namespace=None) . . . 684
query_vector(vector, *, count=None, namespace=None, include_vectors=False, distance_metric='cosine_distance', **kwargs) . . . 684
setup(*, create_schema=True, install_uuid_extension=True, install_vector_extension=True) . . . 686

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs) 686
validate*connection_string(*, connection_string) 687
client() 690
delete_vector(vector_id) 690
index() 690
load_entries(*, namespace=None) 690
load_entry(vector_id, *, namespace=None) 691
query_vector(vector, *, count=None, namespace=None, include_vectors=False, include_metadata=True, **kwargs) 691
upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs) 692
extract_page(page) 693
fetch_url(url) 693
client() 694
try_publish_event_payload(event_payload) 694
try_publish_event_payload_batch(event_payload_batch) 694
Attributes 695
client() 706
delete_vector(vector_id) 706
Parameters 706
load_entries(*, namespace=None, **kwargs) 707
Parameters 707
Returns 707
load_entry(vector_id, *, namespace=None) 708
Parameters 708
Returns 708
query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs) . . 709
Parameters 709
Returns 709
upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, content=None, **kwargs) 710
Parameters 710
Returns 711
Attributes 712
load() 715
store(runs, metadata) 716
Attributes 716
_generate_key(vector_id, namespace=None) 722
_get_doc_prefix(namespace=None) 722
client() 722
delete_vector(vector_id) 722
load_entries(*, namespace=None) 722
Returns 722
load_entry(vector_id, *, namespace=None) 723
Returns 723
query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs) . . 724
Returns 724
upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs) 725
engine() 727
execute_query(query) 727
execute_query_raw(query) 728
get_table_schema(table_name, schema=None) 728
validate*get_connection(*, get_connection) 728
_get_table_schema(engine, table_name, schema=None)cachedstaticmethod 730
engine() 730

execute_query(query) . . . 730
execute_query_raw(query) . . . 730
get_table_schema(table_name, schema=None) . . . 731
Attributes . . . 731
make_additional_params(negative_prompts, device) . . . 734
make_image_param(image) . . . 734
prepare_pipeline(model, device) . . . 734
Attributes . . . 735
make_additional_params(negative_prompts, device) . . . 740
make_image_param(image) . . . 741
prepare_pipeline(model, device) . . . 741
Attributes . . . 741
make_additional_params(negative_prompts, device) . . . 743
make_image_param(image) . . . 744
prepare_pipeline(model, device) . . . 744
client() . . . 745
search(query, **kwargs) . . . 746
extract_page(page) . . . 747
fetch_url(url) . . . 747
Attributes . . . 748
client() . . . 751
try_embed_artifact(artifact, **kwargs) . . . 751
try_embed_chunk(chunk, **kwargs) . . . 751
try_publish_event_payload(event_payload) . . . 752
try_publish_event_payload_batch(event_payload_batch) . . . 752
11.5.8 engines . . . 1828
BaseExtractionEngine . . . 1829
BaseSummaryEngine . . . 1831
CsvExtractionEngine . . . 1832
EvalEngine . . . 1836
JsonExtractionEngine . . . 1841
PromptSummaryEngine . . . 1844
RagEngine . . . 1848
11.5.9 events . . . 1908
str() . . . 1909
Attributes . . . 1910
str() . . . 1910
from_task(task)classmethod . . . 1911
str()abstractmethod . . . 1912
Attributes . . . 1914
enter() . . . 1915
exit(type, value, traceback) . . . 1916
publish_event(event, *, flush=False) . . . 1916
str() . . . 1920
11.5.10 exceptions . . . 1936
init(dummy_class_name, dummy_method_name) . . . 1936
11.5.11 loaders . . . 1937
try_parse(data) . . . 1937
fetch(source) . . . 1938
save(destination, artifact) . . . 1938
Attributes . . . 1939
fetch(source)abstractmethod . . . 1940
load(source) . . . 1940
load_collection(sources) . . . 1940

parse(data) . . . 1941
to_key(source) . . . 1941
try_parse(data) . . . 1941
try_parse(data) . . . 1942
try_parse(data) . . . 1942
EmailQuery . . . 1944
Attributes . . . 1944
_count_messages(message_numbers) . . . 1946
fetch(source) . . . 1946
try_parse(data) . . . 1947
Attributes . . . 1947
try_parse(data) . . . 1948
parse(data) . . . 1949
try_parse(data, *, password=None) . . . 1950
fetch(source) . . . 1950
parse(data) . . . 1951
try_parse(data) . . . 1951
fetch(source) . . . 1952
try_parse(data) . . . 1952
11.5.12 memory . . . 1967
find_input_memory(memory_name) . . . 1970
get_storage_for(artifact) . . . 1970
load_artifacts(namespace) . . . 1970
process_output(tool_activity, subtask, output_artifact) . . . 1970
store_artifact(namespace, artifact) . . . 1971
validate*artifact_storages(*, artifact_storage) . . . 1972
11.5.13 mixins . . . 2008
actions_subtask_origin_mixin . . . 2008
activity_mixin . . . 2010
artifact_file_output_mixin . . . 2017
exponential_backoff_mixin . . . 2018
futures_executor_mixin . . . 2019
rule_mixin . . . 2021
runnable_mixin . . . 2022
serializable_mixin . . . 2023
singleton_mixin . . . 2031
11.5.14 observability . . . 2031
enter() . . . 2032
exit(exc_type, exc_value, exc_traceback) . . . 2032
get_global_driver()staticmethod . . . 2033
get_span_id()staticmethod . . . 2033
observe(call)staticmethod . . . 2033
set_global_driver(driver)staticmethod . . . 2033
11.5.15 rules . . . 2036
str() . . . 2036
to_text()abstractmethod . . . 2036
to_text() . . . 2037
to_text() . . . 2037
attrs_post_init() . . . 2038
11.5.16 schemas . . . 2040
Meta . . . 2048
_get_field_for_type(field_type, serialization_key=None, types_overrides=None)classmethod . . . 2048
Parameters . . . 2048
_get_field_type_info(field_type)classmethod . . . 2049

Parameters . . . 2049
_handle_list(list_type, *, optional, serialization_key=None)classmethod . . . 2050
Parameters . . . 2050
Returns . . . 2050
_handle_union(union_type, *, optional, serialization_key=None)classmethod . . . 2051
Parameters . . . 2051
Returns . . . 2051
_is_enum(field_type)classmethod . . . 2052
_is_list_sequence(field_type)classmethod . . . 2052
_is_union(field_type)classmethod . . . 2053
_resolve_types(attrs_cls, types_override=None)classmethod . . . 2053
Parameters . . . 2053
from_attrs_cls(attrs_cls, *, types_overrides=None, serializable_overrides=None)classmethod 2056
Parameters . . . 2056
_deserialize(value, attr, data, **kwargs) . . . 2058
_serialize(value, attr, obj, **kwargs) . . . 2058
_validate(value) . . . 2058
init(inner_class, types_overrides=None, **kwargs) . . . 2061
_dump(obj, *, update_fields=True, **kwargs) . . . 2061
_load(data, *, partial=None, unknown=None, **kwargs) . . . 2062
dump(obj, *, many=None, **kwargs) . . . 2062
get_data_type(data) . . . 2063
get_obj_type(obj) . . . 2063
load(data, *, many=None, partial=None, unknown=None, **kwargs) . . . 2063
validate(data, *, many=None, partial=None) . . . 2064
_deserialize(value, attr, data, **kwargs) . . . 2065
_serialize(value, attr, obj, **kwargs) . . . 2065
Parameters . . . 2065
init(fields, *, reverse_serialize_candidates=False, **kwargs) . . . 2067
_deserialize(value, attr=None, data=None, **kwargs) . . . 2067
Parameters . . . 2067
Raises . . . 2068
_serialize(value, attr, obj, **kwargs) . . . 2068
Parameters . . . 2068
Raises . . . 2068
11.5.17 structures . . . 2099
attrs_post_init() . . . 2101
_init_task() . . . 2101
add_task(task) . . . 2102
add_tasks(*tasks) . . . 2102
try_run(*args) . . . 2102
validate*fail_fast(*, fail_fast) . . . 2102
validate*prompt_driver(*, prompt_driver) . . . 2103
validate*tasks(*, tasks) . . . 2103
__run_from_task(task) . . . 2104
add_task(task) . . . 2105
context(task) . . . 2105
insert_task(parent_task, task) . . . 2105
try_run(*args) . . . 2106
add(other) . . . 2111
attrs_post_init() . . . 2111
add_task(task)abstractmethod . . . 2111
add_tasks(*tasks) . . . 2111
after_run() . . . 2111

before_run(args) . . . 2112
context(task) . . . 2112
find_task(task_id) . . . 2112
is_finished() . . . 2113
is_running() . . . 2113
resolve_relationships() . . . 2113
run(*args) . . . 2113
run_stream(*args, event_types=None) . . . 2114
try_find_task(task_id) . . . 2114
try_run(*args)abstractmethod . . . 2114
__link_task_to_children(task, child_tasks) . . . 2118
__link_task_to_parents(task, parent_tasks) . . . 2119
__remove_old_parent_child_relationships(parent_tasks, child_tasks) . . . 2119
add_task(task) . . . 2119
context(task) . . . 2120
insert_task(parent_tasks, task, child_tasks, *, preserve_relationship=False) . . . 2120
insert_tasks(parent_tasks, tasks, child_tasks, *, preserve_relationship=False) . . . 2120
Parameters . . . 2120
order_tasks() . . . 2122
to_graph() . . . 2122
try_run(*args) . . . 2122
11.5.18 tasks . . . 2147
__init_from_artifact(artifact) . . . 2156
Parameters . . . 2156
Returns . . . 2156
__init_from_prompt(value) . . . 2157
__parse_actions(actions_matches) . . . 2157
__process_action_object(action_object) . . . 2158
__validate_action(action) . . . 2159
_process_task_input(task_input) . . . 2159
actions_to_dicts() . . . 2160
actions_to_json() . . . 2160
add_to_prompt_stack(stack) . . . 2160
after_run() . . . 2161
attach_to(parent_task) . . . 2162
before_run() . . . 2162
default_generate_assistant_subtask_template(subtask) . . . 2163
default_generate_user_subtask_template(subtask) . . . 2163
run_action(action) . . . 2163
run_actions(actions) . . . 2163
try_run() . . . 2164
Attributes . . . 2164
try_run() . . . 2165
try_run() . . . 2165
after_run() . . . 2166
before_run() . . . 2166
Attributes . . . 2166
_get_negative_prompts() . . . 2169
_get_prompts(prompt) . . . 2169
_read_from_file(path) . . . 2169
add_child(child) . . . 2171
add_parent(parent) . . . 2171
add_to_prompt_stack(stack)abstractmethod . . . 2171
attach_to(parent_task) . . . 2171

State . . . 2176
attrs_post_init() . . . 2177
lshift(other) . . . 2177
rshift(other) . . . 2177
str() . . . 2177
add_child(child) . . . 2177
add_children(children) . . . 2178
add_parent(parent) . . . 2178
add_parents(parents) . . . 2178
after_run() . . . 2178
before_run() . . . 2178
can_run() . . . 2179
is_finished() . . . 2179
is_pending() . . . 2179
is_running() . . . 2179
is_skipped() . . . 2180
preprocess(structure) . . . 2180
reset() . . . 2180
run(*args) . . . 2180
try_run()abstractmethod . . . 2181
after_run() . . . 2182
before_run() . . . 2182
try_run() . . . 2183
try_run() . . . 2184
try_run() . . . 2184
Attributes . . . 2185
try_run() . . . 2188
Attributes . . . 2188
try_run() . . . 2192
add_to_prompt_stack(stack) . . . 2195
after_run() . . . 2195
attach_to(parent_task) . . . 2195
before_run() . . . 2196
default_generate_assistant_subtask_template(subtask) . . . 2196
default_generate_user_subtask_template(subtask) . . . 2196
try_run() . . . 2196
Attributes . . . 2196
try_run() . . . 2199
attrs_post_init() . . . 2208
_process_task_input(task_input) . . . 2208
actions_schema() . . . 2208
add_subtask(subtask) . . . 2209
after_run() . . . 2209
before_run() . . . 2209
default_generate_assistant_subtask_template(subtask) . . . 2210
default*generate_system_template*() . . . 2210
default_generate_user_subtask_template(subtask) . . . 2210
default_run_actions_subtasks(subtask_input) . . . 2210
default_run_output_schema_validation_subtasks(subtask_input) . . . 2211
find_memory(memory_name) . . . 2211
find_subtask(subtask_id) . . . 2212
find_tool(tool_name) . . . 2212
preprocess(structure) . . . 2212
set_default_tools_memory(memory) . . . 2212

try_run() . . . 2213
validate*output_schema(*, output_schema) . . . 2213
validate*tools(*, tools) . . . 2213
try_run() . . . 2214
Attributes . . . 2214
_process_task_input(task_input) . . . 2216
try_run() . . . 2216
try_run() . . . 2217
try_run() . . . 2218
attrs_post_init() . . . 2220
actions_schema() . . . 2221
add_subtask(subtask) . . . 2221
default*generate_system_template()* . . . 2221
find_memory(memory_name) . . . 2221
find_subtask(subtask_id) . . . 2221
find_tool(tool_name) . . . 2222
preprocess(structure) . . . 2222
set_default_tools_memory(memory) . . . 2222
try_run() . . . 2222
try_run() . . . 2223
Attributes . . . 2224
try_run() . . . 2227
11.5.19 tokenizers . . . 2312
count_tokens(text) . . . 2314
count_tokens(text) . . . 2314
attrs_post_init() . . . 2317
_default_max_input_tokens() . . . 2317
_default_max_output_tokens() . . . 2318
count_input_tokens_left(text) . . . 2318
count_output_tokens_left(text) . . . 2318
count_tokens(text)abstractmethod . . . 2319
max_input_tokens() . . . 2319
max_output_tokens() . . . 2319
count_tokens(text) . . . 2320
count_tokens(text) . . . 2320
client() . . . 2321
count_tokens(text) . . . 2321
count_tokens(text) . . . 2322
count_tokens(text) . . . 2323
_default_max_input_tokens() . . . 2327
_default_max_output_tokens() . . . 2327
count_tokens(text, model=None) . . . 2327
count_tokens(text) . . . 2329
count_tokens(text) . . . 2330
11.5.20 tools . . . 2348
transcribe_audio_from_disk(params) . . . 2349
transcribe_audio_from_memory(params) . . . 2350
Attributes . . . 2351
attrs_post_init() . . . 2358
activity_schemas() . . . 2358
after_run(activity, subtask, action, value) . . . 2359
are_requirements_met(requirements_path) . . . 2359
before_run(activity, subtask, action) . . . 2359
find_input_memory(memory_name) . . . 2360

install_dependencies(env=None) . . . 2360
run(activity, subtask, action) . . . 2360
schema() . . . 2361
to_native_tool_name(activity) . . . 2361
Parameters . . . 2361
Returns . . . 2361
tool_dir() . . . 2362
try_run(activity, subtask, action, value) . . . 2362
validate() . . . 2362
validate*output_memory(*, output_memory) . . . 2363
calculate(params) . . . 2364
attrs_post_init() . . . 2368
del() . . . 2369
build_image(tool) . . . 2369
container_name(tool) . . . 2369
default_docker_client() . . . 2369
dependencies() . . . 2370
execute_code(params) . . . 2370
execute_code_in_container(filename, code) . . . 2370
execute_command(params) . . . 2371
execute_command_in_container(command) . . . 2371
image_name(tool) . . . 2372
install_dependencies(env=None) . . . 2372
remove_existing_container(name) . . . 2372
validate*docker_client(*, docker_client) . . . 2373
add_timedelta(timedelta_kwargs, iso_datetime=None) . . . 2375
get_current_datetime() . . . 2375
get_datetime_diff(start_datetime, end_datetime) . . . 2375
get_relative_datetime(params) . . . 2376
Attributes . . . 2377
_create_smtp_client(smtp_host, smtp_port) . . . 2384
retrieve(params) . . . 2384
send(params) . . . 2385
Attributes . . . 2386
extract(params) . . . 2387
Attributes . . . 2388
list_files_from_disk(params) . . . 2391
load_files_from_disk(params) . . . 2392
save_content_to_file(params) . . . 2392
save_memory_artifacts_to_disk(params) . . . 2393
Attributes . . . 2394
attrs_post_init() . . . 2397
__extract_schema_from_ref(schema, schema_ref) . . . 2397
_create_activity_handler(activity_name, description, activity_schema) . . . 2398
_get_schema() . . . 2398
_init_activities() . . . 2399
_map_openapi_type_to_python(openapi_type, schema_info=None) . . . 2399
_parse_schema(schema) . . . 2399
_run_activity(activity_name, params) . . . 2400
query_image_from_disk(params) . . . 2402
query_images_from_memory(params) . . . 2403
Attributes . . . 2405
_generate_inpainting(prompt, negative_prompt, image_artifact, mask_artifact) . . . 2408
image_inpainting_from_file(params) . . . 2409

image_inpainting_from_memory(params) . . . 2409
Attributes . . . 2411
_generate_outpainting(prompt, negative_prompt, image_artifact, mask_artifact) . . . 2414
image_outpainting_from_file(params) . . . 2415
image_outpainting_from_memory(params) . . . 2415
Attributes . . . 2417
generate_image(params) . . . 2418
Attributes . . . 2419
summarize(params) . . . 2421
query(params) . . . 2423
Attributes . . . 2424
search(params) . . . 2425
Attributes . . . 2426
_build_url(base_url, path=None, path_params=None) . . . 2433
delete(params) . . . 2433
get(params) . . . 2434
patch(params) . . . 2435
post(params) . . . 2436
put(params) . . . 2437
execute_query(params) . . . 2439
Attributes . . . 2439
run_structure(params) . . . 2441
provide_output(params) . . . 2442
Attributes . . . 2442
text_to_speech(params) . . . 2444
Attributes . . . 2444
_generate_variation(prompt, negative_prompt, image_artifact) . . . 2447
image_variation_from_file(params) . . . 2448
image_variation_from_memory(params) . . . 2448
Attributes . . . 2449
search(params) . . . 2452
get_content(params) . . . 2453
search(values) . . . 2454
11.5.21 utils . . . 2567
Attributes . . . 2568
ChatPrompt . . . 2572
_call_handle_output(text, *, stream=False) . . . 2573
_has_streaming_tasks() . . . 2573
default_handle_input(prompt_prefix) . . . 2573
default_handle_output(text, *, stream=False) . . . 2573
start() . . . 2573
run(command) . . . 2575
str() . . . 2576
lines() . . . 2576
prompt_stack() . . . 2576
validate_memory(attribute, value) . . . 2577
Attributes . . . 2577
enter() . . . 2580
exit(exc_type, exc_value, exc_traceback) . . . 2580
_to_artifact(value) . . . 2581
event_listener() . . . 2581
observability() . . . 2581
render(**kwargs) . . . 2582
render_from_string(value, **kwargs) . . . 2582

schema() . . . 2583
validate(manifest) . . . 2583
run(code) . . . 2584
Attributes . . . 2584
run(*args) . . . 2585
__render_task(task) . . . 2587
__render_tasks(tasks) . . . 2588
to_url() . . . 2588
Returns . . . 2588
add_tokens(new_tokens) . . . 2589
Parameters . . . 2589
Parameters . . . 2590
Parameters . . . 2591
Returns . . . 2591
Parameters . . . 2593
Parameters . . . 2594
Returns . . . 2594
Parameters . . . 2595
Returns . . . 2595
Parameters . . . 2597
11.6 Agents . . . 2634
11.6.1 Overview . . . 2634
Agent Tools . . . 2634
Agent Input . . . 2634
11.7 Tools in Gen AI Builder . . . 2677
11.7.1 Overview . . . 2677
11.7.2 Custom Tools . . . 2677
11.7.3 Tool Output and Task Memory . . . 2677
11.7.4 Using Tools in Pipelines . . . 2677
Custom Tools . . . 2678
Official Tools . . . 2681
My First Post, A Journey., . . . 2689

Gen AI Builder is a Python framework designed to simplify the development of generative AI applications. It offers a set of straightforward, flexible abstractions for working with areas such as Large Language Models (LLMs), Retrieval-Augmented Generation (RAG), and much more.

1. **Technology-agnostic**: Gen AI Builder is designed to work with any capable LLM, data store, and backend through the abstraction of Drivers.
2. **Minimal prompt engineering**: It's much easier to reason about code written in Python, not natural languages. Gen AI Builder aims to default to Python in most cases unless absolutely necessary.

CHAPTER 1

Why Consider a Framework?

If you're just beginning to experiment with generative AI, it's often best to start by building simple applications from scratch. However, as your project grows in complexity, a framework like Gen AI Builder can provide structure and best practices that help you maintain clarity and scalability.

Gen AI Builder strives to be transparent about the underlying AI techniques it uses. Nevertheless, having some foundational knowledge of LLMs and related technology will be helpful when building more sophisticated applications.

CHAPTER 2

Prerequisites

2.1 1. OpenAI API Key

By default, Gen AI Builder uses OpenAI for LLM interactions. We'll learn how to change this via *Prompt Drivers* later. For now:

1. Obtain an OpenAI API key .
2. Set it as an environment variable:

```
export OPENAI_API_KEY="your-api-key"
```

Prompt Drivers are model agnostic, letting you switch out LLM providers without changing application code.

> **Tip**
>
> To securely and easily manage environment variables, check out tools like python-dotenv or mise .

2.2 2. Install Gen AI Builder

Gen AI Builder can be installed via multiple methods. Below are two popular approaches:

2.3 Using Uv

uv is a fast, user-friendly dependency manager. If you've used poetry , `uv` is a similar project but with a focus on simplicity and speed.

1. *Install `uv* <https://docs.astral.sh/uv/getting-started/installation/>`_ .
2. Initialize a new project:

```
uv init griptape-quickstart
```

3. Change into the new directory:

```
cd griptape-quickstart
```

4. Add Gen AI Builder as a dependency:

```
uv add "griptape[all]"
```

5. A virtual environment will be automatically created. Activate it:

```
source .venv/bin/activate
```

> **Note**
>
> If you use another shell (like *fish*), this command may differ.

> **Tip**
>
> If you're using mise , you can configure it to automatically activate your virtual environment .

2.4 Using Pip

If `uv` isn't your style, you can always install Gen AI Builder using `pip` :

```
pip install "griptape[all]" -U
```

2.5 Installing Optional Dependencies

Gen AI Builder provides optional dependencies that enable specific features and third-party integrations. If you're new to Gen AI Builder, consider installing `[all]` to unlock its full functionality. This ensures you have everything you need right from the start.

However, for a more streamlined setup or if you only need certain features, you have two primary options:

1. **Core Dependencies**: These provide the minimal, foundational set of libraries for Gen AI Builder to run most default features.
2. **Optional Dependencies**: These are additional, vendor-specific drivers (e.g., for Anthropic or Pinecone). Any driver requiring an extra will indicate this in the documentation.

For a core-only installation:

```
uv add griptape
```

To install specific optional dependencies such as Anthropic and Pinecone drivers—use:

```
uv add "griptape[drivers-prompt-anthropic,drivers-vector-pinecone]"
```

To see all available optional dependencies, see Gen AI Builder's pyproject.toml .

CHAPTER 3

Prompt Task

With everything set up, let's start building! One of Gen AI Builder's core abstractions is a *Prompt Task* . We'll start with a *Prompt Task* to prompt the LLM with some text.

Create a file named `app.py` and add:

```
from griptape.tasks import PromptTask

task = PromptTask()

output = task.run("Hello there!")

print(output)
```

Then run:

```
python app.py
```

> **Tip**
>
> Whenever you see a "Logs" tab in the Gen AI Builder documentation, you can click it to view logs and outputs.

If you got a response from the model, congrats! You've created your first generative AI application with Gen AI Builder. This introductory guide sticks to the *Prompt Task* , but Gen AI Builder supports a wide range of add_tasks(*tasks) . Explore them to see what else you can do.

CHAPTER 4

Changing Models

Gen AI Builder's "Prompt Drivers" make switching model providers simple. For instance, to use Anthropic , follow the same steps you used for the OpenAI key, but this time set an *Anthropic API key* :

```
export ANTHROPIC_API_KEY="your-api-key"
```

Then import *Anthropic Prompt Driver* and pass it to your Task:

```
from griptape.drivers.prompt.anthropic import AnthropicPromptDriver
from griptape.tasks import PromptTask

task = PromptTask(prompt_driver=AnthropicPromptDriver(model="claude-3-7-sonnet-latest
↪"))

task.run("Hello there!")
```

Run the script again to see a similar output. Notice you didn't have to change any of the application-level logic; you only changed the driver.

For the remainder of this guide, we'll revert to the *OpenAI Chat Prompt Driver* :

```
from griptape.drivers.prompt.openai_chat_prompt_driver import OpenAiChatPromptDriver
from griptape.tasks import PromptTask

task = PromptTask(prompt_driver=OpenAiChatPromptDriver(model="gpt-4.1"))

task.run("Hello there!")
```

If you'd like to keep using Anthropic, be sure to adjust code accordingly.

CHAPTER 5

Templating

Gen AI Builder uses the popular Jinja templating engine to help create dynamic prompts. Let's adapt our Prompt Task to utilize a Jinja template:

```
from griptape.drivers.prompt.openai_chat_prompt_driver import OpenAiChatPromptDriver
from griptape.tasks import PromptTask

task = PromptTask(
    input="User said: {{ args[0] }}",
    prompt_driver=OpenAiChatPromptDriver(model="gpt-4.1"),
)

task.run("Hi there!")
```

Here, `{{ args }}` is rendered at runtime, allowing you to dynamically insert data into the prompt.

For static, key-value data, we can add `context` to our Task:

```
from griptape.drivers.prompt.openai_chat_prompt_driver import OpenAiChatPromptDriver
from griptape.tasks import PromptTask

task = PromptTask(
    input="You are speaking to: {{ user_name }}. User said: {{ args[0] }}",
    prompt_driver=OpenAiChatPromptDriver(model="gpt-4.1"),
    context={"user_name": "Collin"},
)

task.run("Hi there!")
```

These two methods allow you to build more versatile, data-driven prompts. Next, we'll discuss steering the LLM's output using Gen AI Builder's Rule system.

CHAPTER 6

Rules

LLMs can be unpredictable. Getting them to follow specific instructions can be challenging. Gen AI Builder addresses this by providing *Rules and Rulesets* , which offer a structured approach to steering the model's output.

For example, we can create a simple Rule that gives the LLM a name:

```
from griptape.drivers.prompt.openai_chat_prompt_driver import OpenAiChatPromptDriver
from griptape.rules import Rule
from griptape.tasks import PromptTask

task = PromptTask(
    input="You are speaking to: {{ user_name }}. User said: {{ args[0] }}",
    prompt_driver=OpenAiChatPromptDriver(model="gpt-4.1"),
    context={"user_name": "Collin"},
    rules=[Rule("Your name is Oswald.")],
)

task.run("Hi there, whats your name?")
```

Tip

While we typically recommend you stick to Rule-based steering, you can always *override the system prompt* with a custom one. Though pay close attention to the discussed tradeoffs that come with this approach.

Multiple rules can be grouped into `Ruleset` s. Let's refactor our code to use more than one Ruleset:

```
from griptape.drivers.prompt.openai_chat_prompt_driver import OpenAiChatPromptDriver
from griptape.rules import Rule, Ruleset
from griptape.tasks import PromptTask

task = PromptTask(
    input="You are speaking to: {{ user_name }}. User said: {{ args[0] }}",
```

(continues on next page)

(continued from previous page)

```
    prompt_driver=OpenAiChatPromptDriver(model="gpt-4.1"),
    context={"user_name": "Collin"},
    rulesets=[
        Ruleset(
            name="Backstory",
            rules=[
                Rule("Your name is Oswald."),
            ],
        ),
        Ruleset(
            name="Behavior",
            rules=[
                Rule("Introduce yourself at the start of the conversation."),
            ],
        ),
    ],
)

task.run("Hi there!")
```

Gen AI Builder also includes a specialized Rule, *JsonSchemaRule* , which inserts a JSON Schema into the system prompt, ensuring the LLM attempts to return structured, machine-readable data:

```
import json
from typing import TYPE_CHECKING, cast

from griptape.drivers.prompt.openai_chat_prompt_driver import OpenAiChatPromptDriver
from griptape.rules import JsonSchemaRule, Rule, Ruleset
from griptape.tasks import PromptTask

if TYPE_CHECKING:
    from griptape.artifacts.text_artifact import TextArtifact

task = PromptTask(
    input="You are speaking to: {{ user_name }}. User said: {{ args[0] }}",
    prompt_driver=OpenAiChatPromptDriver(model="gpt-4.1"),
    context={"user_name": "Collin"},
    rulesets=[
        Ruleset(
            name="Backstory",
            rules=[
                Rule("Your name is Oswald."),
            ],
        ),
        Ruleset(
            name="Behavior",
            rules=[
                Rule("Introduce yourself at the start of the conversation."),
            ],
        ),
        Ruleset(
            name="Output Format",
```

(continues on next page)

(continued from previous page)

```
            rules=[
                JsonSchemaRule(
                    {
                        "$schema": "https://json-schema.org/draft/2020-12/schema",
                        "type": "object",
                        "properties": {
                            "answer": {"type": "string"},
                        },
                        "required": ["answer"],
                    }
                )
            ],
        ),
    ],
)

task_output_value = cast("TextArtifact", task.run("Hi there!")).value

print(json.dumps(json.loads(task_output_value), indent=2))
```

Note

The *typing.cast* usage is optional, but helps satisfy our type-checker, pyright . We're actively working to improve Gen AI Builder's type hints to avoid clunky casts like this.

Now let's explore an even more robust way to enforce structured outputs.

CHAPTER 7

Structured Output

While a `JsonSchemaRule` can encourage the LLM to return structured data, it doesn't provide guarantees, especially for complex schemas. For stronger enforcement, Gen AI Builder's Prompt Drivers support *Structured Output* .

Structured Output uses the LLM provider's built-in API features to produce outputs in a well-defined schema. It also simplifies parsing, so you can work directly with Python objects rather than manual JSON parsing.

Let's update our Task to use Structured Output:

```
from typing import cast

from pydantic import BaseModel

from griptape.drivers.prompt.openai_chat_prompt_driver import OpenAiChatPromptDriver
from griptape.rules import Rule, Ruleset
from griptape.tasks import PromptTask

class Output(BaseModel):
    answer: str

task = PromptTask(
    input="You are speaking to: {{ user_name }}. User said: {{ args[0] }}",
    prompt_driver=OpenAiChatPromptDriver(model="gpt-4.1"),
    context={"user_name": "Collin"},
    rulesets=[
        Ruleset(
            name="Backstory",
            rules=[
                Rule("Your name is Oswald."),
            ],
        ),
        Ruleset(
            name="Behavior",
            rules=[
```

(continues on next page)

(continued from previous page)

```
                Rule("Introduce yourself at the start of the conversation."),
            ],
        ),
    ],
    output_schema=Output,
)

task_output_value = cast("Output", task.run("Hi there!").value)

print(task_output_value.answer)
```

The LLM response is not only formatted correctly, but automatically parsed into a Python object.

Tip

You can use either pydantic or schema to define your schemas. We recommend pydantic for its ease of use.

Now that we can generate structured responses, let's make our Task more conversational using Gen AI Builder's memory features.

CHAPTER 8

Conversation Memory

By default, each LLM call is stateless. If you want the model to recall previous interactions, use *Conversation Memory*. It tracks Task run history, providing context for the current run.

```
from pydantic import BaseModel

from griptape.drivers.prompt.openai_chat_prompt_driver import OpenAiChatPromptDriver
from griptape.memory.structure import ConversationMemory
from griptape.rules import Rule, Ruleset
from griptape.tasks import PromptTask

class Output(BaseModel):
    answer: str

task = PromptTask(
    input="You are speaking to: {{ user_name }}. User said: {{ args[0] }}",
    prompt_driver=OpenAiChatPromptDriver(model="gpt-4.1"),
    context={"user_name": "Collin"},
    rulesets=[
        Ruleset(
            name="Backstory",
            rules=[
                Rule("Your name is Oswald."),
            ],
        ),
        Ruleset(
            name="Behavior",
            rules=[
                Rule("Introduce yourself at the start of the conversation."),
            ],
        ),
    ],
    output_schema=Output,
```

(continues on next page)

(continued from previous page)

```
    conversation_memory=ConversationMemory(),
)

task.run("Hi there, my name is Collin!")
task.run("Do you remember my name?")
```

Tip

You can customize where the conversation is stored by providing a *Conversation Memory Drivers* .

With that set up, the LLM will now remember our previous exchanges. Let's move on to enabling the model to perform external actions through Tools.

CHAPTER 9

Tools

LLMs themselves cannot browse the internet or perform external actions. You can enable these abilities via *Tools* . A "Tool" is simply a Python class, and any method marked with the `@activity` decorator becomes available to the LLM.

Gen AI Builder comes with a collection of built-in Tools. Below is an example that uses a web search and scrape Tool:

```
from pydantic import BaseModel
from rich.pretty import pprint

from griptape.drivers.prompt.openai_chat_prompt_driver import OpenAiChatPromptDriver
from griptape.drivers.web_search.duck_duck_go import DuckDuckGoWebSearchDriver
from griptape.memory.structure import ConversationMemory
from griptape.rules import Rule, Ruleset
from griptape.tasks import PromptTask
from griptape.tools import WebScraperTool, WebSearchTool

class Feature(BaseModel):
    name: str
    description: str
    emoji: str

class Output(BaseModel):
    answer: str
    key_features: list[Feature]

task = PromptTask(
    id="project-research",
    input="You are speaking to: {{ user_name }}. User said: {{ args[0] }}",
    prompt_driver=OpenAiChatPromptDriver(model="gpt-4.1"),
    context={"user_name": "Collin"},
    rulesets=[
        Ruleset(
```

(continues on next page)

(continued from previous page)

```
            name="Backstory",
            rules=[
                Rule("Your name is Oswald."),
                Rule("You run Oswalds Open Source, a company for helping people learn␣
↪about open source."),
            ],
        ),
        Ruleset(
            name="Behavior",
            rules=[
                Rule("Introduce yourself at the start of the conversation."),
            ],
        ),
    ],
    output_schema=Output,
    tools=[
        WebSearchTool(
            web_search_driver=DuckDuckGoWebSearchDriver(),
        ),
        WebScraperTool(),
    ],
    conversation_memory=ConversationMemory(),
)

output = task.run("Tell me about the python framework Griptape.")

pprint(output.value)
```

Tools are a powerful method for extending the LLM's capabilities. Consider *writing your own Tools* to enable more specialized functionality.

At this point, we have a Task that can handle real-world operations. Now let's look at orchestrating multiple Tasks.

CHAPTER 10

Structures

So far, we've only worked with individual Tasks. As your app becomes more complex, you may want to organize multiple Tasks that execute in sequence or in parallel.

Gen AI Builder provides three Structures:

- **Agent**: A lightweight wrapper around a single `PromptTask`. Great for quick usage but still under the hood is just a `PromptTask`.
- **Pipeline**: Executes Tasks in a specific, linear order.
- **Workflow**: Creates highly parallel DAGs, allowing you to fan out and converge Tasks.

Of the three, `Workflow` is generally the most versatile. `Agent` and `Pipeline` can be handy in certain scenarios but are less frequently needed if you're comfortable just orchestrating Tasks directly.

Let's place our `project-research` Task into a `Workflow` so multiple Tasks can run in parallel:

```
from pydantic import BaseModel
from rich.pretty import pprint

from griptape.drivers.prompt.openai_chat_prompt_driver import OpenAiChatPromptDriver
from griptape.drivers.web_search.duck_duck_go import DuckDuckGoWebSearchDriver
from griptape.rules import Rule, Ruleset
from griptape.structures import Workflow
from griptape.tasks import PromptTask
from griptape.tools import WebScraperTool, WebSearchTool

class Feature(BaseModel):
    name: str
    description: str
    emoji: str

class Output(BaseModel):
    answer: str
    key_features: list[Feature]
```

(continues on next page)

(continued from previous page)

```
projects = ["django", "flask", "fastapi", "litestar"]

workflow = Workflow(
    rulesets=[
        Ruleset(
            name="Backstory",
            rules=[
                Rule("Your name is Oswald."),
                Rule("You run Oswalds Open Source, a company for helping people learn␣
↪about open source."),
            ],
        ),
    ],
)

for project in projects:
    task = PromptTask(
        id=f"project-research-{project}",
        input="Tell me about the open source project: {{ project }}.",
        prompt_driver=OpenAiChatPromptDriver(model="gpt-4.1"),
        context={"project": project},
        output_schema=Output,
        tools=[
            WebSearchTool(
                web_search_driver=DuckDuckGoWebSearchDriver(),
            ),
            WebScraperTool(),
        ],
    )
    workflow.add_task(task)

workflow.run()

for output in workflow.outputs:
    pprint(output.value)
```

10.1 Directly Using PromptSummaryEngine

Engines can be used standalone or integrated into Tasks. To illustrate, let's summarize the outputs of our Workflow:

```
from pydantic import BaseModel

from griptape.artifacts import ListArtifact
from griptape.drivers.prompt.openai_chat_prompt_driver import OpenAiChatPromptDriver
from griptape.drivers.web_search.duck_duck_go import DuckDuckGoWebSearchDriver
from griptape.engines import PromptSummaryEngine
from griptape.rules import Rule, Ruleset
from griptape.structures import Workflow
from griptape.tasks import PromptTask
from griptape.tools import WebScraperTool, WebSearchTool
```

(continues on next page)

(continued from previous page)

```
class Feature(BaseModel):
    name: str
    description: str
    emoji: str

class Output(BaseModel):
    answer: str
    key_features: list[Feature]

projects = ["django", "flask", "fastapi", "litestar"]

workflow = Workflow(
    rulesets=[
        Ruleset(
            name="Backstory",
            rules=[
                Rule("Your name is Oswald."),
                Rule("You run Oswalds Open Source, a company for helping people learn
↪about open source."),
            ],
        ),
    ],
)

for project in projects:
    task = PromptTask(
        id=f"project-research-{project}",
        input="Tell me about the open source project: {{ project }}.",
        prompt_driver=OpenAiChatPromptDriver(model="gpt-4.1"),
        context={"project": project},
        output_schema=Output,
        tools=[
            WebSearchTool(
                web_search_driver=DuckDuckGoWebSearchDriver(),
            ),
            WebScraperTool(),
        ],
    )
    workflow.add_tasks(task)

workflow.run()

summary_engine = PromptSummaryEngine()
summary = summary_engine.summarize_artifacts(ListArtifact(workflow.outputs))

print(summary.value)
```

This produces a concise summary. But one strength of a Workflow is its ability to fan out and converge results. Let's remove the direct `PromptSummaryEngine` usage and use a built-in summary Task instead.

10.2 TextSummaryTask

The *TextSummaryTask* leverages the `PromptSummaryEngine` behind the scenes. We'll have our project-research Tasks converge into one summary Task:

```
from pydantic import BaseModel

from griptape.drivers.prompt.openai_chat_prompt_driver import OpenAiChatPromptDriver
from griptape.drivers.web_search.duck_duck_go import DuckDuckGoWebSearchDriver
from griptape.rules import Rule, Ruleset
from griptape.structures import Workflow
from griptape.tasks import PromptTask, TextSummaryTask
from griptape.tools import WebScraperTool, WebSearchTool

class Feature(BaseModel):
    name: str
    description: str
    emoji: str

class Output(BaseModel):
    answer: str
    key_features: list[Feature]

projects = ["django", "flask", "fastapi", "litestar"]

workflow = Workflow(
    rulesets=[
        Ruleset(
            name="Backstory",
            rules=[
                Rule("Your name is Oswald."),
                Rule("You run Oswalds Open Source, a company for helping people learn␣
↪about open source."),
            ],
        ),
    ],
)

for project in projects:
    task = PromptTask(
        id=f"project-research-{project}",
        input="Tell me about the open source project: {{ project }}.",
        prompt_driver=OpenAiChatPromptDriver(model="gpt-4.1"),
        context={"project": project},
        output_schema=Output,
        tools=[
            WebSearchTool(
                web_search_driver=DuckDuckGoWebSearchDriver(),
            ),
            WebScraperTool(),
        ],
        child_ids=["summary"],
    )
```

(continues on next page)

(continued from previous page)

```
    workflow.add_tasks(task)
workflow.add_task(TextSummaryTask(input="{{ parents_output_text }}", id="summary"))

workflow.run()

print(workflow.output)
```

Finally, let's visualize our Workflow:

```
from pydantic import BaseModel

from griptape.drivers.prompt.openai_chat_prompt_driver import OpenAiChatPromptDriver
from griptape.drivers.web_search.duck_duck_go import DuckDuckGoWebSearchDriver
from griptape.rules import Rule, Ruleset
from griptape.structures import Workflow
from griptape.tasks import PromptTask, TextSummaryTask
from griptape.tools import WebScraperTool, WebSearchTool
from griptape.utils import StructureVisualizer

class Feature(BaseModel):
    name: str
    description: str
    emoji: str

class Output(BaseModel):
    answer: str
    key_features: list[Feature]

projects = ["django", "flask", "fastapi", "litestar"]

workflow = Workflow(
    rulesets=[
        Ruleset(
            name="Backstory",
            rules=[
                Rule("Your name is Oswald."),
                Rule("You run Oswalds Open Source, a company for helping people learn␣
↪about open source."),
            ],
        ),
    ],
)
prompt_driver = OpenAiChatPromptDriver(model="gpt-4.1")

for project in projects:
    task = PromptTask(
        id=f"project-research-{project}",
        input="Tell me about the open source project: {{ project }}.",
        prompt_driver=prompt_driver,
        context={"project": project},
        output_schema=Output,
        tools=[
```

(continues on next page)

(continued from previous page)

```
            WebSearchTool(
                web_search_driver=DuckDuckGoWebSearchDriver(),
            ),
            WebScraperTool(),
        ],
        child_ids=["summary"],
    )
    workflow.add_tasks(task)
workflow.add_task(TextSummaryTask(input="{{ parents_output_text }}", id="summary"))

print(StructureVisualizer(workflow).to_url())
```

Visit the printed URL in your logs to view a diagram illustrating the Workflow's structure:

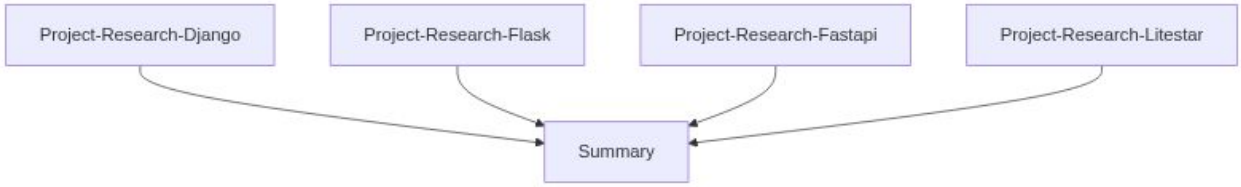

Fig. 1: Workflow Diagram

CHAPTER 11

Conclusion and Next Steps

That's it—you've built your first generative AI application with Gen AI Builder, exploring:

- Simple and advanced Task usage
- Dynamic prompt creation with Jinja templating
- Structured output for more reliable data handling
- Conversation memory for stateful interactions
- Tools for extending LLM capabilities
- Parallel Task orchestration with Workflows
- Text summarization using Engines

This is just the tip of the iceberg. For topic-based guides, continue exploring the Gen AI Builder documentation. For more detailed guides, demos, and advanced techniques, check out Gen AI Builder Tradeschool .

11.1 Data abstractions

11.1.1 Overview

Gen AI Builder provides several abstractions for working with data.

Artifacts are used for passing different types of data, such as text, lists, and blobs, between Gen AI Builder components.

Embedding Drivers are used to generate vector embeddings from text.

Loaders are used to load textual data from different sources.

Chunkers are used to split arbitrarily long text into chunks of certain token length.

Tokenizers are used to tokenize and detokenize text in order to track LLM token limits.

RAG Engines are used for implementing modular RAG pipelines.

Extraction Engines are used for extracting structured content.

Summary Engines are used for summarizing text content.

Vector Store Drivers are used to store and query vector databases.

Prompt Drivers are used to call LLM APIs.

Artifacts

Overview

** *Artifacts* ** are the core data structure in Gen AI Builder. They are used to encapsulate data and enhance it with metadata.

Text

TextArtifact s store textual data. They offer methods such as *token_count()* for counting tokens with a tokenizer, and *generate_embedding()* for creating text embeddings. You can also access the embedding via the *Embedding* property.

When `TextArtifact` s are returned from Tools, they will be stored in *Task Memory and Off Prompt* if the Tool has set `off_prompt=True` .

Blob

BlobArtifact s store binary large objects (blobs).

When `BlobArtifact` s are returned from Tools, they will be stored in *Task Memory and Off Prompt* if the Tool has set `off_prompt=True` .

Image

ImageArtifact s store image data. This includes binary image data along with metadata such as MIME type and dimensions. They are a subclass of *BlobArtifactStorage* .

Url

ImageUrlArtifact s store URLs pointing to resources.

Image Url

ImageUrlArtifact s store URLs pointing to images. They are a subclass of *ImageUrlArtifact* .

Audio

AudioArtifact s store audio content. This includes binary audio data and metadata such as format, and duration. They are a subclass of *BlobArtifactStorage* .

List

ListArtifact s store lists of Artifacts.

When `ListArtifact` s are returned from Tools, their elements will be stored in *Task Memory and Off Prompt* if the element is either a `TextArtifact` or a `BlobArtifact` and the Tool has set `off_prompt=True` .

Info

InfoArtifact s store small pieces of textual information. These are useful for conveying messages about the execution or results of an operation, such as “No results found” or “Operation completed successfully.”

JSON

JsonArtifact s store JSON-serializable data. Any data assigned to the `value` property is processed using `json.dumps(json.loads(value))`.

Error

ErrorArtifact s store exception information, providing a structured way to convey errors.

Action

ActionArtifact s represent actions taken by an LLM. Currently, the only supported action type is *ToolAction* , which is used to execute a *RAG Tool* .

Generic

GenericArtifact s provide a flexible way to pass data that does not fit into any other artifact category. While not generally recommended, they can be useful for specific use cases. For instance, see talking to a video, which demonstrates using a `GenericArtifact` to pass a Gemini-specific video file.

Chunkers

Overview

Chunkers are used to split arbitrarily long text into chunks of certain token length. Each chunker has a tokenizer, a max token count, and a list of default separators used to split up text into *TextArtifact* s. Different types of chunkers provide lists of separators for specific text shapes:

- `:ref:`TextChunker <TextChunker>`  : works on most texts.`
- `:ref:`PdfChunker <PdfChunker>`  : works on text from PDF docs.`
- `:ref:`MarkdownChunker <MarkdownChunker>`  works on markdown text.`

Here is how to use a chunker:

```
from griptape.chunkers import TextChunker
from griptape.tokenizers import OpenAiTokenizer

TextChunker(
    # set an optional custom tokenizer
    tokenizer=OpenAiTokenizer(model="gpt-4.1"),
    # optionally modify default number of tokens
    max_tokens=100,
).chunk("long text")
```

The most common use of a Chunker is to split up a long text into smaller chunks for inserting into a Vector Database when doing Retrieval Augmented Generation (RAG).

See *RagEngine* for more information on how to use Chunkers in RAG pipelines.

Loaders

Overview

Loaders are used to load data from sources and parse it into *Artifacts* s. Each loader can be used to load a single "source" with *load()* or multiple sources with *load_collection()* .

File

The following Loaders load a file using a *AmazonS3FileManagerDriver* and loads the resulting data into an *Artifacts* for the respective file type.

As a convenience, File Loaders also have a `save()` method that can save an Artifact to a destination.

```
from griptape.drivers.file_manager.local import LocalFileManagerDriver
from griptape.loaders import TextLoader

loader = TextLoader(file_manager_driver=LocalFileManagerDriver())

data = loader.load("tests/resources/test.txt")

data.value = data.value.replace("foo", "bar")

loader.save("tests/resources/test.txt", data)
```

Text

Loads text files into *TextArtifact* s:

```
import urllib.request
from pathlib import Path

from griptape.loaders import TextLoader

TextLoader().load("tests/resources/test.txt")

urllib.request.urlretrieve("https://example-files.online-convert.com/document/txt/
↪example.txt", "example.txt")

TextLoader().load(Path("example.txt"))

TextLoader().load_collection(["tests/resources/test.txt", Path("example.txt")])
```

PDF

Loads PDF files into *ListArtifact* s, where each element is a *TextArtifact* containing a page of the PDF:

```
import urllib.request
from pathlib import Path

from griptape.loaders import PdfLoader

urllib.request.urlretrieve("https://arxiv.org/pdf/1706.03762.pdf", "attention.pdf")
```

(continues on next page)

(continued from previous page)

```
#  Load a single PDF file

PdfLoader().load("attention.pdf")

#  You can also pass a Path object

PdfLoader().load(Path("attention.pdf"))

urllib.request.urlretrieve("https://arxiv.org/pdf/1706.03762.pdf", "CoT.pdf")

#  Load multiple PDF files

PdfLoader().load_collection([Path("attention.pdf"), Path("CoT.pdf")])
```

CSV

Loads CSV files into *ListArtifact* s, where each element is a *TextArtifact* containing a row of the CSV:

```
from pathlib import Path

from griptape.loaders import CsvLoader

#  Load a single CSV file

CsvLoader().load("tests/resources/cities.csv")

#  You can also pass a Path object

CsvLoader().load(Path("tests/resources/cities.csv"))

#  Load multiple CSV files

CsvLoader().load_collection([Path("tests/resources/cities.csv"), "tests/resources/
↪addresses.csv"])
```

Image

Loads images into *ImageArtifact* s:

```
from pathlib import Path

from griptape.loaders import ImageLoader

#  Load an image from disk

ImageLoader().load("tests/resources/mountain.png")

#  You can also pass a Path object

ImageLoader().load(Path("tests/resources/mountain.png"))
```

By default, the Image Loader will load images in their native format, but not all models work on all formats. To

normalize the format of Artifacts returned by the Loader, set the `format` field.

```
from pathlib import Path

from griptape.loaders import ImageLoader

#  Load a single image in BMP format

ImageLoader(format="bmp").load("tests/resources/mountain.png")

#  You can also pass a Path object

ImageLoader(format="bmp").load(Path("tests/resources/mountain.png"))

#  Load multiple images in BMP format

ImageLoader().load_collection([Path("tests/resources/mountain.png"), "tests/resources/
↪cow.png"])
```

Audio

Loads audio files into *AudioArtifact* s:

The Loader will load audio in its native format and populates the resulting Artifact's `format` field by making a best-effort guess of the underlying audio format using the `filetype` package.

```
from pathlib import Path

from griptape.loaders import AudioLoader

#  Load an image from disk

AudioLoader().load("tests/resources/sentences.wav")

#  You can also pass a Path object

AudioLoader().load(Path("tests/resources/sentences.wav"))
```

JSON

Loads JSON files into *JsonArtifact* s:

```
from pathlib import Path

from griptape.loaders import JsonLoader

#  Load an image from disk

JsonLoader().load("tests/resources/test.json")

#  You can also pass a Path object

JsonLoader().load(Path("tests/resources/test.json"))
```

Web

Scrapes web pages using a *BaseWebScraperDriver* and loads the resulting text into *TextArtifact* s.

```
from griptape.loaders import WebLoader

WebLoader().load("https://www.griptape.ai")

WebLoader().load_collection(["https://www.griptape.ai", "https://docs.griptape.ai"])
```

SQL

Loads data from a SQL database using a *AmazonRedshiftSqlDriver* and loads the resulting data into *ListArtifact* s, where each element is a *TextArtifact* containing a row of the SQL query.

```
from griptape.drivers.sql.sql_driver import SqlDriver
from griptape.loaders import SqlLoader

SqlLoader(sql_driver=SqlDriver(engine_url="sqlite:///:memory:")).load("SELECT foo, bar
↪")

SqlLoader(sql_driver=SqlDriver(engine_url="sqlite:///:memory:")).load_collection(
    ["SELECT foo, bar;", "SELECT fizz, buzz;"]
)
```

Email

Loads data from an imap email server into a *ListArtifact* s, where each element is a *TextArtifact* containing an email.

```
from griptape.loaders import EmailLoader

loader = EmailLoader(imap_url="an.email.server.hostname", username="username",␣
↪password="password")

loader.load(EmailLoader.EmailQuery(label="INBOX"))

loader.load_collection([EmailLoader.EmailQuery(label="INBOX"), EmailLoader.
↪EmailQuery(label="SENT")])
```

11.2 Prompt

11.2.1 Overview

Prompt Drivers are used by Gen AI Builder Structures to make API calls to the underlying LLMs. *OpenAI Chat* is the default prompt driver used in all structures.

You can instantiate drivers and pass them to structures:

```
from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.rules import Rule
from griptape.structures import Agent

agent = Agent(
```

(continues on next page)

(continued from previous page)

```
    prompt_driver=OpenAiChatPromptDriver(model="gpt-4.1", temperature=0.3),
    input="You will be provided with a tweet, and your task is to classify its␣
↪sentiment as positive, neutral, or negative. Tweet: {{ args[0] }}",
    rules=[Rule(value="Output only the sentiment.")],
)

agent.run("I loved the new Batman movie!")
```

Or use them independently:

```
from griptape.common import PromptStack
from griptape.drivers.prompt.openai import OpenAiChatPromptDriver

stack = PromptStack()

stack.add_system_message("You will be provided with Python code, and your task is to␣
↪calculate its time complexity.")
stack.add_user_message(
    """
    def foo(n, k):
        accum = 0
        for i in range(n):
            for l in range(k):
                accum += i
        return accum
    """
)

result = OpenAiChatPromptDriver(model="gpt-3.5-turbo-16k", temperature=0).run(stack)

print(result.value)
```

You can pass images to the Driver if the model supports it:

```
from griptape.artifacts import ListArtifact, TextArtifact
from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.loaders import ImageLoader

driver = OpenAiChatPromptDriver(model="gpt-4.1", max_tokens=256)

image_artifact = ImageLoader().load("./tests/resources/mountain.jpg")
text_artifact = TextArtifact("Describe the weather in the image")

driver.run(ListArtifact([text_artifact, image_artifact]))
```

11.2.2 Structured Output

Some LLMs provide functionality often referred to as “Structured Output”. This means instructing the LLM to output data in a particular format, usually JSON. This can be useful for forcing the LLM to output in a parsable format that can be used by downstream systems.

> **Warning**
>
> Each Driver may have a different default setting depending on the LLM provider's capabilities.

Prompt Task

The easiest way to get started with structured output is by using a *PromptTask* 's *default_run_output_schema_validation_subtasks(subtask_input)* parameter.

You can change *how* the output is structured by setting the Driver's validate*structured_output_strategy(*, value) to one of:

- `native` : The Driver will use the LLM's structured output functionality provided by the API.
- `tool` : The Driver will add a special tool, *StructuredOutputTool* , and will try to force the LLM to use the Tool.
- `rule` : The Driver will add a *JsonSchemaRule* to the Task's system prompt. This strategy does not guarantee that the LLM will output JSON and should only be used as a last resort.

You can specify `output_schema` using either `pydantic` or `schema` libraries, though `pydantic` is recommended.

```
from pydantic import BaseModel
from rich.pretty import pprint

from griptape.rules import Rule
from griptape.structures import Pipeline
from griptape.tasks import PromptTask

class Step(BaseModel):
    explanation: str
    output: str

class Output(BaseModel):
    steps: list[Step]
    final_answer: str

pipeline = Pipeline(
    tasks=[
        PromptTask(
            output_schema=Output,
            rules=[
                Rule("You are a helpful math tutor. Guide the user through the␣
↪solution step by step."),
            ],
        )
    ]
)

output = pipeline.run("How can I solve 8x + 7 = -23").output.value

pprint(output)  # OutputModel
```

```
import schema
from rich.pretty import pprint
```

(continues on next page)

(continued from previous page)

```
from griptape.rules import Rule
from griptape.structures import Pipeline
from griptape.tasks import PromptTask

pipeline = Pipeline(
    tasks=[
        PromptTask(
            output_schema=schema.Schema(
                {
                    "steps": [schema.Schema({"explanation": str, "output": str})],
                    "final_answer": str,
                }
            ),
            rules=[
                Rule("You are a helpful math tutor. Guide the user through the␣
↪solution step by step."),
            ],
        )
    ]
)

output = pipeline.run("How can I solve 8x + 7 = -23").output.value

pprint(output)  # dict
pprint(output["steps"])
```

11.2.3 Prompt Drivers

Gen AI Builder offers the following Prompt Drivers for interacting with LLMs.

OpenAI Chat

The *AzureOpenAiChatPromptDriver* connects to the OpenAI Chat API. This driver uses *OpenAi function calling* when using *Tools* .

```
import os

from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.structures import Agent

agent = Agent(
    prompt_driver=OpenAiChatPromptDriver(
        api_key=os.environ["OPENAI_API_KEY"],
        model="gpt-4.1",
        temperature=0.1,
        seed=42,
    ),
    input="You will be provided with a description of a mood, and your task is to␣
↪generate the CSS color code for a color that matches it. Description: {{ args[0] }}
↪",
)
```

(continues on next page)

(continued from previous page)

```
agent.run("Blue sky at dusk.")
```

`

OpenAi's *reasoning models* can also be used, and come with an additional parameter: *reasoning_effort* .

```
import os

from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.structures import Agent

agent = Agent(
    prompt_driver=OpenAiChatPromptDriver(
        api_key=os.environ["OPENAI_API_KEY"],
        model="o3-mini",
        reasoning_effort="medium",
    ),
)

agent.run("""Write a bash script that takes a matrix represented as a string with
format [1,2],[3,4],[5,6] and prints the transpose in the same format.""")
```

OpenAI Compatible

Many services such as LMStudio and OhMyGPT provide OpenAI-compatible APIs. You can use the *AzureOpenAiChatPromptDriver* to interact with these services. Simply set the `base_url` to the service's API endpoint and the `model` to the model name. If the service requires an API key, you can set it in the `api_key` field.

For example, here is how we can connect to LMStudio's API:

```
from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.rules import Rule
from griptape.structures import Agent

agent = Agent(
    prompt_driver=OpenAiChatPromptDriver(
        base_url="http://127.0.0.1:1234/v1", model="lmstudio-community/Meta-Llama-3-
↪8B-Instruct-GGUF", stream=True
    ),
    rules=[Rule(value="You are a helpful coding assistant.")],
)

agent.run("How do I init and update a git submodule?")
```

Or Groq's API:

```
import os

from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.structures import Agent

agent = Agent(
    prompt_driver=OpenAiChatPromptDriver(
```

(continues on next page)

(continued from previous page)

```
        api_key=os.environ["GROQ_API_KEY"],
        base_url="https://api.groq.com/openai/v1",
        model="llama-3.3-70b-versatile",
    ),
)

agent.run("Hello")
```

Tip

Make sure to include *v1* at the end of the *base_url* to match the OpenAI API endpoint.

Azure OpenAI Chat

The *AzureOpenAiChatPromptDriver* connects to Azure OpenAI *Chat Completion* APIs. This driver uses *Azure OpenAi function calling* when using *Tools* .

```
import os

from griptape.drivers.prompt.openai import AzureOpenAiChatPromptDriver
from griptape.rules import Rule
from griptape.structures import Agent

agent = Agent(
    prompt_driver=AzureOpenAiChatPromptDriver(
        api_key=os.environ["AZURE_OPENAI_API_KEY_1"],
        model="gpt-4.1",
        azure_endpoint=os.environ["AZURE_OPENAI_ENDPOINT_1"],
    ),
    rules=[
        Rule(
            value="You will be provided with text, and your task is to translate it␣
↪into emojis. "
            "Do not use any regular text. Do your best with emojis only."
        )
    ],
)

agent.run("Artificial intelligence is a technology with great promise.")
```

Gen AI Builder

The *GriptapeCloudPromptDriver* connects to the Gen AI Builder chat messages API.

```
import os

from griptape.drivers.prompt.griptape_cloud import GriptapeCloudPromptDriver
from griptape.rules import Rule
from griptape.structures import Agent

agent = Agent(
```

(continues on next page)

(continued from previous page)

```
    prompt_driver=GriptapeCloudPromptDriver(
        api_key=os.environ["GT_CLOUD_API_KEY"],
    ),
    rules=[
        Rule(
            "You will be provided with a product description and seed words, and your␣
↪task is to generate product names.",
        ),
    ],
)

agent.run("Product description: A home milkshake maker. Seed words: fast, healthy,␣
↪compact.")
```

Cohere

The *CoherePromptDriver* connects to the Cohere Chat API. This driver uses *Cohere tool use* when using *Tools* .

```
import os

from griptape.drivers.prompt.cohere import CoherePromptDriver
from griptape.structures import Agent

agent = Agent(
    prompt_driver=CoherePromptDriver(
        model="command-r",
        api_key=os.environ["COHERE_API_KEY"],
    )
)

agent.run(What is the sentiment of this review? Review: "I really enjoyed this movie!
↪")
```

Anthropic

The *AnthropicPromptDriver* connects to the Anthropic Messages API. This driver uses *Anthropic tool use* when using *Tools* .

```
import os

from griptape.drivers.prompt.anthropic import AnthropicPromptDriver
from griptape.structures import Agent

agent = Agent(
    prompt_driver=AnthropicPromptDriver(
        model="claude-3-opus-20240229",
        api_key=os.environ["ANTHROPIC_API_KEY"],
    )
)

agent.run("Where is the best place to see cherry blossums in Japan?")
```

Google

The *GooglePromptDriver* connects to the Google Generative AI <Optional title for the content. Only works with `retrieval_document task type.>` API. This driver uses Gemini function calling <Optional title for the content. Only works with `retrieval_document task type.>` when using *Tools* .

```
import os

from griptape.drivers.prompt.google import GooglePromptDriver
from griptape.structures import Agent

agent = Agent(
    prompt_driver=GooglePromptDriver(
        model="gemini-2.0-flash",
        api_key=os.environ["GOOGLE_API_KEY"],
    )
)

agent.run("Briefly explain how a computer works to a young child.")
```

Amazon Bedrock

The *AmazonBedrockPromptDriver* uses Amazon Bedrock 's *Converse API* . This driver uses *Bedrock tool use* when using *Tools* .

All models supported by the Converse API are available for use with this driver.

```
from griptape.drivers.prompt.amazon_bedrock import AmazonBedrockPromptDriver
from griptape.rules import Rule
from griptape.structures import Agent

agent = Agent(
    prompt_driver=AmazonBedrockPromptDriver(
        model="anthropic.claude-3-sonnet-20240229-v1:0",
    ),
    rules=[
        Rule(
            value="You are a customer service agent that is classifying emails by␣
↪type. I want you to give your answer and then explain it."
        )
    ],
)
agent.run(
    """How would you categorize this email?
    <email>
    Can I use my Mixmaster 4000 to mix paint, or is it only meant for mixing food?
    </email>

    Categories are:
    (A) Pre-sale question
    (B) Broken or defective item
    (C) Billing question
    (D) Other (please explain)"""
)
```

Ollama

The *OllamaPromptDriver* connects to the *Ollama Chat Completion API* . This driver uses *Ollama tool calling* when using *Tools* .

```
from griptape.drivers.prompt.ollama import OllamaPromptDriver
from griptape.structures import Agent
from griptape.tools import CalculatorTool

agent = Agent(
    prompt_driver=OllamaPromptDriver(
        model="llama3.1",
    ),
    tools=[CalculatorTool()],
)
agent.run("What is (192 + 12) ^ 4")
```

Hugging Face Hub

The *HuggingFaceHubPromptDriver* connects to the Hugging Face Hub API .

Warning

Not all models featured on the Hugging Face Hub are supported by this driver. Models that are not supported by *Hugging Face serverless inference* will not work with this driver. Due to the limitations of Hugging Face serverless inference, only models that are than 10GB are supported.

```
import os

from griptape.drivers.prompt.huggingface_hub import HuggingFaceHubPromptDriver
from griptape.rules import Rule, Ruleset
from griptape.structures import Agent

agent = Agent(
    prompt_driver=HuggingFaceHubPromptDriver(
        model="HuggingFaceH4/zephyr-7b-beta",
        api_token=os.environ["HUGGINGFACE_HUB_ACCESS_TOKEN"],
    ),
    rulesets=[
        Ruleset(
            name="Girafatron",
            rules=[
                Rule(
                    value="You are Girafatron, a giraffe-obsessed robot. You are␣
↪talking to a human. "
                    "Girafatron is obsessed with giraffes, the most glorious animal␣
↪on the face of this Earth. "
                    "Giraftron believes all other animals are irrelevant when␣
↪compared to the glorious majesty of the giraffe."
                )
            ],
        )
```

(continues on next page)

(continued from previous page)

```
    ],
)

agent.run("Hello Girafatron, what is your favorite animal?")
```

Text Generation Interface

The *HuggingFaceHubPromptDriver* also supports Text Generation Interface for running models locally. To use Text Generation Interface, just set `model` to a TGI endpoint.

```
import os

from griptape.drivers.prompt.huggingface_hub import HuggingFaceHubPromptDriver
from griptape.structures import Agent

agent = Agent(
    prompt_driver=HuggingFaceHubPromptDriver(
        model="http://127.0.0.1:8080",
        api_token=os.environ["HUGGINGFACE_HUB_ACCESS_TOKEN"],
    ),
)

agent.run("Write the code for a snake game.")
```

Hugging Face Pipeline

The *HuggingFacePipelinePromptDriver* uses *Hugging Face Pipelines* for inference locally.

> **Warning**
>
> Running a model locally can be a computationally expensive process.

```
from griptape.drivers.prompt.huggingface_pipeline import␣
↪HuggingFacePipelinePromptDriver
from griptape.rules import Rule, Ruleset
from griptape.structures import Agent

agent = Agent(
    prompt_driver=HuggingFacePipelinePromptDriver(
        model="TinyLlama/TinyLlama-1.1B-Chat-v1.0",
    ),
    rulesets=[
        Ruleset(
            name="Pirate",
            rules=[Rule(value="You are a pirate chatbot who always responds in pirate␣
↪speak!")],
        )
    ],
)

agent.run("How many helicopters can a human eat in one sitting?")
```

Amazon SageMaker Jumpstart

The *AmazonSageMakerJumpstartPromptDriver* uses Amazon SageMaker Jumpstart for inference on AWS.

Amazon Sagemaker Jumpstart provides a wide range of models with varying capabilities. This Driver has been primarily *chat-optimized* models that have a *Huggingface Chat Template* available. If your model does not fit this use-case, we suggest sub-classing *AmazonSageMakerJumpstartPromptDriver* and overriding the `_to_model_input` and `_to_model_params` methods.

```
import os

from griptape.drivers.prompt.amazon_sagemaker_jumpstart import␣
↪AmazonSageMakerJumpstartPromptDriver
from griptape.structures import Agent

agent = Agent(
    prompt_driver=AmazonSageMakerJumpstartPromptDriver(
        endpoint=os.environ["SAGEMAKER_LLAMA_3_INSTRUCT_ENDPOINT_NAME"],
        model="meta-llama/Meta-Llama-3-8B-Instruct",
    )
)

agent.run("What is a good lasagna recipe?")
```

Grok

The *GrokPromptDriver* uses *Grok's chat completion* endpoint.

```
import os

from griptape.drivers.prompt.grok import GrokPromptDriver
from griptape.rules import Rule
from griptape.structures import Agent

agent = Agent(
    prompt_driver=GrokPromptDriver(model="grok-2-latest", api_key=os.environ["GROK_
↪API_KEY"]),
    rules=[Rule("You are Grok, a chatbot inspired by the Hitchhikers Guide to the␣
↪Galaxy.")],
)

agent.run("What is the meaning of life, the universe, and everything?")
```

Perplexity

The *PerplexityPromptDriver* uses *Perplexity Sonar's chat completion* endpoint.

While you can use this Driver directly, we recommend using it through its accompanying *Web Search Drivers* .

```
import os

from griptape.drivers.prompt.perplexity import PerplexityPromptDriver
from griptape.rules import Rule
from griptape.structures import Agent
from griptape.tasks import PromptTask
```

(continues on next page)

(continued from previous page)

```
agent = Agent(
    tasks=[
        PromptTask(
            prompt_driver=PerplexityPromptDriver(model="sonar-pro", api_key=os.
↪environ["PERPLEXITY_API_KEY"]),
            rules=[
                Rule("Be precise and concise"),
            ],
        )
    ],
)

agent.run("How many stars are there in our galaxy?")
```

Assistant

Overview

Assistant Drivers can be used to interact with various "assistant" services.

Assistant Drivers

Gen AI Builder

The *GriptapeCloudAssistantDriver* is used to run Gen AI Builder Assistants.

```
import os

from griptape.drivers.assistant.griptape_cloud import GriptapeCloudAssistantDriver
from griptape.structures import Pipeline
from griptape.tasks import AssistantTask
from griptape.utils.stream import Stream

pipeline = Pipeline(
    tasks=[
        AssistantTask(
            assistant_driver=GriptapeCloudAssistantDriver(
                assistant_id=os.environ["GT_CLOUD_ASSISTANT_ID"],
                stream=True,
            ),
        ),
    ]
)

for chunk in Stream(pipeline).run("Write me long poem"):
    print(chunk, end="", flush=True)
```

OpenAI

The *OpenAiAssistantDriver* is used to run *OpenAI Assistants* .

```
import os

from griptape.drivers.assistant.openai import OpenAiAssistantDriver
from griptape.structures import Pipeline
from griptape.tasks import AssistantTask
from griptape.utils.stream import Stream

pipeline = Pipeline(
    tasks=[
        AssistantTask(
            assistant_driver=OpenAiAssistantDriver(
                assistant_id=os.environ["OPENAI_ASSISTANT_ID"], thread_id=os.environ[
↪"OPENAI_THREAD_ID"]
            ),
        ),
    ]
)

for chunk in Stream(pipeline).run("I need to solve the equation `3x + 11 = 14`. Can␣
↪you help me?"):
    print(chunk, end="", flush=True)
```

Audio Transcription

Overview

Audio Transcription Drivers extract text from spoken audio.

Audio Transcription Drivers

OpenAI

The *OpenAI Audio Transcription Driver* utilizes OpenAI's sophisticated `whisper` model to accurately transcribe spoken audio into text. This model supports multiple languages, ensuring precise transcription across a wide range of dialects.

```
from griptape.drivers.audio_transcription.openai import OpenAiAudioTranscriptionDriver
from griptape.structures import Agent
from griptape.tools.audio_transcription.tool import AudioTranscriptionTool

driver = OpenAiAudioTranscriptionDriver(model="whisper-1")

tool = AudioTranscriptionTool(
    off_prompt=False,
    audio_transcription_driver=driver,
)

Agent(tools=[tool]).run("Transcribe the following audio file: tests/resources/
↪sentences.wav")
```

Conversation Memory

Overview

You can persist and load memory by using Conversation Memory Drivers. You can build drivers for your own data stores by extending *BaseConversationMemoryDriver* .

Conversation Memory Drivers

Gen AI Builder

The *GriptapeCloudConversationMemoryDriver* allows you to persist Conversation Memory in Gen AI Builder. It provides seamless integration with Gen AI Builder's cloud-based `Threads` and `Messages` resources.

```
import os

from griptape.drivers.memory.conversation.griptape_cloud import␣
↪GriptapeCloudConversationMemoryDriver
from griptape.memory.structure import ConversationMemory
from griptape.structures import Agent

cloud_conversation_driver = GriptapeCloudConversationMemoryDriver(
    api_key=os.environ["GT_CLOUD_API_KEY"],
    alias="my_thread_alias",
)
agent = Agent(conversation_memory=ConversationMemory(conversation_memory_driver=cloud_
↪conversation_driver))

agent.run("My name is Jeff.")
agent.run("What is my name?")
```

Local

The *LocalConversationMemoryDriver* allows you to persist Conversation Memory in a local JSON file.

```
from griptape.drivers.memory.conversation.local import LocalConversationMemoryDriver
from griptape.memory.structure import ConversationMemory
from griptape.structures import Agent

local_driver = LocalConversationMemoryDriver(persist_file="memory.json")
agent = Agent(conversation_memory=ConversationMemory(conversation_memory_driver=local_
↪driver))

agent.run("Surfing is my favorite sport.")
agent.run("What is my favorite sport?")
```

Amazon DynamoDb

The *AmazonDynamoDbConversationMemoryDriver* allows you to persist Conversation Memory in Amazon DynamoDb .

```
import os
import uuid
```

(continues on next page)

(continued from previous page)

```
from griptape.drivers.memory.conversation.amazon_dynamodb import␣
↪AmazonDynamoDbConversationMemoryDriver
from griptape.memory.structure import ConversationMemory
from griptape.structures import Agent

conversation_id = uuid.uuid4().hex
dynamodb_driver = AmazonDynamoDbConversationMemoryDriver(
    table_name=os.environ["DYNAMODB_TABLE_NAME"],
    partition_key="id",
    value_attribute_key="memory",
    partition_key_value=conversation_id,
)

agent = Agent(conversation_memory=ConversationMemory(conversation_memory_
↪driver=dynamodb_driver))

agent.run("My name is Jeff.")
agent.run("What is my name?")
```

Optional parameters `sort_key` and `sort_key_value` can be supplied for tables with a composite primary key.

Redis

The *RedisConversationMemoryDriver* allows you to persist Conversation Memory in Redis .

```
import os
import uuid

from griptape.drivers.memory.conversation.redis import RedisConversationMemoryDriver
from griptape.memory.structure import ConversationMemory
from griptape.structures import Agent

conversation_id = uuid.uuid4().hex
redis_conversation_driver = RedisConversationMemoryDriver(
    host=os.environ["REDIS_HOST"],
    port=int(os.environ["REDIS_PORT"]),
    username=os.environ["REDIS_USERNAME"],
    password=os.environ["REDIS_PASSWORD"],
    index=os.environ["REDIS_INDEX"],
    conversation_id=conversation_id,
)

agent = Agent(conversation_memory=ConversationMemory(conversation_memory_driver=redis_
↪conversation_driver))

agent.run("My name is Jeff.")
agent.run("What is my name?")
```

Embedding

Overview

Embeddings in Gen AI Builder are multidimensional representations of text or image data. Embeddings carry semantic information, making them powerful for use-cases like text or image similarity search in a *RAG Engine* .

Embedding Drivers

OpenAI

The *OpenAiEmbeddingDriver* uses the *OpenAI Embeddings API* .

```
from griptape.drivers.embedding.openai import OpenAiEmbeddingDriver

embeddings = OpenAiEmbeddingDriver().embed("Hello Griptape!")

#  display the first 3 embeddings

print(embeddings[:3])
```

```
[0.0017853748286142945, 0.006118456833064556, -0.005811543669551611]
```

OpenAI Compatible

Many services such as LMStudio and OhMyGPT provide OpenAI-compatible APIs. You can use the *OpenAiEmbeddingDriver* to interact with these services. Simply set the `base_url` to the service's API endpoint and the `model` to the model name. If the service requires an API key, you can set it in the `api_key` field.

```
from griptape.drivers.embedding.openai import OpenAiEmbeddingDriver

embedding_driver = OpenAiEmbeddingDriver(
    base_url="http://127.0.0.1:1234/v1",
    model="nomic-ai/nomic-embed-text-v1.5-GGUF/nomic-embed-text-v1.5.Q2_K",
)

embeddings = embedding_driver.embed("Hello world!")

#  display the first 3 embeddings

print(embeddings[:3])
```

> **Tip**
>
> Make sure to include *v1* at the end of the *base_url* to match the OpenAI API endpoint.

Azure OpenAI

The *AzureOpenAiEmbeddingDriver* uses the same parameters as *OpenAiEmbeddingDriver*

with updated defaults.

Bedrock Titan

The *AmazonBedrockTitanEmbeddingDriver* uses the Amazon Bedrock Embeddings API .

```
from griptape.drivers.embedding.amazon_bedrock import␣
↪AmazonBedrockTitanEmbeddingDriver
from griptape.loaders import ImageLoader

embedding_driver = AmazonBedrockTitanEmbeddingDriver()
embeddings = embedding_driver.embed("Hello world!")

print(embeddings[:3])

#  Some models support images!

multi_modal_embedding_driver = AmazonBedrockTitanEmbeddingDriver(model="amazon.titan-
↪embed-image-v1")
image = ImageLoader().load("tests/resources/cow.png")
image_embeddings = multi_modal_embedding_driver.embed(image)

print(image_embeddings[:3])
```

```
[-0.234375, -0.024902344, -0.14941406]
```

Google

The *GoogleEmbeddingDriver* uses the Google Embeddings API <Optional title for the content. Only works with `retrieval_document task type.>` .

```
from griptape.drivers.embedding.google import GoogleEmbeddingDriver

embeddings = GoogleEmbeddingDriver().embed("Hello world!")

#  display the first 3 embeddings

print(embeddings[:3])
```

```
[0.0588633, 0.0033929371, -0.072810836]
```

Hugging Face Hub

The *HuggingFaceHubEmbeddingDriver* connects to the Hugging Face Hub API . It supports models with the following tasks:

- feature-extraction

```
import os

from griptape.drivers.embedding.huggingface_hub import HuggingFaceHubEmbeddingDriver
from griptape.tokenizers import HuggingFaceTokenizer

driver = HuggingFaceHubEmbeddingDriver(
    api_token=os.environ["HUGGINGFACE_HUB_ACCESS_TOKEN"],
```

(continues on next page)

(continued from previous page)

```
    model="sentence-transformers/all-MiniLM-L6-v2",
    tokenizer=HuggingFaceTokenizer(
        model="sentence-transformers/all-MiniLM-L6-v2",
        max_output_tokens=512,
    ),
)

embeddings = driver.embed("Hello world!")

#  display the first 3 embeddings

print(embeddings[:3])
```

Ollama

The *OllamaEmbeddingDriver* uses the *Ollama Embeddings API* .

```
from griptape.drivers.embedding.ollama import OllamaEmbeddingDriver

driver = OllamaEmbeddingDriver(
    model="all-minilm",
)

results = driver.embed("Hello world!")

#  display the first 3 embeddings

print(results[:3])
```

Amazon SageMaker Jumpstart

The *AmazonSageMakerJumpstartEmbeddingDriver* uses the Amazon SageMaker Endpoints to generate embeddings on AWS.

```
import os

from griptape.drivers.embedding.amazon_sagemaker_jumpstart import␣
↪AmazonSageMakerJumpstartEmbeddingDriver

driver = AmazonSageMakerJumpstartEmbeddingDriver(
    endpoint=os.environ["SAGEMAKER_ENDPOINT"],
    model=os.environ["SAGEMAKER_TENSORFLOW_HUB_MODEL"],
)

embeddings = driver.embed("Hello world!")

#  display the first 3 embeddings

print(embeddings[:3])
```

VoyageAI

The *VoyageAiEmbeddingDriver* uses the *VoyageAI Embeddings API* .

```
import os

from griptape.drivers.embedding.voyageai import VoyageAiEmbeddingDriver
from griptape.loaders import ImageLoader

embedding_driver = VoyageAiEmbeddingDriver(api_key=os.environ["VOYAGE_API_KEY"])
embeddings = embedding_driver.embed("Hello world!")

print(embeddings[:3])

#  Some models support images!

multi_modal_embedding_driver = VoyageAiEmbeddingDriver(
    api_key=os.environ["VOYAGE_API_KEY"], model="voyage-multimodal-3"
)
image = ImageLoader().load("tests/resources/cow.png")
image_embeddings = multi_modal_embedding_driver.embed(image)

print(image_embeddings[:3])
```

Cohere

The *CohereEmbeddingDriver* uses the *Cohere Embeddings API* .

```
import os

from griptape.drivers.embedding.cohere import CohereEmbeddingDriver

embedding_driver = CohereEmbeddingDriver(
    model="embed-english-v3.0",
    api_key=os.environ["COHERE_API_KEY"],
    input_type="search_document",
)

embeddings = embedding_driver.embed("Hello world!")

#  display the first 3 embeddings

print(embeddings[:3])
```

Nvidia NIM

The *NvidiaNimEmbeddingDriver* uses the Nvidia NIM API .

Override Default Structure Embedding Driver

Here is how you can override the Embedding Driver that is used by default in Structures.

```
from griptape.configs import Defaults
from griptape.configs.drivers import DriversConfig
from griptape.drivers.embedding.voyageai import VoyageAiEmbeddingDriver
from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.structures import Agent
from griptape.tools import PromptSummaryTool, WebScraperTool

Defaults.drivers_config = DriversConfig(
    prompt_driver=OpenAiChatPromptDriver(model="gpt-4.1"),
    embedding_driver=VoyageAiEmbeddingDriver(),
)

Defaults.drivers_config = DriversConfig(
    prompt_driver=OpenAiChatPromptDriver(model="gpt-4.1"),
    embedding_driver=VoyageAiEmbeddingDriver(),
)

agent = Agent(
    tools=[WebScraperTool(off_prompt=True), PromptSummaryTool(off_prompt=False)],
)

agent.run("based on https://www.griptape.ai/, tell me what Griptape is")
```

Event Listener

Overview

Event Listener Drivers are used to send Gen AI Builder *Events* to external services.

You can instantiate Drivers and pass them to Event Listeners in your Structure:

```
import os

from griptape.drivers.event_listener.amazon_sqs import AmazonSqsEventListenerDriver
from griptape.events import EventBus, EventListener
from griptape.rules import Rule
from griptape.structures import Agent

EventBus.add_event_listeners(
    [
        EventListener(
            event_listener_driver=AmazonSqsEventListenerDriver(
                queue_url=os.environ["AMAZON_SQS_QUEUE_URL"],
            ),
        ),
    ]
)

agent = Agent(
    rules=[
        Rule(value="You will be provided with a block of text, and your task is to␣
↪extract a list of keywords from it.")
    ],
```

(continues on next page)

(continued from previous page)

```
)

agent.run(
    """Black-on-black ware is a 20th- and 21st-century pottery tradition developed by␣
↪the Puebloan Native American ceramic artists in Northern New Mexico.
    Traditional reduction-fired blackware has been made for centuries by pueblo␣
↪artists.
    Black-on-black ware of the past century is produced with a smooth surface, with␣
↪the designs applied through selective burnishing or the application of refractory␣
↪slip.
    Another style involves carving or incising designs and selectively polishing the␣
↪raised areas.
    For generations several families from Khapo Owingeh and Pohwhóge Owingeh pueblos␣
↪have been making black-on-black ware with the techniques passed down from matriarch␣
↪potters. Artists from other pueblos have also produced black-on-black ware.
    Several contemporary artists have created works honoring the pottery of their␣
↪ancestors."""
)
```

Or use them independently:

```
from griptape.artifacts import TextArtifact
from griptape.drivers.event_listener.griptape_cloud import␣
↪GriptapeCloudEventListenerDriver
from griptape.events import FinishStructureRunEvent

#  By default, GriptapeCloudEventListenerDriver uses the api key provided

#  in the GT_CLOUD_API_KEY environment variable.

event_driver = GriptapeCloudEventListenerDriver()

done_event = FinishStructureRunEvent(
    output_task_input=TextArtifact("Just started!"),
    output_task_output=TextArtifact("All done!"),
)

event_driver.publish_event(done_event)
```

Event Listener Drivers

Gen AI Builder offers the following Event Listener Drivers for forwarding Gen AI Builder Events.

Amazon SQS

The *AmazonSqsEventListenerDriver* sends Events to an Amazon SQS queue.

```
import os

from griptape.drivers.event_listener.amazon_sqs import AmazonSqsEventListenerDriver
from griptape.events import EventBus, EventListener
from griptape.rules import Rule
```

(continues on next page)

(continued from previous page)

```
from griptape.structures import Agent

EventBus.add_event_listeners(
    [
        EventListener(
            event_listener_driver=AmazonSqsEventListenerDriver(
                queue_url=os.environ["AMAZON_SQS_QUEUE_URL"],
            ),
        ),
    ]
)

agent = Agent(
    rules=[
        Rule(value="You will be provided with a block of text, and your task is to␣
↪extract a list of keywords from it.")
    ],
)

agent.run(
    """Black-on-black ware is a 20th- and 21st-century pottery tradition developed by␣
↪the Puebloan Native American ceramic artists in Northern New Mexico.
    Traditional reduction-fired blackware has been made for centuries by pueblo␣
↪artists.
    Black-on-black ware of the past century is produced with a smooth surface, with␣
↪the designs applied through selective burnishing or the application of refractory␣
↪slip.
    Another style involves carving or incising designs and selectively polishing the␣
↪raised areas.
    For generations several families from Khapo Owingeh and Pohwhóge Owingeh pueblos␣
↪have been making black-on-black ware with the techniques passed down from matriarch␣
↪potters. Artists from other pueblos have also produced black-on-black ware.
    Several contemporary artists have created works honoring the pottery of their␣
↪ancestors."""
)
```

AWS IoT

The *AwsIotCoreEventListenerDriver* sends Events to the AWS IoT Message Broker .

```
import os

from griptape.configs import Defaults
from griptape.configs.drivers import DriversConfig
from griptape.drivers.event_listener.aws_iot_core import AwsIotCoreEventListenerDriver
from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.events import EventBus, EventListener, FinishStructureRunEvent
from griptape.rules import Rule
from griptape.structures import Agent

Defaults.drivers_config = DriversConfig(prompt_driver=OpenAiChatPromptDriver(model=
↪"gpt-3.5-turbo", temperature=0.7))
```

(continues on next page)

(continued from previous page)

```
EventBus.add_event_listeners(
    [
        EventListener(
            event_types=[FinishStructureRunEvent],
            event_listener_driver=AwsIotCoreEventListenerDriver(
                topic=os.environ["AWS_IOT_CORE_TOPIC"],
                iot_endpoint=os.environ["AWS_IOT_CORE_ENDPOINT"],
            ),
        ),
    ]
)

agent = Agent(
    rules=[Rule(value="You will be provided with a text, and your task is to extract␣
↪the airport codes from it.")],
)

agent.run("I want to fly from Orlando to Boston")
```

Gen AI Builder

The *GriptapeCloudEventListenerDriver* sends Events to Gen AI Builder .

Note

This Driver is required when using the Gen AI Builder Managed Structures feature. For local development, you can use the Skatepark Emulator .

```
from griptape.drivers.event_listener.griptape_cloud import␣
↪GriptapeCloudEventListenerDriver
from griptape.events import EventBus, EventListener, FinishStructureRunEvent
from griptape.structures import Agent

EventBus.add_event_listeners(
    [
        EventListener(
            event_types=[FinishStructureRunEvent],
            # By default, GriptapeCloudEventListenerDriver uses the api key provided
            # in the GT_CLOUD_API_KEY environment variable.
            event_listener_driver=GriptapeCloudEventListenerDriver(),
        ),
    ]
)

agent = Agent()
agent.run("Create a list of 8 questions for an interview with a science fiction␣
↪author.")
```

Webhook Event Listener Driver

The *WebhookEventListenerDriver* sends Events to any Webhook URL.

```
import os

from griptape.drivers.event_listener.webhook import WebhookEventListenerDriver
from griptape.events import EventBus, EventListener, FinishStructureRunEvent
from griptape.structures import Agent

EventBus.add_event_listeners(
    [
        EventListener(
            event_types=[FinishStructureRunEvent],
            event_listener_driver=WebhookEventListenerDriver(
                webhook_url=os.environ["WEBHOOK_URL"],
            ),
        ),
    ]
)

agent = Agent()

agent.run("Analyze the pros and cons of remote work vs. office work")
```

Pusher

The *PusherEventListenerDriver* sends Events to Pusher .

```
import os

from griptape.drivers.event_listener.pusher import PusherEventListenerDriver
from griptape.events import EventBus, EventListener, FinishStructureRunEvent
from griptape.structures import Agent

EventBus.add_event_listeners(
    [
        EventListener(
            event_types=[FinishStructureRunEvent],
            event_listener_driver=PusherEventListenerDriver(
                batched=False,
                app_id=os.environ["PUSHER_APP_ID"],
                key=os.environ["PUSHER_KEY"],
                secret=os.environ["PUSHER_SECRET"],
                cluster=os.environ["PUSHER_CLUSTER"],
                channel="my-channel",
                event_name="my-event",
            ),
        ),
    ],
)

agent = Agent()
```

(continues on next page)

(continued from previous page)

```
agent.run("Analyze the pros and cons of remote work vs. office work")
```

File Manager

Overview

File Manager Drivers can be used to load and save files with local or external file systems.

You can use File Manager Drivers with Loaders:

```
from griptape.drivers.file_manager.local import LocalFileManagerDriver
from griptape.loaders import TextLoader

local_file_manager_driver = LocalFileManagerDriver()

loader = TextLoader(file_manager_driver=local_file_manager_driver)
text_artifact = loader.load("tests/resources/test.txt")

print(text_artifact.value)
```

Or use them independently as shown below for each driver:

File Manager Drivers

Gen AI Builder

The *GriptapeCloudFileManagerDriver* allows you to load and save files sourced from Gen AI Builder Asset and Bucket resources.

```
import os

from griptape.drivers.file_manager.griptape_cloud import␣
↪GriptapeCloudFileManagerDriver

gtc_file_manager_driver = GriptapeCloudFileManagerDriver(
    api_key=os.environ["GT_CLOUD_API_KEY"],
    bucket_id=os.environ["GT_CLOUD_BUCKET_ID"],
)

#  Download File

file_contents = gtc_file_manager_driver.load_file(os.environ["GT_CLOUD_ASSET_NAME"])

print(file_contents)

#  Upload File

response = gtc_file_manager_driver.save_file(os.environ["GT_CLOUD_ASSET_NAME"], file_
↪contents.value)

print(response)
```

Local

The *LocalFileManagerDriver* allows you to load and save files sourced from a local directory.

```
from griptape.drivers.file_manager.local import LocalFileManagerDriver

local_file_manager_driver = LocalFileManagerDriver()

#  Download File

file_contents = local_file_manager_driver.load_file("tests/resources/test.txt")

print(file_contents)

#  Upload File

response = local_file_manager_driver.save_file("tests/resources/test.txt", file_
↪contents.value)

print(response)
```

Amazon S3

The *LocalFile ManagerDriver* allows you to load and save files sourced from an Amazon S3 bucket.

```
import os

import boto3

from griptape.drivers.file_manager.amazon_s3 import AmazonS3FileManagerDriver

amazon_s3_file_manager_driver = AmazonS3FileManagerDriver(
    bucket=os.environ["AMAZON_S3_BUCKET"],
    session=boto3.Session(
        region_name=os.environ["AWS_DEFAULT_REGION"],
        aws_access_key_id=os.environ["AWS_ACCESS_KEY_ID"],
        aws_secret_access_key=os.environ["AWS_SECRET_ACCESS_KEY"],
    ),
)

#  Download File

file_contents = amazon_s3_file_manager_driver.load_file(os.environ["AMAZON_S3_KEY"])

print(file_contents)

#  Upload File

response = amazon_s3_file_manager_driver.save_file(os.environ["AMAZON_S3_KEY"], file_
↪contents.value)

print(response)
```

Image Generation

Overview

Image Generation Drivers build and execute API calls to image generation models.

Provide a Driver to a *RAG Tool* for use by an *agent* :

```
from griptape.drivers.image_generation.openai import OpenAiImageGenerationDriver
from griptape.structures import Agent
from griptape.tools import PromptImageGenerationTool

driver = OpenAiImageGenerationDriver(
    model="gpt-image-1",
)

agent = Agent(
    tools=[
        PromptImageGenerationTool(image_generation_driver=driver),
    ]
)

agent.run("Generate a watercolor painting of a dog riding a skateboard")
```

Image Generation Drivers

Gen AI Builder

The *Gen AI Builder Image Generation Driver* provides access to image generation models hosted by Gen AI Builder.

Today, the only accessible model is `dall-e-3` .

```
import os
from io import BytesIO

from PIL import Image

from griptape.drivers.image_generation.griptape_cloud import␣
↪GriptapeCloudImageGenerationDriver

driver = GriptapeCloudImageGenerationDriver(
    api_key=os.environ["GT_CLOUD_API_KEY"],
    model="dall-e-3",
)

image = driver.run_text_to_image(["A capybara sitting on a rock in the sun."])

Image.open(BytesIO(image.value)).show()
```

Amazon Bedrock

The *Amazon Bedrock Image Generation Driver* provides multi-model access to image generation models hosted by Amazon Bedrock. This Driver manages API calls to the Bedrock API, while the specific Model Drivers below format the API requests and parse the responses.

Stable Diffusion

The *Bedrock Stable Diffusion Model Driver* provides support for Stable Diffusion models hosted by Amazon Bedrock. This Model Driver supports configurations specific to Stable Diffusion, like style presets, clip guidance presets, and sampler.

This Model Driver supports negative prompts. When provided, the image generation request will include negatively-weighted prompts describing features or characteristics to avoid in the resulting generation.

```
from griptape.drivers.image_generation.amazon_bedrock import␣
↪AmazonBedrockImageGenerationDriver
from griptape.drivers.image_generation_model.bedrock_stable_diffusion import (
    BedrockStableDiffusionImageGenerationModelDriver,
)
from griptape.structures import Agent
from griptape.tools import PromptImageGenerationTool

model_driver = BedrockStableDiffusionImageGenerationModelDriver(
    style_preset="pixel-art",
)

driver = AmazonBedrockImageGenerationDriver(
    image_generation_model_driver=model_driver,
    model="stability.stable-diffusion-xl-v1",
)

agent = Agent(
    tools=[
        PromptImageGenerationTool(image_generation_driver=driver),
    ]
)

agent.run("Generate an image of a dog riding a skateboard")
```

Titan

The *Bedrock Titan Image Generator Model Driver* provides support for Titan Image Generator models hosted by Amazon Bedrock. This Model Driver supports configurations specific to Titan Image Generator, like quality, seed, and cfg_scale.

This Model Driver supports negative prompts. When provided, the image generation request will include negatively-weighted prompts describing features or characteristics to avoid in the resulting generation.

```
from griptape.drivers.image_generation.amazon_bedrock import␣
↪AmazonBedrockImageGenerationDriver
from griptape.drivers.image_generation_model.bedrock_titan import␣
↪BedrockTitanImageGenerationModelDriver
from griptape.structures import Agent
from griptape.tools import PromptImageGenerationTool

model_driver = BedrockTitanImageGenerationModelDriver()

driver = AmazonBedrockImageGenerationDriver(
    image_generation_model_driver=model_driver,
```

(continues on next page)

(continued from previous page)

```
    model="amazon.titan-image-generator-v2:0",
)

agent = Agent(
    tools=[
        PromptImageGenerationTool(image_generation_driver=driver),
    ]
)

agent.run("Generate a watercolor painting of a dog riding a skateboard")
```

Azure OpenAI

The *Azure OpenAI Image Generation Driver* provides access to OpenAI models hosted by Azure. In addition to the configurations provided by the underlying OpenAI Driver, the Azure OpenAI Driver allows configuration of Azure-specific deployment values.

```
import os

from griptape.drivers.image_generation.openai import AzureOpenAiImageGenerationDriver
from griptape.structures import Agent
from griptape.tools import PromptImageGenerationTool

driver = AzureOpenAiImageGenerationDriver(
    model="dall-e-3",
    azure_endpoint=os.environ["AZURE_OPENAI_ENDPOINT_2"],
    api_key=os.environ["AZURE_OPENAI_API_KEY_2"],
)

agent = Agent(
    tools=[
        PromptImageGenerationTool(image_generation_driver=driver),
    ]
)

agent.run("Generate a watercolor painting of a dog riding a skateboard")
```

Leonardo.Ai

The *Leonardo Image Generation Driver* enables image generation using models hosted by Leonardo.ai .

This Driver supports configurations like model selection, image size, specifying a generation seed, and generation steps. For details on supported configuration parameters, see *Leonardo.Ai's image generation documentation* .

This Driver supports negative prompts. When provided, the image generation request will include negatively-weighted prompts describing features or characteristics to avoid in the resulting generation.

```
import os

from griptape.drivers.image_generation.leonardo import LeonardoImageGenerationDriver
from griptape.structures import Agent
from griptape.tools import PromptImageGenerationTool
```

(continues on next page)

(continued from previous page)

```
driver = LeonardoImageGenerationDriver(
    model=os.environ["LEONARDO_MODEL_ID"],
    api_key=os.environ["LEONARDO_API_KEY"],
    image_width=512,
    image_height=1024,
)

agent = Agent(
    tools=[
        PromptImageGenerationTool(image_generation_driver=driver),
    ]
)

agent.run("Generate a watercolor painting of a dog riding a skateboard")
```

OpenAI

The *OpenAI Image Generation Driver* provides access to OpenAI image generation models. Like other OpenAI Drivers, the image generation Driver will implicitly load an API key in the `OPENAI_API_KEY` environment variable if one is not explicitly provided.

This Driver supports image generation configurations like style presets, image quality preference, and image size. For details on supported configuration values, see the *OpenAI documentation* .

```
from griptape.drivers.image_generation.openai import OpenAiImageGenerationDriver
from griptape.structures import Agent
from griptape.tools import FileManagerTool, PromptImageGenerationTool

driver = OpenAiImageGenerationDriver(model="gpt-image-1")

agent = Agent(tools=[PromptImageGenerationTool(image_generation_driver=driver, off_
↪prompt=True), FileManagerTool()])

agent.run("Generate a watercolor painting of a dog riding a skateboard and save it to␣
↪dog.png")
```

HuggingFace Pipelines

The *HuggingFace Pipelines Image Generation Driver* enables image generation through locally-hosted models using the HuggingFace *Diffusers* library. This Driver requires a *Stable Diffusion 3 Image Generation Pipeline Driver* to prepare the appropriate Pipeline.

This Driver requires a `model` configuration, specifying the model to use for image generation. The value of the `model` configuration must be one of the following:

- A model name from the HuggingFace Model Hub, like `stabilityai/stable-diffusion-3-medium-diffusers`
- A path to the directory containing a model on the filesystem, like `./models/stable-diffusion-3/`
- A path to a file containing a model on the filesystem, like `./models/sd3_medium_incl_clips.safetensors`

The `device` configuration specifies the hardware device used to run inference. Common values include `cuda` (supporting CUDA-enabled GPUs), `cpu` (supported by a device's CPU), and `mps` (supported by Apple silicon GPUs). For more information, see HuggingFace's documentation on GPU inference.

Stable Diffusion 3 Image Generation Pipeline Driver

The *Stable Diffusion 3 Image Generation Pipeline Driver* provides a Stable `Diffusion3DiffusionPipeline` for text-to-image generations via the *HuggingFace Pipelines Image Generation Driver's* `.try_text_to_image()` method. This Driver accepts a text prompt and configurations including Stable Diffusion 3 model, output image size, generation seed, and inference steps.

Image generation consumes substantial memory. On devices with limited VRAM, it may be necessary to enable the `enable_model_cpu_offload` or `drop_t5_encoder` configurations. For more information, see HuggingFace's documentation on reduced memory usage.

```
from griptape.artifacts import TextArtifact
from griptape.drivers.image_generation.huggingface_pipeline import␣
↪HuggingFacePipelineImageGenerationDriver
from griptape.drivers.image_generation_pipeline.stable_diffusion_3 import␣
↪StableDiffusion3ImageGenerationPipelineDriver
from griptape.structures import Pipeline
from griptape.tasks import PromptImageGenerationTask

image_generation_task = PromptImageGenerationTask(
    input=TextArtifact("landscape photograph, verdant, countryside, 8k"),
    image_generation_driver=HuggingFacePipelineImageGenerationDriver(
        model="stabilityai/stable-diffusion-3-medium-diffusers",
        device="cuda",
        pipeline_driver=StableDiffusion3ImageGenerationPipelineDriver(
            height=512,
            width=512,
        ),
    ),
)

output_artifact = Pipeline(tasks=[image_generation_task]).run().output
```

Stable Diffusion 3 Img2Img Image Generation Pipeline Driver

The *Stable Diffusion 3 Img2Img Image Generation Pipeline Driver* provides a `StableDiffusion3Img2Img-Pipeline` for image-to-image generations, accepting a text prompt and input image. This Driver accepts a text prompt, an input image, and configurations including Stable Diffusion 3 model, output image size, inference steps, generation seed, and strength of generation over the input image.

```
from griptape.artifacts import TextArtifact
from griptape.drivers.image_generation.huggingface_pipeline import␣
↪HuggingFacePipelineImageGenerationDriver
from griptape.drivers.image_generation_pipeline.stable_diffusion_3_img_2_img import (
    StableDiffusion3Img2ImgImageGenerationPipelineDriver,
)
from griptape.loaders import ImageLoader
from griptape.structures import Pipeline
from griptape.tasks import VariationImageGenerationTask
```

(continues on next page)

(continued from previous page)

```
prompt_artifact = TextArtifact("landscape photograph, verdant, countryside, 8k")
input_image_artifact = ImageLoader().load("tests/resources/mountain.png")

image_variation_task = VariationImageGenerationTask(
    input=(prompt_artifact, input_image_artifact),
    image_generation_driver=HuggingFacePipelineImageGenerationDriver(
        model="stabilityai/stable-diffusion-3-medium-diffusers",
        device="cuda",
        pipeline_driver=StableDiffusion3Img2ImgImageGenerationPipelineDriver(
            height=1024,
            width=1024,
        ),
    ),
)

output_artifact = Pipeline(tasks=[image_variation_task]).run().output
```

StableDiffusion3ControlNetImageGenerationPipelineDriver

Note

The *Stable Diffusion 3 Image Generation Pipeline Driver* requires the *drivers-image-generation-huggingface* extra.

The *StableDiffusion3ControlNetImageGenerationPipelineDriver* provides a `StableDiffusion3ControlNetPipeline` for image-to-image generations, accepting a text prompt and a control image. This Driver accepts a text prompt, a control image, and configurations including Stable Diffusion 3 model, ControlNet model, output image size, generation seed, inference steps, and the degree to which the model adheres to the control image.

```
from griptape.artifacts import TextArtifact
from griptape.drivers.image_generation.huggingface_pipeline import␣
↪HuggingFacePipelineImageGenerationDriver
from griptape.drivers.image_generation_pipeline.stable_diffusion_3_controlnet import (
    StableDiffusion3ControlNetImageGenerationPipelineDriver,
)
from griptape.loaders import ImageLoader
from griptape.structures import Pipeline
from griptape.tasks import VariationImageGenerationTask

prompt_artifact = TextArtifact("landscape photograph, verdant, countryside, 8k")
control_image_artifact = ImageLoader().load("canny_control_image.png")

controlnet_task = VariationImageGenerationTask(
    input=(prompt_artifact, control_image_artifact),
    image_generation_driver=HuggingFacePipelineImageGenerationDriver(
        model="stabilityai/stable-diffusion-3-medium-diffusers",
        device="cuda",
        pipeline_driver=StableDiffusion3ControlNetImageGenerationPipelineDriver(
            controlnet_model="InstantX/SD3-Controlnet-Canny",
            height=768,
            width=1024,
```

(continues on next page)

(continued from previous page)

```
        ),
    ),
)

output_artifact = Pipeline(tasks=[controlnet_task]).run().output
```

Observability

Overview

Observability Drivers are used by *Observability* to send telemetry (metrics and traces) related to the execution of an LLM application. The telemetry can be used to monitor the application and to diagnose and troubleshoot issues. All Observability Drivers implement the following methods:

- `__enter__()` sets up the Driver.
- `__exit__()` tears down the Driver.
- `observe()` wraps all functions and methods marked with the `@observable` decorator. At a bare minimum, implementations call the wrapped function and return its result (a no-op). This enables the Driver to generate telemetry related to the invocation's call arguments, return values, exceptions, latency, etc.

Observability Drivers

Gen AI Builder

The Gen AI Builder Observability Driver instruments `@observable` functions and methods with metrics and traces for use with the Gen AI Builder.

> **Note**
>
> For the Gen AI Builder Observability Driver to function as intended, it must be run from within either a Managed Structure on Gen AI Builder or locally via the Skatepark Emulator .

Here is an example of how to use the `GriptapeCloudObservabilityDriver` with the `Observability` context manager to send the telemetry to Gen AI Builder:

```
from griptape.drivers.observability.griptape_cloud import␣
↪GriptapeCloudObservabilityDriver
from griptape.observability import Observability
from griptape.rules import Rule
from griptape.structures import Agent

observability_driver = GriptapeCloudObservabilityDriver()

with Observability(observability_driver=observability_driver):
    agent = Agent(rules=[Rule("Output one word")])
    agent.run("Name an animal")
```

OpenTelemetry

The OpenTelemetry Observability Driver instruments `@observable` functions and methods with metrics and traces for use with OpenTelemetry. You must configure a destination for the telemetry by providing a `SpanProcessor` to the Driver.

Here is an example of how to use the `OpenTelemetryObservabilityDriver` with the `Observability` context manager to output the telemetry directly to the console:

```
from opentelemetry.sdk.trace.export import BatchSpanProcessor, ConsoleSpanExporter

from griptape.drivers.observability.open_telemetry import␣
↪OpenTelemetryObservabilityDriver
from griptape.observability import Observability
from griptape.rules import Rule
from griptape.structures import Agent

observability_driver = OpenTelemetryObservabilityDriver(
    service_name="name-an-animal", span_
↪processor=BatchSpanProcessor(ConsoleSpanExporter())
)

with Observability(observability_driver=observability_driver):
    agent = Agent(rules=[Rule("Output one word")])
    agent.run("Name an animal")
```

Rerank

Overview

Rerank Drivers can be used to rerank search results for a particular query. Every Rerank Driver implements the following methods that can be used directly:

- `run(query: str, artifacts: list *TextArtifact* based on the original query.

Rerank Drivers can also be used with a RagEngine's Rerank Module .

Rerank Drivers

Local

The *LocalRerankDriver* uses a simple relatedness calculation.

```
from griptape.artifacts import TextArtifact
from griptape.drivers.rerank.local import LocalRerankDriver

rerank_driver = LocalRerankDriver()

artifacts = rerank_driver.run(
    "What is the capital of France?",
    [
        TextArtifact("Hotdog"),
        TextArtifact("San Francisco"),
        TextArtifact("Paris"),
        TextArtifact("Baguette"),
```

(continues on next page)

(continued from previous page)

```
        TextArtifact("French Hotdog"),
    ],
)
for artifact in artifacts:
    print(artifact.value)
```

Cohere

The *CohereRerankDriver* uses *Cohere's Rerank* model.

```
import os

from griptape.artifacts import TextArtifact
from griptape.drivers.rerank.cohere import CohereRerankDriver

rerank_driver = CohereRerankDriver(
    api_key=os.environ["COHERE_API_KEY"],
)

artifacts = rerank_driver.run(
    "Where is NYC located?",
    [
        TextArtifact("NYC Media"),
        TextArtifact("New York City Police Department"),
        TextArtifact("New York City"),
        TextArtifact("New York City Subway"),
    ],
)
for artifact in artifacts:
    print(artifact.value)
```

Nvidia NIM

The *NvidiaNimRerankDriver* uses the *Nvidia NIM Reranking API* .

```
from griptape.artifacts import TextArtifact
from griptape.drivers.rerank.nvidia_nim import NvidiaNimRerankDriver

rerank_driver = NvidiaNimRerankDriver(
    model="nvidia/bert-base-uncased",
    base_url="http://localhost:8000",
)

artifacts = rerank_driver.run(
    "Where is NYC located?",
    [
        TextArtifact("NYC Media"),
        TextArtifact("New York City Police Department"),
        TextArtifact("New York City"),
        TextArtifact("New York City Subway"),
    ],
```

(continues on next page)

(continued from previous page)

```
)
for artifact in artifacts:
    print(artifact.value)
```

Ruleset

Overview

Ruleset Drivers can be used to load rules in from external sources.

Ruleset Drivers

Local

The *LocalRulesetDriver* allows you to load a Ruleset from a local JSON file. The `persist_dir` parameter is used to specify a local directory where one or more Ruleset files are located. If no `persist_dir` parameter is given, the `.load` method is a no-op.

```
import json
import os
from pathlib import Path

from griptape.drivers.ruleset.local import LocalRulesetDriver
from griptape.rules import Ruleset

ruleset_dir = "path/to/ruleset/dir"
ruleset_name = "my_local_ruleset.json"
ruleset_path = Path(os.path.join(ruleset_dir, ruleset_name))

os.makedirs(ruleset_dir, exist_ok=True)

ruleset_path.write_text(json.dumps({"rules": [{"value": "Always talk like a pirate."}
↪]}))

ruleset = Ruleset(
    name=ruleset_name,
    ruleset_driver=LocalRulesetDriver(persist_dir=ruleset_dir),
)
```

Gen AI Builder

The *GriptapeCloudRulesetDriver* allows you to load a Gen AI Builder Ruleset resource. `Ruleset.name` is used to try and find a Gen AI Builder Ruleset with that alias.

```
from griptape.drivers.ruleset.griptape_cloud import GriptapeCloudRulesetDriver
from griptape.rules import Ruleset

rulset = Ruleset(
    name="my_griptape_cloud_ruleset_alias",
    ruleset_driver=GriptapeCloudRulesetDriver(),
)
```

SQL

Overview

SQL drivers can be used to make SQL queries and load table schemas. They are used by the *SqlLoader* to process data. All loaders implement the following methods:

- `execute_query()` executes a query and returns *RowResultdataclass* s.
- `execute_query_row()` executes a query and returns a raw result from SQL.
- `get_table_schema()` returns a table schema.

SQL Drivers

SQL

Note that you may need to install the appropriate database driver for your SQL database. For example, to use the `psycopg2` driver for PostgreSQL, you can install it with `pip install psycopg2-binary`.

This is a basic SQL loader based on SQLAlchemy 2.0 . Here is an example of how to use it:

```
from griptape.drivers.sql.sql_driver import SqlDriver

driver = SqlDriver(engine_url="sqlite:///:memory:")

driver.execute_query("select foo, bar;")
```

Amazon Redshift

This is a SQL driver for interacting with the Amazon Redshift Data API

to execute statements. Here is an example of how to use it for Redshift Serverless:

```
import os

import boto3

from griptape.drivers.sql.amazon_redshift import AmazonRedshiftSqlDriver

session = boto3.Session()

driver = AmazonRedshiftSqlDriver(
    database=os.environ["REDSHIFT_DATABASE"],
    session=session,
    cluster_identifier=os.environ["REDSHIFT_CLUSTER_IDENTIFIER"],
)

driver.execute_query("select * from people;")
```

Snowflake

This is a SQL driver based on the *Snowflake SQLAlchemy Toolkit* which runs on top of the Snowflake Connector for Python. Here is an example of how to use it:

```
import os

import snowflake.connector
from snowflake.connector import SnowflakeConnection

from griptape.drivers.sql.snowflake import SnowflakeSqlDriver

def get_snowflake_connection() -> SnowflakeConnection:
    return snowflake.connector.connect(
        account=os.environ["SNOWFLAKE_ACCOUNT"],
        user=os.environ["SNOWFLAKE_USER"],
        password=os.environ["SNOWFLAKE_PASSWORD"],
        database=os.environ["SNOWFLAKE_DATABASE"],
        schema=os.environ["SNOWFLAKE_SCHEMA"],
        warehouse=os.environ["SNOWFLAKE_WAREHOUSE"],
    )

driver = SnowflakeSqlDriver(get_connection=get_snowflake_connection)

driver.execute_query("select * from people;")
```

Structure Run

Overview

Structure Run Drivers can be used to run Gen AI Builder Structures in a variety of runtime environments. When combined with the *Structure Run Task* or *Structure Run Tool* you can create complex, multi-agent pipelines that span multiple runtime environments.

Structure Run Drivers

Local

The *LocalStructureRunDriver* is used to run Gen AI Builder Structures in the same runtime environment as the code that is running the Structure.

```
from griptape.drivers.structure_run.local import LocalStructureRunDriver
from griptape.rules import Rule
from griptape.structures import Agent, Pipeline
from griptape.tasks import StructureRunTask

def build_joke_teller() -> Agent:
    return Agent(
        rules=[
            Rule(
                value="You are very funny.",
            )
        ],
    )

def build_joke_rewriter() -> Agent:
    return Agent(
        rules=[
```

(continues on next page)

(continued from previous page)

```
            Rule(
                value="You are the editor of a joke book. But you only speak in␣
↪riddles",
            )
        ],
    )

joke_coordinator = Pipeline(
    tasks=[
        StructureRunTask(
            structure_run_driver=LocalStructureRunDriver(
                create_structure=build_joke_teller,
            ),
        ),
        StructureRunTask(
            ("Rewrite this joke: {{ parent_output }}",),
            structure_run_driver=LocalStructureRunDriver(
                create_structure=build_joke_rewriter,
            ),
        ),
    ]
)

joke_coordinator.run("Tell me a joke")
```

Gen AI Builder

The *GriptapeCloudStructureRunDriver* is used to run Gen AI Builder Structures in the Gen AI Builder.

```
import os

from griptape.drivers.structure_run.griptape_cloud import␣
↪GriptapeCloudStructureRunDriver
from griptape.drivers.structure_run.local import LocalStructureRunDriver
from griptape.rules import Rule
from griptape.structures import Agent, Pipeline
from griptape.tasks import StructureRunTask

base_url = os.environ["GT_CLOUD_BASE_URL"]
api_key = os.environ["GT_CLOUD_API_KEY"]
structure_id = os.environ["GT_CLOUD_STRUCTURE_ID"]

pipeline = Pipeline(
    tasks=[
        StructureRunTask(
            ("Think of a question related to Retrieval Augmented Generation.",),
            structure_run_driver=LocalStructureRunDriver(
                create_structure=lambda: Agent(
                    rules=[
                        Rule(
                            value="You are an expert in Retrieval Augmented␣
↪Generation.",
```

(continues on next page)

(continued from previous page)

```
                        ),
                        Rule(
                            value="Only output your answer, no other information.",
                        ),
                    ]
                )
            ),
        ),
        StructureRunTask(
            ("{{ parent_output }}",),
            structure_run_driver=GriptapeCloudStructureRunDriver(
                base_url=base_url,
                api_key=api_key,
                structure_id=structure_id,
            ),
        ),
    ]
)

pipeline.run()
```

Text to Speech

Overview

Text to Speech Drivers are used to build and execute API calls to audio generation models.

Provide a Driver to a *RAG Tool* for use by an *agent* :

Text to Speech Drivers

Eleven Labs

The *Eleven Labs Text to Speech Driver* provides support for text-to-speech models hosted by Eleven Labs. This Driver supports configurations specific to Eleven Labs, like voice selection and output format.

```
import os

from griptape.drivers.text_to_speech.elevenlabs import ElevenLabsTextToSpeechDriver
from griptape.structures import Agent
from griptape.tools.text_to_speech.tool import TextToSpeechTool

driver = ElevenLabsTextToSpeechDriver(
    api_key=os.environ["ELEVEN_LABS_API_KEY"],
    model="eleven_multilingual_v2",
    voice="Matilda",
)

tool = TextToSpeechTool(
    text_to_speech_driver=driver,
)

Agent(tools=[tool]).run("Generate audio from this text: Hello, world!")
```

OpenAI

The *OpenAI Text to Speech Driver* provides support for text-to-speech models hosted by OpenAI. This Driver supports configurations specific to OpenAI, like voice selection and output format.

```
from griptape.drivers.text_to_speech.openai import OpenAiTextToSpeechDriver
from griptape.structures import Agent
from griptape.tools.text_to_speech.tool import TextToSpeechTool

driver = OpenAiTextToSpeechDriver()

tool = TextToSpeechTool(
    text_to_speech_driver=driver,
)

Agent(tools=[tool]).run("Generate audio from this text: Hello, world!")
```

Azure OpenAI

The *Azure OpenAI Text to Speech Driver* provides support for text-to-speech models hosted in your Azure OpenAI instance. This Driver supports configurations specific to OpenAI, like voice selection and output format.

```
import os

from griptape.drivers.text_to_speech.openai import AzureOpenAiTextToSpeechDriver
from griptape.structures import Agent
from griptape.tools.text_to_speech.tool import TextToSpeechTool

driver = AzureOpenAiTextToSpeechDriver(
    api_key=os.environ["AZURE_OPENAI_API_KEY_4"],
    model="tts",
    azure_endpoint=os.environ["AZURE_OPENAI_ENDPOINT_4"],
)

tool = TextToSpeechTool(
    text_to_speech_driver=driver,
)

Agent(tools=[tool]).run("Generate audio from this text: Hello, world!")
```

Vector Store

Overview

Gen AI Builder provides a way to build drivers for vector DBs where embeddings can be stored and queried. Every Vector Store Driver implements the following methods:

- `upsert()` for updating or inserting new text, *TextArtifact* s or *ImageArtifact* s into vector DBs. The method will automatically generate embeddings for a given value.
- `upsert_collection()` for performing an `upsert()` in parallel.
- `upsert_vector()` for updating new vectors directly.
- `query()` for querying vector DBs.

Each Vector Store Driver takes a *BaseEmbeddingDriver* used to dynamically generate embeddings for strings.

Vector Store Drivers

Local

The *LocalVectorStoreDriver* can be used to load and query data from memory. Here is a complete example of how the Driver can be used to load a webpage into the Driver and query it later:

```
import os

from griptape.chunkers import TextChunker
from griptape.drivers.embedding.openai import OpenAiEmbeddingDriver
from griptape.drivers.vector.local import LocalVectorStoreDriver
from griptape.loaders import WebLoader

#  Initialize an Embedding Driver

embedding_driver = OpenAiEmbeddingDriver(api_key=os.environ["OPENAI_API_KEY"])

vector_store_driver = LocalVectorStoreDriver(embedding_driver=embedding_driver)

#  Load Artifacts from the web

artifact = WebLoader().load("https://www.griptape.ai")
chunks = TextChunker(max_tokens=100).chunk(artifact)

#  Upsert Artifacts into the Vector Store Driver

vector_store_driver.upsert_collection({"griptape": chunks})

results = vector_store_driver.query(query="What is griptape?")

values = [r.to_artifact().value for r in results]

print("\n\n".join(values))
```

Gen AI Builder Knowledge Base

The *GriptapeCloudVectorStoreDriver* can be used to query data from a Gen AI Builder Knowledge Base. Loading into Knowledge Bases is not supported at this time, only querying. Here is a complete example of how the Driver can be used to query an existing Knowledge Base:

```
import os

from griptape.drivers.vector.griptape_cloud import GriptapeCloudVectorStoreDriver

#  Initialize environment variables

gt_cloud_api_key = os.environ["GT_CLOUD_API_KEY"]
gt_cloud_knowledge_base_id = os.environ["GT_CLOUD_KB_ID"]

vector_store_driver = GriptapeCloudVectorStoreDriver(
```

(continues on next page)

(continued from previous page)

```
    api_key=gt_cloud_api_key, knowledge_base_id=gt_cloud_knowledge_base_id
)

results = vector_store_driver.query(query="What is griptape?")

values = [r.to_artifact().value for r in results]

print("\n\n".join(values))
```

Pinecone

The *PineconeVectorStoreDriver* supports the *Pinecone vector database* .

Here is an example of how the Driver can be used to load and query information in a Pinecone cluster:

```
import os

from griptape.chunkers import TextChunker
from griptape.drivers.embedding.openai import OpenAiEmbeddingDriver
from griptape.drivers.vector.pinecone import PineconeVectorStoreDriver
from griptape.loaders import WebLoader

NAMESPACE = "griptape"

#  Initialize an Embedding Driver

embedding_driver = OpenAiEmbeddingDriver(api_key=os.environ["OPENAI_API_KEY"])

vector_store_driver = PineconeVectorStoreDriver(
    api_key=os.environ["PINECONE_API_KEY"],
    environment=os.environ["PINECONE_ENVIRONMENT"],
    index_name=os.environ["PINECONE_INDEX_NAME"],
    embedding_driver=embedding_driver,
)

#  Load Artifacts from the web

artifact = WebLoader().load("https://www.griptape.ai")
chunks = TextChunker(max_tokens=100).chunk(artifact)

#  Upsert Artifacts into the Vector Store Driver

vector_store_driver.upsert_collection({NAMESPACE: chunks})

results = vector_store_driver.query(query="What is griptape?", namespace=NAMESPACE)

values = [r.to_artifact().value for r in results]

print("\n\n".join(values))
```

Marqo

The *MarqoVectorStoreDriver* supports the Marqo vector database.

Here is an example of how the Driver can be used to load and query information in a Marqo cluster:

```
import os

from griptape.chunkers import TextChunker
from griptape.drivers.embedding.openai import OpenAiEmbeddingDriver
from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.drivers.vector.marqo import MarqoVectorStoreDriver
from griptape.loaders import WebLoader

#  Initialize an Embedding Driver

embedding_driver = OpenAiEmbeddingDriver(api_key=os.environ["OPENAI_API_KEY"])
prompt_driver = OpenAiChatPromptDriver(model="gpt-3.5-turbo")

#  Define the namespace

namespace = "griptape-ai"

#  Initialize the Vector Store Driver

vector_store_driver = MarqoVectorStoreDriver(
    api_key=os.environ["MARQO_API_KEY"],
    url=os.environ["MARQO_URL"],
    index=os.environ["MARQO_INDEX_NAME"],
    embedding_driver=embedding_driver,
)

#  Load Artifacts from the web

artifact = WebLoader().load("https://www.griptape.ai")
chunks = TextChunker(max_tokens=200).chunk(artifact)

#  Upsert Artifacts into the Vector Store Driver

vector_store_driver.upsert_collection(
    {
        "griptape": chunks,
    }
)

results = vector_store_driver.query(query="What is griptape?")

values = [r.to_artifact().value for r in results]

print("\n\n".join(values))
```

Mongodb Atlas

The *MongoDbAtlasVectorStoreDriver* provides support for storing vector data in a MongoDB Atlas database.

Here is an example of how the Driver can be used to load and query information in a MongoDb Atlas Cluster:

```
import os

from griptape.chunkers import TextChunker
from griptape.drivers.embedding.openai import OpenAiEmbeddingDriver
from griptape.drivers.vector.mongodb_atlas import MongoDbAtlasVectorStoreDriver
from griptape.loaders import WebLoader

#  Initialize an Embedding Driver

embedding_driver = OpenAiEmbeddingDriver(api_key=os.environ["OPENAI_API_KEY"])

host = os.environ["MONGODB_HOST"]
username = os.environ["MONGODB_USERNAME"]
password = os.environ["MONGODB_PASSWORD"]
database_name = os.environ["MONGODB_DATABASE_NAME"]
collection_name = os.environ["MONGODB_COLLECTION_NAME"]
index_name = os.environ["MONGODB_INDEX_NAME"]
vector_path = os.environ["MONGODB_VECTOR_PATH"]

#  Initialize the Vector Store Driver

vector_store_driver = MongoDbAtlasVectorStoreDriver(
    connection_string=f"mongodb+srv://{username}:{password}@{host}/{database_name}",
    database_name=database_name,
    collection_name=collection_name,
    embedding_driver=embedding_driver,
    index_name=index_name,
    vector_path=vector_path,
)

#  Load Artifacts from the web

artifact = WebLoader().load("https://www.griptape.ai")
chunks = TextChunker(max_tokens=200).chunk(artifact)

#  Upsert Artifacts into the Vector Store Driver

vector_store_driver.upsert_collection({"griptape": chunks})

results = vector_store_driver.query(query="What is griptape?")

values = [r.to_artifact().value for r in results]

print("\n\n".join(values))
```

The format for creating a vector index should look similar to the following:

```
{
```

(continues on next page)

(continued from previous page)

```
  "fields": [
    {
      "numDimensions": 1536,
      "path": "<path_to_vector>",
      "similarity": "euclidean",
      "type": "vector"
    },
    {
      "path": "namespace",
      "type": "filter"
    }
  ]
}
```

Replace `path_to_vector` with the expected field name where the vector content will be.

Azure MongoDB

The *AzureMongoDbVectorStoreDriver* provides support for storing vector data in an Azure CosmosDb database account using the MongoDb vCore API

Here is an example of how the Driver can be used to load and query information in an Azure CosmosDb MongoDb vCore database. It is very similar to the Driver for *Mongodb Atlas* :

```
import os

from griptape.chunkers import TextChunker
from griptape.drivers.embedding.openai import OpenAiEmbeddingDriver
from griptape.drivers.vector.azure_mongodb import AzureMongoDbVectorStoreDriver
from griptape.loaders import WebLoader

#  Initialize an Embedding Driver

embedding_driver = OpenAiEmbeddingDriver(api_key=os.environ["OPENAI_API_KEY"])

azure_host = os.environ["AZURE_MONGODB_HOST"]
username = os.environ["AZURE_MONGODB_USERNAME"]
password = os.environ["AZURE_MONGODB_PASSWORD"]
database_name = os.environ["AZURE_MONGODB_DATABASE_NAME"]
collection_name = os.environ["AZURE_MONGODB_COLLECTION_NAME"]
index_name = os.environ["AZURE_MONGODB_INDEX_NAME"]
vector_path = os.environ["AZURE_MONGODB_VECTOR_PATH"]

#  Initialize the Vector Store Driver

vector_store_driver = AzureMongoDbVectorStoreDriver(
    connection_string=f"mongodb+srv://{username}:{password}@{azure_host}/{database_
↪name}?tls=true&authMechanism=SCRAM-SHA-256&retrywrites=false&maxIdleTimeMS=120000",
    database_name=database_name,
    collection_name=collection_name,
    embedding_driver=embedding_driver,
    index_name=index_name,
    vector_path=vector_path,
```

(continues on next page)

(continued from previous page)

```
)

#  Load Artifacts from the web

artifact = WebLoader().load("https://www.griptape.ai")
chunks = TextChunker(max_tokens=200).chunk(artifact)

#  Upsert Artifacts into the Vector Store Driver

vector_store_driver.upsert_collection(
    {
        "griptape": chunks,
    }
)

results = vector_store_driver.query(query="What is griptape?")

values = [r.to_artifact().value for r in results]

print("\n\n".join(values))
```

Redis

The *RedisVectorStoreDriver* integrates with the Redis vector storage system.

Here is an example of how the Driver can be used to load and query information in a Redis Cluster:

```
import os

from griptape.chunkers import TextChunker
from griptape.drivers.embedding.openai import OpenAiEmbeddingDriver
from griptape.drivers.vector.redis import RedisVectorStoreDriver
from griptape.loaders import WebLoader

#  Initialize an Embedding Driver

embedding_driver = OpenAiEmbeddingDriver(api_key=os.environ["OPENAI_API_KEY"])

vector_store_driver = RedisVectorStoreDriver(
    host=os.environ["REDIS_HOST"],
    port=int(os.environ["REDIS_PORT"]),
    username=os.environ["REDIS_USERNAME"],
    password=os.environ["REDIS_PASSWORD"],
    index=os.environ["REDIS_INDEX"],
    embedding_driver=embedding_driver,
)

#  Load Artifacts from the web

artifact = WebLoader().load("https://www.griptape.ai")
chunks = TextChunker(max_tokens=200).chunk(artifact)
```

(continues on next page)

(continued from previous page)

```
#  Upsert Artifacts into the Vector Store Driver

vector_store_driver.upsert_collection(
    {
        "griptape": chunks,
    }
)

results = vector_store_driver.query(query="What is griptape?")

values = [r.to_artifact().value for r in results]

print("\n\n".join(values))
```

The format for creating a vector index should be similar to the following:

```
FT.CREATE idx:griptape ON hash PREFIX 1 "griptape:" SCHEMA namespace TAG vector␣
↪VECTOR FLAT 6 TYPE FLOAT32 DIM 1536 DISTANCE_METRIC COSINE
```

OpenSearch

The *AmazonOpenSearchVectorStoreDriver* integrates with the OpenSearch platform, allowing for storage, retrieval, and querying of vector data.

Here is an example of how the Driver can be used to load and query information in an OpenSearch Cluster:

```
import os

import boto3

from griptape.chunkers import TextChunker
from griptape.drivers.embedding.openai import OpenAiEmbeddingDriver
from griptape.drivers.vector.amazon_opensearch import␣
↪AmazonOpenSearchVectorStoreDriver
from griptape.loaders import WebLoader

#  Initialize an Embedding Driver

embedding_driver = OpenAiEmbeddingDriver(api_key=os.environ["OPENAI_API_KEY"])

vector_store_driver = AmazonOpenSearchVectorStoreDriver(
    host=os.environ["AMAZON_OPENSEARCH_HOST"],
    index_name=os.environ["AMAZON_OPENSEARCH_INDEX_NAME"],
    session=boto3.Session(),
    embedding_driver=embedding_driver,
)

#  Load Artifacts from the web

artifact = WebLoader().load("https://www.griptape.ai")
chunks = TextChunker(max_tokens=200).chunk(artifact)
```

(continues on next page)

(continued from previous page)

```
#  Upsert Artifacts into the Vector Store Driver

vector_store_driver.upsert_collection(
    {
        "griptape": chunks,
    }
)

results = vector_store_driver.query(query="What is griptape?")

values = [r.to_artifact().value for r in results]

print("\n\n".join(values))
```

The body mappings for creating a vector index should look similar to the following:

```
{
    "mappings": {
        "properties": {
            "vector": {"type": "knn_vector", "dimension": 1536},
            "namespace": {"type": "keyword"},
            "metadata": {"type": "object", "enabled": true}
        }
    }
}
```

PGVector

The *PgVectorVectorStoreDriver* integrates with PGVector, a vector storage and search extension for Postgres. While Gen AI Builder will handle enabling the extension, PGVector must be installed and ready for use in your Postgres instance before using this Vector Store Driver.

Here is an example of how the Driver can be used to load and query information in a Postgres database:

```
import os

from griptape.chunkers import TextChunker
from griptape.drivers.embedding.openai import OpenAiEmbeddingDriver
from griptape.drivers.vector.pgvector import PgVectorVectorStoreDriver
from griptape.loaders import WebLoader

#  Initialize an Embedding Driver.

embedding_driver = OpenAiEmbeddingDriver(api_key=os.environ["OPENAI_API_KEY"])

db_user = os.environ["POSTGRES_USER"]
db_pass = os.environ["POSTGRES_PASSWORD"]
db_host = os.environ["POSTGRES_HOST"]
db_port = os.environ["POSTGRES_PORT"]
db_name = os.environ["POSTGRES_DB"]
db_connection_string = f"postgresql://{db_user}:{db_pass}@{db_host}:{db_port}/{db_
↪name}"
```

(continues on next page)

(continued from previous page)

```
vector_store_driver = PgVectorVectorStoreDriver(
    connection_string=db_connection_string,
    embedding_driver=embedding_driver,
    table_name="griptape_vectors",
)

#  Install required Postgres extensions and create database schema.

vector_store_driver.setup()

#  Load Artifacts from the web

artifact = WebLoader().load("https://www.griptape.ai")
chunks = TextChunker().chunk(artifact)

#  Upsert Artifacts into the Vector Store Driver

vector_store_driver.upsert_collection(
    {
        "griptape": chunks,
    }
)

results = vector_store_driver.query(query="What is griptape?")

values = [r.to_artifact().value for r in results]

print("\n\n".join(values))
```

Qdrant

The QdrantVectorStoreDriver supports the *Qdrant vector database* .

Here is an example of how the Driver can be used to query information in a Qdrant collection:

```
import os

from qdrant_client.http.models import Distance, VectorParams

from griptape.chunkers import TextChunker
from griptape.drivers.embedding.openai import OpenAiEmbeddingDriver
from griptape.drivers.vector.qdrant import QdrantVectorStoreDriver
from griptape.loaders import WebLoader

#  Set up environment variables

host = os.environ["QDRANT_CLUSTER_ENDPOINT"]
api_key = os.environ["QDRANT_CLUSTER_API_KEY"]

#  Initialize an Embedding Driver.

embedding_driver = OpenAiEmbeddingDriver(api_key=os.environ["OPENAI_API_KEY"])
```

(continues on next page)

(continued from previous page)

```
vector_store_driver = QdrantVectorStoreDriver(
    url=host,
    collection_name="griptape",
    content_payload_key="content",
    embedding_driver=embedding_driver,
    api_key=api_key,
)

#  Load Artifacts from the web

artifact = WebLoader().load("https://www.griptape.ai")
chunks = TextChunker(max_tokens=100).chunk(artifact)

#  Recreate Qdrant collection

vector_store_driver.client.recreate_collection(
    collection_name=vector_store_driver.collection_name,
    vectors_config=VectorParams(size=1536, distance=Distance.COSINE),
)

#  Upsert Artifacts into the Vector Store Driver

vector_store_driver.upsert_collection({"griptape": chunks})

results = vector_store_driver.query(query="What is griptape?")

values = [r.to_artifact().value for r in results]

print("\n\n".join(values))
```

Astra DB

The AstraDbVectorStoreDriver supports DataStax Astra DB .

The following example shows how to store vector entries and query the information using the driver:

```
import os

from griptape.chunkers import TextChunker
from griptape.drivers.embedding.openai import OpenAiEmbeddingDriver
from griptape.drivers.vector.astradb import AstraDbVectorStoreDriver
from griptape.loaders import WebLoader

#  Astra DB secrets and connection parameters

api_endpoint = os.environ["ASTRA_DB_API_ENDPOINT"]
token = os.environ["ASTRA_DB_APPLICATION_TOKEN"]
astra_db_namespace = os.environ.get("ASTRA_DB_KEYSPACE")  # optional

#  Initialize an Embedding Driver.
```

(continues on next page)

(continued from previous page)

```
embedding_driver = OpenAiEmbeddingDriver(api_key=os.environ["OPENAI_API_KEY"])

vector_store_driver = AstraDbVectorStoreDriver(
    embedding_driver=embedding_driver,
    api_endpoint=api_endpoint,
    token=token,
    collection_name="griptape_test_collection",
    astra_db_namespace=astra_db_namespace,  # optional
)

#  Load Artifacts from the web

artifact = WebLoader().load("https://www.griptape.ai")
chunks = TextChunker().chunk(artifact)

#  Upsert Artifacts into the Vector Store Driver

vector_store_driver.upsert_collection({"griptape": chunks})

results = vector_store_driver.query(query="What is griptape?")

values = [r.to_artifact().value for r in results]

print("\n\n".join(values))
```

PG.AI

The PgAiKnowledgeBaseVectorStoreDriver integrates with PG.AI, a managed postgres platform from EnterpriseDB .

Here is an example of how the Driver can be used to load and query information in a PG.AI Knowledge Base:

```
import os

from griptape.drivers.vector.pgai import PgAiKnowledgeBaseVectorStoreDriver

#  PG.AI connection parameters

connection_string = os.environ["PGAI_CONNECTION_STRING"]
knowledge_base_name = os.environ["PGAI_KNOWLEDGE_BASE_NAME"]

vector_store_driver = PgAiKnowledgeBaseVectorStoreDriver(
    connection_string=connection_string,
    knowledge_base_name=knowledge_base_name,  # optional
)

results = vector_store_driver.query(query="What is griptape?")

values = [r.to_artifact().value for r in results]

print("\n\n".join(values))
```

Web Scraper

Overview

Web Scraper Drivers can be used to scrape text from the web. They are used by *WebLoader* to provide its functionality. All Web Scraper Drivers implement the following methods:

- `scrape_url()` scrapes text from a website and returns a *TextArtifact* . The format of the scrapped text is determined by the Driver.

Web Scraper Drivers

Proxy

The *ProxyWebScraperDriver* uses the `requests` library with a provided set of proxies to do web scraping. Paid webscraping services like ZenRows or ScraperAPI offer a way to use their API via a set of proxies passed to `requests.get()`

Example using `ProxyWebScraperDriver` directly:

```
import os

from griptape.drivers.web_scraper.proxy import ProxyWebScraperDriver

query_params = [
    "markdown_response=true",
    "js_render=false",
    "premium_proxy=false",
]
proxy_url = f"http://{os.environ[ZENROWS_API_KEY]}:{&.join(query_params)}@proxy.
↪zenrows.com:8001"

driver = ProxyWebScraperDriver(
    proxies={
        "http": proxy_url,
        "https": proxy_url,
    },
    params={"verify": False},
)

driver.scrape_url("https://griptape.ai")
```

Markdownify

To install the playwright browsers, run `playwright install` in your terminal. If you are using uv, run `uv run playwright install` instead. The `playwright` command should already be installed as a dependency of the `drivers-web-scraper-markdownify` extra. For more details about playwright, see the playwright docs .

Note that if you skip installing the playwright browsers, you will see the following error when you run your code:

```
playwright._impl._errors.Error: Executable doesnt exist at ...
╔═══════════════════════════════════════════════════════════╗
║ Looks like Playwright was just installed or updated.      ║
║ Please run the following command to download new browsers: ║
║                                                           ║
```

(continues on next page)

(continued from previous page)

```
╟     playwright install                                                    ╢
╟                                                                           ╢
╟ <3 Playwright Team                                                       ╢
╚═══════════════════════════════════════════════════════════════════════════╝
```

The *MarkdownifyWebScraperDriver* outputs the scraped text in markdown format. It uses playwright to render web pages along with dynamically loaded content, and a combination of beautifulsoup4 and markdownify to produce a markdown representation of a webpage. It makes a best effort to produce a markdown representation of a webpage that is concise yet human (and LLM) readable.

Example using `MarkdownifyWebScraperDriver` directly:

```
from griptape.drivers.web_scraper.markdownify import MarkdownifyWebScraperDriver

driver = MarkdownifyWebScraperDriver()

driver.scrape_url("https://griptape.ai")
```

Example of using `MarkdownifyWebScraperDriver` with an agent:

```
from griptape.drivers.web_scraper.markdownify import MarkdownifyWebScraperDriver
from griptape.loaders import WebLoader
from griptape.structures import Agent
from griptape.tools import WebScraperTool

agent = Agent(
    tools=[
        WebScraperTool(
            web_loader=WebLoader(web_scraper_
↪driver=MarkdownifyWebScraperDriver(timeout=1000)),
            off_prompt=False,
        ),
    ],
)
agent.run("List all email addresses on griptape.ai in a flat numbered markdown list.")
```

Trafilatura

The *TrafilaturaWebScraperDriver* scrapes text from a webpage using the Trafilatura library.

Example of using `TrafilaturaWebScraperDriver` directly:

```
from griptape.drivers.web_scraper.trafilatura import TrafilaturaWebScraperDriver

driver = TrafilaturaWebScraperDriver()

driver.scrape_url("https://griptape.ai")
```

Web Search

Overview

Web Search Drivers can be used to search for links from a search query. They are used by *BaseWebSearchDriver* to provide its functionality. All Web Search Drivers implement the following methods:

- `search()` searches the web and returns a *ListArtifact* that contains JSON-serializable *TextArtifact* s with the search results.

You can use Web Search Drivers with *Structures* :

```
from griptape.drivers.web_search.duck_duck_go import DuckDuckGoWebSearchDriver
from griptape.structures import Agent
from griptape.tools import PromptSummaryTool, WebSearchTool

agent = Agent(
    tools=[WebSearchTool(web_search_driver=DuckDuckGoWebSearchDriver()),␣
↪PromptSummaryTool(off_prompt=False)],
)

agent.run("Give me some websites with information about AI frameworks.")
```

```
PromptTask 45a53f1024494baab41a1f10a67017b1
    Output: Here are some websites with information about AI
    frameworks:

      1. [The Top 16 AI Frameworks and Libraries: A Beginners Guide -      ␣
↪DataCamp](https://www.datacamp.com/blog/top-ai-frameworks-and-lib      raries)
      2. [AI Frameworks: Top Types To Adopt in 2024 -      Splunk](https://www.splunk.
↪com/en_us/blog/learn/ai-frameworks.htm      l)
      3. [Top AI Frameworks in 2024: A Review -      BairesDev](https://www.bairesdev.
↪com/blog/ai-frameworks/)
      4. [The Top 16 AI Frameworks and Libraries - AI      Slackers](https://
↪aislackers.com/the-top-16-ai-frameworks-and-lib      raries/)
      5. [Top AI Frameworks in 2024: Artificial Intelligence Frameworks      ␣
↪Comparison - Clockwise      Software](https://clockwise.software/blog/artificial-
↪intelligence      -framework/)
```

Or use them independently:

```
from griptape.drivers.web_search.duck_duck_go import DuckDuckGoWebSearchDriver

driver = DuckDuckGoWebSearchDriver()

driver.search("griptape ai")
```

```
{"title": "The Top 16 AI Frameworks and Libraries: A Beginners Guide", "url": "https:/
↪/www.datacamp.com/blog/top-ai-frameworks-and-libraries", "description": "PyTorch.␣
↪Torch is an open-source machine learning library known for its dynamic␣
↪computational graph and is favored by researchers. The framework is excellent for␣
↪prototyping and experimentation. Moreover, its empowered by growing community␣
↪support, with tools like PyTorch being built on the library."}

{"title": "Top 11 AI Frameworks and Tools in 2024 | Fively | 5ly.co", "url": "https://
↪5ly.co/blog/best-ai-frameworks/", "description": "Discover the top 11 modern␣
↪artificial intelligence tools and frameworks to build robust architectures for your␣
↪AI-powered apps. ... - Some advanced use cases may need further fine-tuning. Caffe␣
↪2. Now we move on to deep learning tools and frameworks. The first one is Caffe 2:␣
↪an open-source deep learning framework with modularity and speed in ..."}
```

(continues on next page)

(continued from previous page)

```
{"title": "The Top 16 AI Frameworks and Libraries | AI Slackers", "url": "https://
↪aislackers.com/the-top-16-ai-frameworks-and-libraries/", "description": "Experiment␣
↪with different frameworks to find the one that aligns with your needs and goals as␣
↪a data practitioner. Embrace the world of AI frameworks, and embark on a journey of␣
↪building smarter software with confidence. Discover the top AI frameworks and␣
↪libraries like PyTorch, Scikit-Learn, TensorFlow, Keras, LangChain, and more."}
```

Web Search Drivers

Google

The *GoogleWebSearchDriver* uses the *Google Custom Search JSON API* for web searching.

```
import os

from griptape.drivers.web_search.google import GoogleWebSearchDriver

driver = GoogleWebSearchDriver(
    api_key=os.environ["GOOGLE_API_KEY"],
    search_id=os.environ["GOOGLE_API_SEARCH_ID"],
)

driver.search("griptape ai")
```

DuckDuckGo

The *DuckDuckGoWebSearchDriver* uses the duckduckgo_search SDK for web searching.

```
from griptape.drivers.web_search.duck_duck_go import DuckDuckGoWebSearchDriver

driver = DuckDuckGoWebSearchDriver()

driver.search("griptape ai")
```

Tavily

```
import os

from griptape.drivers.web_search.tavily import TavilyWebSearchDriver

driver = TavilyWebSearchDriver(api_key=os.environ["TAVILY_API_KEY"])

driver.search("griptape ai")
```

Exa

```
import os

from griptape.drivers.web_search.exa import ExaWebSearchDriver

```

(continues on next page)

(continued from previous page)

```
driver = ExaWebSearchDriver(api_key=os.environ["EXA_API_KEY"])

driver.search("griptape ai")
```

Perplexity

The *PerplexityWebSearchDriver* uses the *Perplexity Prompt Driver* internally.

```
import os

from griptape.drivers.web_search.perplexity import PerplexityWebSearchDriver

driver = PerplexityWebSearchDriver(
    api_key=os.environ["PERPLEXITY_API_KEY"],
)
result = driver.search("griptape ai")

print(result.to_text())

#  The search results will only contain a single Artifact.

#  This Artifact contains the search results and any citations.

for citation in result[0].meta["citations"]:
    print(citation)
```

`

2. Set up your OpenAI API key:

```
export OPENAI_API_KEY=your_api_key_here
```

3. Create a Python file (e.g., `app.py`) and import necessary modules from Griptape[3].
4. Build your AI application using Griptape's components such as agents, pipelines, and tools[3].

Griptape offers a powerful and flexible way to create AI-driven applications, from simple chatbots to complex multi-agent systems. Its modular approach allows developers to easily integrate custom tools and external resources, enhancing the capabilities of AI agents[3][4]. https://www.griptape.ai https://github.com/griptape-ai https://www.youtube.com/watch?v=AEcztigpnc0 https://www.youtube.com/watch?v=1FFeMZyWxJA https://github.com/griptape-ai/griptape https://www.youtube.com/@griptape-ai

````

## 11.3 RAG Engine

### 11.3.1 RAG Engines

*RAG Engine* is an abstraction for implementing modular retrieval augmented generation (RAG) pipelines.
````

RAG Stages

`RagEngine` s consist of three *stages*: `QueryRagStage` , `RetrievalRagStage` , and `ResponseRagStage` . These stages are always executed sequentially. Each stage comprises multiple *modules*, which are executed in a customized manner. Due to this unique structure, `RagEngines` are not intended to replace *Workflows* or *HuggingFace Pipelines* .

RAG Modules

RAG modules are used to implement actions in the different stages of the RAG pipeline. `RagEngine` enables developers to easily add new modules to experiment with novel RAG strategies.

The three stages of the pipeline implemented in RAG Engines, together with their purposes and associated modules, are as follows:

Query Stage

This stage is used for modifying input queries before they are submitted.

Query Stage Modules

- `TranslateQueryRagModule` is for translating the query into another language.

Retrieval Stage

Results are retrieved in this stage, either from a vector store in the form of chunks, or with a text loader. You may optionally use a rerank module in this stage to rerank results in order of their relevance to the original query.

Retrieval Stage Modules

- `TextChunksRerankRagModule` is for re-ranking retrieved results.
- `TextLoaderRetrievalRagModule` is for retrieving data with text loaders in real time.
- `VectorStoreRetrievalRagModule` is for retrieving text chunks from a vector store.

Response Stage

Responses are generated in this final stage.

Response Stage Modules

- `PromptResponseRagModule` is for generating responses based on retrieved text chunks.
- `TextChunksResponseRagModule` is for responding with retrieved text chunks.
- `FootnotePromptResponseRagModule` is for responding with automatic footnotes from text chunk references.

RAG Context

`RagContext` is a container object for passing around queries, text chunks, module configs, and other metadata. `RagContext` is modified by modules when appropriate. Some modules support runtime config overrides through `RagContext.module_configs` .

Simple Example

The following example shows a simple RAG pipeline that retrieves data from a local vector store and generates a response:

```
from griptape.chunkers import TextChunker
from griptape.drivers.embedding.openai import OpenAiEmbeddingDriver
from griptape.drivers.vector.local import LocalVectorStoreDriver
from griptape.engines.rag import RagContext, RagEngine
from griptape.engines.rag.modules import (
    PromptResponseRagModule,
    VectorStoreRetrievalRagModule,
)
from griptape.engines.rag.stages import (
    ResponseRagStage,
    RetrievalRagStage,
)
from griptape.loaders import WebLoader

vector_store = LocalVectorStoreDriver(embedding_driver=OpenAiEmbeddingDriver())

#  Load some data from a couple sources.

web_artifact = WebLoader().load("https://griptape.ai")
chunks = TextChunker(max_tokens=250).chunk(web_artifact)
vector_store.upsert_collection({"griptape": chunks})

rag_engine = RagEngine(
    # This stage is responsible for retrieving the relevant chunks.
    retrieval_stage=RetrievalRagStage(
        retrieval_modules=[
            VectorStoreRetrievalRagModule(
                name="WebRetriever",
                vector_store_driver=vector_store,
                query_params={"namespace": "griptape"},
            ),
        ],
    ),
    # This stage is responsible for generating the final response.
    response_stage=ResponseRagStage(
        response_modules=[
            PromptResponseRagModule(),
        ]
    ),
)

rag_context = RagContext(query="What is Griptape?")
rag_context = rag_engine.process(rag_context)
print(rag_context.outputs[0].to_text())
```

Advanced Example

The following example shows an advanced RAG pipeline that does the following:

- Translates incoming queries into English.
- Retrieves data from a local vector store and a local file.
- Reranks the results using the local rerank driver.

- Generates multiple types of response.

```
from rich.pretty import pprint

from griptape.chunkers import TextChunker
from griptape.common.reference import Reference
from griptape.drivers.embedding.openai import OpenAiEmbeddingDriver
from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.drivers.rerank.local import LocalRerankDriver
from griptape.drivers.vector.local import LocalVectorStoreDriver
from griptape.engines.rag import RagContext, RagEngine
from griptape.engines.rag.modules import (
    FootnotePromptResponseRagModule,
    PromptResponseRagModule,
    TextChunksRerankRagModule,
    TextChunksResponseRagModule,
    TextLoaderRetrievalRagModule,
    TranslateQueryRagModule,
    VectorStoreRetrievalRagModule,
)
from griptape.engines.rag.stages import (
    QueryRagStage,
    ResponseRagStage,
    RetrievalRagStage,
)
from griptape.loaders import TextLoader, WebLoader
from griptape.rules import Rule, Ruleset

prompt_driver = OpenAiChatPromptDriver(model="gpt-4.1")
vector_store = LocalVectorStoreDriver(embedding_driver=OpenAiEmbeddingDriver())
rerank_driver = LocalRerankDriver()
web_loader = WebLoader()
text_chunker = TextChunker(max_tokens=250)

#  Load some data from a couple sources.

sites = [
    {
        "title": "Griptape Site",
        "url": "https://www.griptape.ai",
    },
    {"title": "Griptape GitHub", "url": "https://github.com/griptape-ai/griptape"},
]
site_artifacts = list(web_loader.load_collection([site["url"] for site in sites]).
↪values())

#  Set a reference on each artifact so that the FootnotePromptResponseRagModule can␣
↪generate footnotes.

for site_arifact, site in zip(site_artifacts, sites):
    site_arifact.reference = Reference(title=site["title"], url=site["url"])

#  Chunk each site artifact.
```

(continues on next page)

(continued from previous page)

```
site_artifacts_chunks = [text_chunker.chunk(artifact) for artifact in site_artifacts]

#  Flatten the list of chunks before inserting them into the vector store.

site_artifacts_chunks = [
    site_artifact_chunk for site_artifact_chunk in site_artifacts_chunks for site_
↪artifact_chunk in site_artifact_chunk
]
vector_store.upsert_collection({"griptape": site_artifacts_chunks})

rag_engine = RagEngine(
    # This stage is responsible for producing the query. It can include things like␣
↪translation, rewriting, etc.
    query_stage=QueryRagStage(query_modules=[TranslateQueryRagModule(prompt_
↪driver=prompt_driver, language="english")]),
    # This stage is responsible for retrieving the relevant chunks.
    retrieval_stage=RetrievalRagStage(
        max_chunks=5,
        retrieval_modules=[  # Modules can pull from different sources.
            # Such as a vector store
            VectorStoreRetrievalRagModule(
                name="WebRetriever",
                vector_store_driver=vector_store,
                query_params={"top_n": 20, "namespace": "griptape"},
            ),
            # Or a text source.
            TextLoaderRetrievalRagModule(
                loader=TextLoader(),
                vector_store_driver=vector_store,
                source="README.md",
            ),
        ],
        # We can rerank the chunks before passing them to the response stage to␣
↪ensure the best ones are used.
        rerank_module=TextChunksRerankRagModule(rerank_driver=LocalRerankDriver()),
    ),
    # This stage is responsible for generating the final response.
    response_stage=ResponseRagStage(
        response_modules=[
            # You can have multiple response modules to generate different types of␣
↪responses.
            PromptResponseRagModule(
                prompt_driver=prompt_driver,
                rulesets=[Ruleset(name="Concise", rules=[Rule("Answer concisely")])],
            ),
            # This one generates a response with footnotes.
            FootnotePromptResponseRagModule(prompt_driver=prompt_driver),
            # This one just returns the text chunks directly with no LLM generation.
            TextChunksResponseRagModule(),
        ]
    ),
```

(continues on next page)

(continued from previous page)

```
)

rag_context = RagContext(
    query="¿Qué ofrecen los servicios en la nube de Griptape?",  # What do Griptapes␣
↪cloud services offer?
)
rag_context = rag_engine.process(rag_context)

#  Lets print out the interesting responses

for output in rag_context.outputs[:2]:
    print(output.to_text() + "\n")

#  We can also see the references that were configured up-front

pprint(rag_context.get_references())
```

RAG Tool

See *RagTool* for an example of how to integrate the RAG Tool with an Agent.

Eval Engine

Overview

The *Eval Engine* is used to evaluate the performance of an LLM's output against a given input. The engine returns a score between 0 and 1, along with a reason for the score.

Eval Engines require either validate*criteria(*, value) or validate*evaluation_steps(*, value) to be set. If `criteria` is set, Gen AI Builder will generate `evaluation_steps` for you. This is useful for getting started, but you may to explicitly set `evaluation_steps` for more complex evaluations. Either `criteria` or `evaluation_steps` must be set, but not both.

```
import json

from griptape.engines import EvalEngine

engine = EvalEngine(
    criteria="Determine whether the actual output is factually correct based on the␣
↪expected output.",
)

score, reason = engine.evaluate(
    input="If you have a red house made of red bricks, a blue house made of blue␣
↪bricks, what is a greenhouse made of?",
    expected_output="Glass",
    actual_output="Glass",
)

print("Eval Steps", json.dumps(engine.evaluation_steps, indent=2))
print(f"Score: {score}")
print(f"Reason: {reason}")
```

Extraction Engines

Overview

Extraction Engines enable the extraction of structured data from unstructured text. These Engines can be used with Structures via *Extraction Tasks* . As of now, Gen AI Builder supports two types of Extraction Engines: the CSV Extraction Engine and the JSON Extraction Engine.

CSV

The CSV Extraction Engine extracts comma separated values from unstructured text.

```
from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.engines import CsvExtractionEngine

#  Initialize the CsvExtractionEngine instance

csv_engine = CsvExtractionEngine(
    prompt_driver=OpenAiChatPromptDriver(model="gpt-3.5-turbo"), column_names=["name",
↪ "age", "location"]
)

#  Define some unstructured data

sample_text = """
Alice, 28, lives in New
York.
Bob, 35 lives in California
Charlie is 40
Collin is 28 and lives in San Francisco
"""

#  Extract CSV rows using the engine

result = csv_engine.extract_text(sample_text)

for row in result:
    print(row.to_text())
```

```
name,age,location
Alice,28,New York
Bob,35,California
Charlie,40,
Collin,28,San Francisco
```

JSON

The JSON Extraction Engine extracts JSON-formatted values from unstructured text.

```
import json

from schema import Literal, Schema

```

(continues on next page)

(continued from previous page)

```python
from griptape.artifacts import ListArtifact
from griptape.chunkers import TextChunker
from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.engines import JsonExtractionEngine
from griptape.loaders import WebLoader

#  Define a schema for extraction

json_engine = JsonExtractionEngine(
    prompt_driver=OpenAiChatPromptDriver(model="gpt-4.1"),
    template_schema=Schema(
        {
            Literal("model", description="Name of an LLM model."): str,
            Literal("notes", description="Any notes of substance about the model."):␣
↪Schema([str]),
        }
    ).json_schema("ProductSchema"),
)

#  Load data from the web

web_data = WebLoader().load("https://en.wikipedia.org/wiki/Large_language_model")
chunks = TextChunker().chunk(web_data)

#  Extract data using the engine

result = json_engine.extract_artifacts(ListArtifact(chunks))

for artifact in result:
    print(json.dumps(artifact.value, indent=2))
```

```
{
  "model": "GPT-3.5",
  "notes": [
    "Part of OpenAIs GPT series.",
    "Used in ChatGPT and Microsoft Copilot."
  ]
}
{
  "model": "GPT-4",
  "notes": [
    "Part of OpenAIs GPT series.",
    "Praised for increased accuracy and multimodal capabilities.",
    "Architecture and number of parameters not revealed."
  ]
}
...Output truncated for brevity...
```

Summary Engine

Overview

Summary engines are used to summarize text and collections of *TextArtifact* s.

Prompt

Used to summarize texts with LLMs. You can set a custom *prompt_driver()* , *default_generate_system_template(context, artifacts)* , *default_generate_user_template(query, language)* , and *Chunkers* .

Use the summarize_artifacts(artifacts, *, rulesets=None)abstractmethod method to summarize a list of artifacts or summarize_text(text, *, rulesets=None) to summarize an arbitrary string.

```
import requests

from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.engines import PromptSummaryEngine
from griptape.loaders import PdfLoader

response = requests.get("https://arxiv.org/pdf/1706.03762.pdf")
engine = PromptSummaryEngine(
    prompt_driver=OpenAiChatPromptDriver(model="gpt-3.5-turbo"),
)

artifact = PdfLoader().parse(response.content)

engine.summarize_artifacts(artifact)
```

11.4 Events

11.4.1 Overview

You can configure the global *EventBus* with :ref:``EventListenerDriver.on_event` Return Value Behavior <*EventListenerDriver.on_event* Return Value Behavior>` s to listen for various framework events. See *Event Listener Drivers* for examples on forwarding events to external services.

11.4.2 Specific Event Types

You can listen to specific event types:

```
from griptape.events import (
    BaseEvent,
    EventBus,
    EventListener,
    FinishActionsSubtaskEvent,
    FinishPromptEvent,
    FinishTaskEvent,
    StartActionsSubtaskEvent,
    StartPromptEvent,
    StartTaskEvent,
)
from griptape.structures import Agent
```

(continues on next page)

(continued from previous page)

```
def on_event(event: BaseEvent) -> None:
    print(event.__class__)

EventBus.add_event_listeners(
    [
        EventListener(
            on_event,
            event_types=[
                StartTaskEvent,
                FinishTaskEvent,
                StartActionsSubtaskEvent,
                FinishActionsSubtaskEvent,
                StartPromptEvent,
                FinishPromptEvent,
            ],
        )
    ]
)

agent = Agent()

agent.run("tell me about griptape")
```

11.4.3 All Event Types

Or listen to all events:

```
from griptape.events import BaseEvent, EventBus, EventListener
from griptape.structures import Agent

def handler1(event: BaseEvent) -> None:
    print("Handler 1", event.__class__)

def handler2(event: BaseEvent) -> None:
    print("Handler 2", event.__class__)

EventBus.add_event_listeners(
    [
        EventListener(handler1),
        EventListener(handler2),
    ]
)

agent = Agent()

agent.run("tell me about griptape")
```

11.4.4 Stream Iterator

You can use `Structure.run_stream()` for streaming Events from the `Structure` in the form of an iterator.

> **Tip**
>
> Set *stream=True* on your *Prompt Drivers* in order to receive completion chunk events.

```
from griptape.events import BaseEvent
from griptape.structures import Agent

agent = Agent()

for event in agent.run_stream("Hi!", event_types=[BaseEvent]):  # All Events
    print(type(event))
```

11.4.5 Context Managers

You can also use :ref:`EventListenerDriver.on_event` Return Value Behavior <`EventListenerDriver.on_event` Return Value Behavior>` s as a Python Context Manager. The ``EventListener` will automatically be added and removed from the *EventBus* when entering and exiting the context.

```
from griptape.events import EventBus, EventListener, FinishStructureRunEvent,␣
↪StartPromptEvent
from griptape.structures import Agent

EventBus.add_event_listeners(
    [EventListener(lambda e: print(f"Out of context: {e.type}"), event_
↪types=[StartPromptEvent])]
)

agent = Agent(input="Hello!")

with EventListener(lambda e: print(f"In context: {e.type}"), event_
↪types=[FinishStructureRunEvent]):
    agent.run()

agent.run()
```

11.4.6 Streaming

You can use the *BaseChunkEvent* to stream the completion results from Prompt Drivers.

```
from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.events import BaseChunkEvent, EventBus, EventListener
from griptape.structures import Pipeline
from griptape.tasks import PromptTask
from griptape.tools import PromptSummaryTool, WebScraperTool

EventBus.add_event_listeners(
    [
```

(continues on next page)

(continued from previous page)

```
        EventListener(
            lambda e: print(str(e), end="", flush=True),
            event_types=[BaseChunkEvent],
        ),
    ]
)

pipeline = Pipeline()
pipeline.add_tasks(
    PromptTask(
        "Based on https://griptape.ai, tell me what griptape is.",
        prompt_driver=OpenAiChatPromptDriver(model="gpt-4.1", stream=True),
        tools=[WebScraperTool(off_prompt=True), PromptSummaryTool(off_prompt=False)],
    )
)

pipeline.run()
```

You can also use the *TextChunkEvent* and *ActionChunkEvent* to further differentiate the different types of chunks for more customized output.

```
from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.events import ActionChunkEvent, EventBus, EventListener, TextChunkEvent
from griptape.structures import Pipeline
from griptape.tasks import PromptTask
from griptape.tools import PromptSummaryTool, WebScraperTool

EventBus.add_event_listeners(
    [
        EventListener(
            lambda e: print(str(e), end="", flush=True),
            event_types=[TextChunkEvent],
        ),
        EventListener(
            lambda e: print(str(e), end="", flush=True),
            event_types=[ActionChunkEvent],
        ),
    ]
)

pipeline = Pipeline()
pipeline.add_tasks(
    PromptTask(
        "Based on https://griptape.ai, tell me what griptape is.",
        prompt_driver=OpenAiChatPromptDriver(model="gpt-4.1", stream=True),
        tools=[WebScraperTool(off_prompt=True), PromptSummaryTool(off_prompt=False)],
    )
)

pipeline.run()
```

If you want Gen AI Builder to handle the chunk events for you, use the *Stream Iterator* utility to automatically wrap *BaseChunkEvent* s in a Python iterator.

The `Stream` utility does not automatically enable streaming on the Drivers that produce `BaseChunkEvent` s. Make sure to enable streaming on the Drivers or else `Stream` will yield no iterations.

```
import logging

from griptape.configs import Defaults
from griptape.structures import Agent
from griptape.tools import PromptSummaryTool, WebScraperTool
from griptape.utils import Stream

#  Hide Griptapes usual output

logging.getLogger(Defaults.logging_config.logger_name).setLevel(logging.ERROR)

agent = Agent(
    input="Based on https://griptape.ai, tell me what griptape is.",
    tools=[
        PromptSummaryTool(off_prompt=True),
        WebScraperTool(off_prompt=False),
    ],
    stream=True,
)

for artifact in Stream(agent).run():
    print(artifact.value, end="", flush=True)
```

Sometimes, streaming can be too verbose. You can use `Stream.event_types` to only listen to specific event types. A good example is to remove the `ActionChunkEvent` from the stream if you don't need to see events related to Tool usage.

```
import logging

from griptape.configs import Defaults
from griptape.events import FinishPromptEvent, FinishStructureRunEvent, TextChunkEvent
from griptape.structures import Agent
from griptape.tools import PromptSummaryTool, WebScraperTool
from griptape.utils import Stream

#  Hide Griptapes usual output

logging.getLogger(Defaults.logging_config.logger_name).setLevel(logging.ERROR)

agent = Agent(
    input="Based on https://griptape.ai, tell me what griptape is.",
    tools=[
        PromptSummaryTool(off_prompt=True),
        WebScraperTool(off_prompt=False),
    ],
    stream=True,
)

#  Listen for the following event types

event_types = [TextChunkEvent, FinishPromptEvent, FinishStructureRunEvent]
```

(continues on next page)

(continued from previous page)

```
for artifact in Stream(agent, event_types=event_types).run():
    print(artifact.value, end="", flush=True)
```

11.4.7 Counting Tokens

To count tokens, you can use Event Listeners and the *TokenCounter* util:

```
- -8<-- "docs/griptape-framework/misc/src/events_5.py"
```

11.4.8 Inspecting Payloads

You can use the *StartPromptEvent* to inspect the Prompt Stack and final prompt string before it is sent to the LLM.

```
- -8<-- "docs/griptape-framework/misc/src/events_6.py"
```

```
...
Prompt Stack Messages:
system:
user: Write me a poem.
Final Prompt String:

User: Write me a poem.

Assistant:
...
```

11.4.9 `EventListenerDriver.on_event` Return Value Behavior

The value that gets returned from the :ref:`EventListener.on_event` is None <`EventListener.on_event` is None>` will determine what gets sent to the ``event_listener_driver`.

`EventListener.on_event` is None

By default, the `EventListener.on_event` function is `None`. Any events that the `EventListener` is listening for will get sent to the `event_listener_driver` as-is.

Return `BaseEvent` or `dict`

You can return a `BaseEvent` or `dict` object from `EventListener.on_event`, and it will get sent to the `event_listener_driver`.

Return `None`

You can return `None` in the on_event function to prevent the event from getting sent to the `event_listener_driver`.

```
- -8<-- "docs/griptape-framework/misc/src/events_no_publish.py"
```

Serialization

Overview

Many components in Gen AI Builder can be serialized and deserialized using the to_dict(*, types_overrides=None, serializable_overrides=None) and from_dict(data, *, types_overrides=None, serializable_overrides=None)classmethod methods. There are also *value_to_json(value)classmethod* and from_json(data, *, types_overrides=None, serializable_overrides=None)classmethod as a convenience.

Here is how we can serialize an `Agent` and then deserialize it back:

```
from griptape.structures import Agent

agent = Agent()
agent.run("My name is Collin")

agent_dict = agent.to_dict()

new_agent = Agent.from_dict(agent_dict)

new_agent.run("Whats my name?")
```

Serialization Overrides

All classes that implement the *SerializableMixin* can be serialized using the above methods. However, only fields marked with `metadata={"serializable": True}` will be included in the serialization process. If you need to add or remove fields in the serialization process, you can pass *serialization_overrides* to any of the serialization methods.

```
from rich.pretty import pprint

from griptape.structures import Agent

agent = Agent()
agent.run("My name is Collin")

agent_dict = agent.to_dict(serializable_overrides={"max_meta_memory_entries": True,
↪"id": False})

#  `max_meta_memory_entries` will be included, `id` will not

pprint(agent_dict)
```

Types Overrides

Due to some unfortunate internals of the Gen AI Builder's serialization process, you may occasionally run into a `NameError` when serializing. It will look something like this:

This is something we're actively working on fixing, but in the meantime, you can use the from_dict(data, *, types_overrides=None, serializable_overrides=None)classmethod parameter to pass in a dictionary of types that need to be overridden during serialization.

Here is an example of how you can use `types_overrides` :

```
from duckduckgo_search import DDGS
from rich.pretty import pprint
```

(continues on next page)

(continued from previous page)

```
from griptape.drivers.web_search import BaseWebSearchDriver
from griptape.drivers.web_search.duck_duck_go import DuckDuckGoWebSearchDriver
from griptape.tools import WebSearchTool

web_driver = DuckDuckGoWebSearchDriver()
web_tool = WebSearchTool(web_search_driver=web_driver)

web_tool_dict = web_tool.to_dict(
    serializable_overrides={"web_search_driver": True},
    types_overrides={"BaseWebSearchDriver": BaseWebSearchDriver, "DDGS": DDGS},
)

pprint(web_tool_dict)
```

Tokenizers

Overview

Tokenizers are used throughout Gen AI Builder to calculate the number of tokens in a piece of text. They are particularly useful for ensuring that the LLM token limits are not exceeded.

Tokenizers are a low level abstraction that you will rarely interact with directly.

Tokenizers

OpenAI

```
from griptape.tokenizers import OpenAiTokenizer

tokenizer = OpenAiTokenizer(model="gpt-4.1")

print(tokenizer.count_tokens("Hello world!"))
print(tokenizer.count_input_tokens_left("Hello world!"))
print(tokenizer.count_output_tokens_left("Hello world!"))
```

Cohere

```
import os

from cohere import Client

from griptape.tokenizers import CohereTokenizer

tokenizer = CohereTokenizer(model="command", client=Client(os.environ["COHERE_API_KEY
↪"]))

print(tokenizer.count_tokens("Hello world!"))
print(tokenizer.count_input_tokens_left("Hello world!"))
print(tokenizer.count_output_tokens_left("Hello world!"))
```

Anthropic

```
from griptape.tokenizers import AnthropicTokenizer

tokenizer = AnthropicTokenizer(model="claude-3-opus-20240229")

print(tokenizer.count_tokens("Hello world!"))
print(tokenizer.count_input_tokens_left("Hello world!"))
print(tokenizer.count_output_tokens_left("Hello world!"))
```

Google

```
import os

from griptape.tokenizers import GoogleTokenizer

tokenizer = GoogleTokenizer(model="gemini-2.0-flash", api_key=os.environ["GOOGLE_API_
↪KEY"])

print(tokenizer.count_tokens("Hello world!"))
print(tokenizer.count_input_tokens_left("Hello world!"))
print(tokenizer.count_output_tokens_left("Hello world!"))
```

Hugging Face

```
from griptape.tokenizers import HuggingFaceTokenizer

tokenizer = HuggingFaceTokenizer(
    model="sentence-transformers/all-MiniLM-L6-v2",
    max_output_tokens=512,
)

print(tokenizer.count_tokens("Hello world!"))
print(tokenizer.count_input_tokens_left("Hello world!"))
print(tokenizer.count_output_tokens_left("Hello world!"))
```

Amazon Bedrock

```
from griptape.tokenizers import AmazonBedrockTokenizer

tokenizer = AmazonBedrockTokenizer(model="amazon.titan-text-express-v1")

print(tokenizer.count_tokens("Hello world!"))
print(tokenizer.count_input_tokens_left("Hello world!"))
print(tokenizer.count_output_tokens_left("Hello world!"))
```

Grok

```
import os

```

(continues on next page)

(continued from previous page)

```
from griptape.tokenizers import GrokTokenizer

tokenizer = GrokTokenizer(
    model="grok-2-latest",
    api_key=os.environ["GROK_API_KEY"],
)

print(tokenizer.count_tokens("Hello world!"))
print(tokenizer.count_input_tokens_left("Hello world!"))
print(tokenizer.count_output_tokens_left("Hello world!"))
```

Simple

Not all LLM providers have a public tokenizer API. In this case, you can use the `SimpleTokenizer` to count tokens based on a simple heuristic.

```
from griptape.tokenizers import SimpleTokenizer

tokenizer = SimpleTokenizer(max_input_tokens=1024, max_output_tokens=1024, characters_
↪per_token=6)

print(tokenizer.count_tokens("Hello world!"))
print(tokenizer.count_input_tokens_left("Hello world!"))
print(tokenizer.count_output_tokens_left("Hello world!"))
```

11.5 SDK reference

11.5.1 artifacts

- `__all__ = ['ActionArtifact', 'AudioArtifact', 'BaseArtifact', 'BlobArtifact', 'UrlArtifact', 'BooleanArtifact', 'ErrorArtifact', 'GenericArtifact', 'ImageArtifact', 'ImageUrlArtifact', 'InfoArtifact', 'JsonArtifact', 'ListArtifact', 'ModelArtifact', 'TextArtifact']` module-attribute

Bases: *BaseArtifact* “)

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">ToolAction</code>	The Action to take. Currently only supports ToolAction.

Source Code in griptape/artifacts/action_artifact.py

```
@define()
class ActionArtifact(BaseArtifact):
    """Represents the LLM taking an action to use a Tool.

    Attributes:
        value: The Action to take. Currently only supports ToolAction.
    """

    value: ToolAction = field(metadata={"serializable": True})

    def to_text(self) -> str:
        return str(self.value)
```

- `value = field(metadata={'serializable': True})` class-attribute instance-attribute

to_text()

Source Code in griptape/artifacts/action_artifact.py

```
def to_text(self) -> str:
    return str(self.value)
```

AudioArtifact

Bases: *BlobArtifact* “)

Attributes

Name	Type	Description
<a href="./#griptape.artifacts.AudioArtifact.format" title="format = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.AudioArtifact.format)"><code class="language-python">	*str*	The audio format, e.g. "wav" or "mp3".

Source Code in griptape/artifacts/audio_artifact.py

```
@define
class AudioArtifact(BlobArtifact):
    """Stores audio data.
```

(continues on next page)

(continued from previous page)

```
    Attributes:
        format: The audio format, e.g. "wav" or "mp3".
    """

    format: str = field(kw_only=True, metadata={"serializable": True})

    def __attrs_post_init__(self) -> None:
        # Generating the name string requires attributes set by child classes.
        # This waits until all attributes are available before generating a name.
        if self.name == self.id:
            self.name = self.make_name()

    @property
    def mime_type(self) -> str:
        return f"audio/{self.format}"

    def to_bytes(self, **kwargs) -> bytes:
        return self.value

    def to_text(self) -> str:
        return f"Audio, format: {self.format}, size: {len(self.value)} bytes"

    def make_name(self) -> str:
        entropy = "".join(random.choices(string.ascii_lowercase + string.digits, k=4))
        fmt_time = time.strftime("%y%m%d%H%M%S", time.localtime())

        return f"audio_artifact_{fmt_time}_{entropy}.{self.format}"
```

- `format = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `mime_type` property

attrs_post_init()

Source Code in griptape/artifacts/audio_artifact.py

```
def __attrs_post_init__(self) -> None:
    # Generating the name string requires attributes set by child classes.
    # This waits until all attributes are available before generating a name.
    if self.name == self.id:
        self.name = self.make_name()
```

make_name()

Source Code in griptape/artifacts/audio_artifact.py

```
def make_name(self) -> str:
    entropy = "".join(random.choices(string.ascii_lowercase + string.digits, k=4))
    fmt_time = time.strftime("%y%m%d%H%M%S", time.localtime())

    return f"audio_artifact_{fmt_time}_{entropy}.{self.format}"
```

to_bytes(**kwargs)

Source Code in griptape/artifacts/audio_artifact.py

```
def to_bytes(self, **kwargs) -> bytes:
    return self.value
```

to_text()

Source Code in griptape/artifacts/audio_artifact.py

```
def to_text(self) -> str:
    return f"Audio, format: {self.format}, size: {len(self.value)} bytes"
```

BaseArtifact

Bases: *SerializableMixin* “) , ABC

Attributes

Name	Type	Description
<a href="./#griptape.artifacts.BaseArtifact.id" title="id = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.BaseArtifact.id)"><code class="language-python">	*str*	The unique identifier of the Artifact. Defaults to a random UUID.
<a href="./#griptape.artifacts.BaseArtifact.reference" title="reference = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.BaseArtifact.reference)"><code class="language-python">	<a href="../common#griptape.common.Reference" title="Reference (griptape.common.Reference)"><code class="language-python">Optional[Reference]</code>	The optional Reference to the Artifact.

continues on next page

Table 3 – continued from previous page

Name	Type	Description
<a href="./#griptape.artifacts.BaseArtifact.meta" title="meta = field(factory=dict, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.BaseArtifact.meta)"><code class="language-python">	*dict[str, Any]*	The metadata associated with the Artifact. Defaults to an empty dictionary.
<a href="./#griptape.artifacts.BaseArtifact.name" title="name = field(default=Factory(lambda self: self.id, takes_self=True), kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.BaseArtifact.name)"><code class="language-python">	*str*	The name of the Artifact. Defaults to the id.
<a href="./#griptape.artifacts.BaseArtifact.value" title="value = field() class-attribute instance-attribute (griptape.artifacts.BaseArtifact.value)"><code class="language-python">	*Any*	The value of the Artifact.
<a href="./#griptape.artifacts.BaseArtifact.encoding" title="encoding = field(default='utf-8', kw_only=True) class-attribute instance-attribute (griptape.artifacts.BaseArtifact.encoding)"><code class="language-python">	*str*	The encoding to use when encoding/decoding the value.

continues on next page

Table 3 – continued from previous page

Name	Type	Description
<a href="./#griptape.artifacts.BaseArtifact.encoding_error_handler" title="encoding_error_handler = field(default='strict', kw_only=True) class-attribute instance-attribute (griptape.artifacts.BaseArtifact.encoding_error_handler)"><code class="language-python">	*str*	The error on_event to use when encoding/decoding the value.

Source Code in griptape/artifacts/base_artifact.py

```
@define
class BaseArtifact(SerializableMixin, ABC):
    """Serves as the base class for all Artifacts.

    Artifacts are used to encapsulate data and enhance it with metadata.

    Attributes:
        id: The unique identifier of the Artifact. Defaults to a random UUID.
        reference: The optional Reference to the Artifact.
        meta: The metadata associated with the Artifact. Defaults to an empty␣
↪dictionary.
        name: The name of the Artifact. Defaults to the id.
        value: The value of the Artifact.
        encoding: The encoding to use when encoding/decoding the value.
        encoding_error_handler: The error on_event to use when encoding/decoding the␣
↪value.
    """

    id: str = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata=
↪{"serializable": True})
    reference: Optional[Reference] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    meta: dict[str, Any] = field(factory=dict, kw_only=True, metadata={"serializable
↪": True})
    name: str = field(
        default=Factory(lambda self: self.id, takes_self=True),
        kw_only=True,
        metadata={"serializable": True},
    )
    value: Any = field()
    encoding_error_handler: str = field(default="strict", kw_only=True)
    encoding: str = field(default="utf-8", kw_only=True)

    def __str__(self) -> str:
```

(continues on next page)

(continued from previous page)

```
        return self.to_text()

    def __bool__(self) -> bool:
        return bool(self.value)

    def __len__(self) -> int:
        return len(self.value)

    def to_bytes(self, **kwargs) -> bytes:
        return self.to_text().encode(encoding=self.encoding, errors=self.encoding_
↪error_handler)

    @abstractmethod
    def to_text(self) -> str: ...
```

- `encoding = field(default='utf-8', kw_only=True)` class-attribute instance-attribute
- `encoding_error_handler = field(default='strict', kw_only=True)` class-attribute instance-attribute
- `id = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `meta = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `name = field(default=Factory(lambda self: self.id, takes_self=True), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `reference = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `value = field()` class-attribute instance-attribute

bool()

Source Code in griptape/artifacts/base_artifact.py

```
def __bool__(self) -> bool:
    return bool(self.value)
```

len()

Source Code in griptape/artifacts/base_artifact.py

```
def __len__(self) -> int:
    return len(self.value)
```

str()

Source Code in griptape/artifacts/base_artifact.py

```
def __str__(self) -> str:
    return self.to_text()
```

to_bytes(**kwargs)

Source Code in griptape/artifacts/base_artifact.py

```
def to_bytes(self, **kwargs) -> bytes:
    return self.to_text().encode(encoding=self.encoding, errors=self.encoding_error_
↪handler)
```

to_text()abstractmethod

Source Code in griptape/artifacts/base_artifact.py

```
@abstractmethod
def to_text(self) -> str: ...
```

BlobArtifact

Bases: *BaseArtifact* “)

Attributes

Name	Type	Description
<a href="./#griptape.artifacts.BlobArtifact.value" title="value = field(converter=lambda value: value if isinstance(value, bytes) else str(value).encode(), metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.BlobArtifact.value)"><code class="language-python":	*bytes*	The binary data.

Source Code in griptape/artifacts/blob_artifact.py

```
@define
class BlobArtifact(BaseArtifact):
    """Stores arbitrary binary data.

    Attributes:
        value: The binary data.
    """

    value: bytes = field(
        converter=lambda value: value if isinstance(value, bytes) else str(value).
↪encode(),
        metadata={"serializable": True},
```

(continues on next page)

(continued from previous page)

```
    )

    @property
    def base64(self) -> str:
        return base64.b64encode(self.value).decode(self.encoding)

    @property
    def mime_type(self) -> str:
        return "application/octet-stream"

    def to_bytes(self, **kwargs) -> bytes:
        return self.value

    def to_text(self) -> str:
        return self.value.decode(encoding=self.encoding, errors=self.encoding_error_
↪handler)
```

- `base64` property
- `mime_type` property
- `value = field(converter=lambda value: value if isinstance(value, bytes) else str(value).encode(), metadata={'serializable': True})` class-attribute instance-attribute

to_bytes(**kwargs)

Source Code in griptape/artifacts/blob_artifact.py

```
def to_bytes(self, **kwargs) -> bytes:
    return self.value
```

to_text()

Source Code in griptape/artifacts/blob_artifact.py

```
def to_text(self) -> str:
    return self.value.decode(encoding=self.encoding, errors=self.encoding_error_
↪handler)
```

BooleanArtifact

Bases: *BaseArtifact* “)

Attributes

Name	Type	Description
<a href="./#griptape.artifacts.BooleanArtifact.value" title="value = field(converter=bool, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.BooleanArtifact.value)"><code class="language-python">	*bool*	The boolean value.

Source Code in griptape/artifacts/boolean_artifact.py

```python
@define
class BooleanArtifact(BaseArtifact):
    """Stores a boolean value.

    Attributes:
        value: The boolean value.
    """

    value: bool = field(converter=bool, metadata={"serializable": True})

    @classmethod
    def parse_bool(cls, value: Union[str, bool]) -> BooleanArtifact:
        """Convert a string literal or bool to a BooleanArtifact. The string must be either "true" or "false"."""
        if value is not None:
            if isinstance(value, str):
                if value.lower() == "true":
                    return BooleanArtifact(value=True)
                if value.lower() == "false":
                    return BooleanArtifact(value=False)
            elif isinstance(value, bool):
                return BooleanArtifact(value)
        raise ValueError(f"Cannot convert {value} to BooleanArtifact")

    def __add__(self, other: BaseArtifact) -> BooleanArtifact:
        raise ValueError("Cannot add BooleanArtifact with other artifacts")

    def __eq__(self, value: object) -> bool:
        return self.value == value

    def to_text(self) -> str:
        return str(self.value).lower()
```

- `value = field(converter=bool, metadata={'serializable': True})` class-attribute instance-attribute

add(other)

Source Code in griptape/artifacts/boolean_artifact.py

```
def __add__(self, other: BaseArtifact) -> BooleanArtifact:
    raise ValueError("Cannot add BooleanArtifact with other artifacts")
```

eq(value)

Source Code in griptape/artifacts/boolean_artifact.py

```
def __eq__(self, value: object) -> bool:
    return self.value == value
```

parse_bool(value)classmethod

Source Code in griptape/artifacts/boolean_artifact.py

```
@classmethod
def parse_bool(cls, value: Union[str, bool]) -> BooleanArtifact:
    """Convert a string literal or bool to a BooleanArtifact. The string must be␣
↪either "true" or "false"."""
    if value is not None:
        if isinstance(value, str):
            if value.lower() == "true":
                return BooleanArtifact(value=True)
            if value.lower() == "false":
                return BooleanArtifact(value=False)
        elif isinstance(value, bool):
            return BooleanArtifact(value)
    raise ValueError(f"Cannot convert {value} to BooleanArtifact")
```

to_text()

Source Code in griptape/artifacts/boolean_artifact.py

```
def to_text(self) -> str:
    return str(self.value).lower()
```

ErrorArtifact

Bases: *BaseArtifact* “)

Attributes

Name	Type	Description
<a href="./#griptape.artifacts.ErrorArtifact.value" title="value = field(converter=str, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.ErrorArtifact.value)"><code class="language-python">	*str*	The error message.
<a href="./#griptape.artifacts.ErrorArtifact.exception" title="exception = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.artifacts.ErrorArtifact.exception)"><code class="language-python">	*Optional[Exception]*	The exception that caused the error. Defaults to None.

Source Code in griptape/artifacts/error_artifact.py

```
@define
class ErrorArtifact(BaseArtifact):
    """Represents an error that may want to be conveyed to the LLM.

    Attributes:
        value: The error message.
        exception: The exception that caused the error. Defaults to None.
    """

    value: str = field(converter=str, metadata={"serializable": True})
    exception: Optional[Exception] = field(default=None, kw_only=True, metadata={
↪"serializable": False})

    def to_text(self) -> str:
        return self.value
```

- `exception = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `value = field(converter=str, metadata={'serializable': True})` class-attribute instance-attribute

to_text()

Source Code in griptape/artifacts/error_artifact.py

```
def to_text(self) -> str:
    return self.value
```

GenericArtifact

Bases: *BaseArtifact* “) , *Generic[T*

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">T</code>	The value of the Artifact.

Source Code in griptape/artifacts/generic_artifact.py

```
@define
class GenericArtifact(BaseArtifact, Generic[T]):
    """Serves as an escape hatch for artifacts that dont fit into any other category.

    Attributes:
        value: The value of the Artifact.
    """

    value: T = field(metadata={"serializable": True})

    def to_text(self) -> str:
        return str(self.value)
```

- `value = field(metadata={'serializable': True})` class-attribute instance-attribute

to_text()

Source Code in griptape/artifacts/generic_artifact.py

```
def to_text(self) -> str:
    return str(self.value)
```

ImageArtifact

Bases: *BlobArtifact* “)

Attributes

Name	Type	Description
<a href="./#griptape.artifacts.ImageArtifact.format" title="format = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.ImageArtifact.format)"><code class="language-python">	*str*	The format of the image data. Used when building the MIME type.
<a href="./#griptape.artifacts.ImageArtifact.width" title="width = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.ImageArtifact.width)"><code class="language-python">	*int*	The width of the image.
<a href="./#griptape.artifacts.ImageArtifact.height" title="height = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.ImageArtifact.height)"><code class="language-python">	*int*	The height of the image

Source Code in griptape/artifacts/image_artifact.py

```
@define
class ImageArtifact(BlobArtifact):
    """Stores image data.
```

(continues on next page)

(continued from previous page)

```
    Attributes:
        format: The format of the image data. Used when building the MIME type.
        width: The width of the image.
        height: The height of the image
    """

    format: str = field(kw_only=True, metadata={"serializable": True})
    width: int = field(kw_only=True, metadata={"serializable": True})
    height: int = field(kw_only=True, metadata={"serializable": True})

    def __attrs_post_init__(self) -> None:
        # Generating the name string requires attributes set by child classes.
        # This waits until all attributes are available before generating a name.
        if self.name == self.id:
            self.name = self.make_name()

    @property
    def mime_type(self) -> str:
        return f"image/{self.format}"

    def to_bytes(self, **kwargs) -> bytes:
        return self.value

    def to_text(self) -> str:
        return f"Image, format: {self.format}, size: {len(self.value)} bytes"

    def make_name(self) -> str:
        entropy = "".join(random.choices(string.ascii_lowercase + string.digits, k=4))
        fmt_time = time.strftime("%y%m%d%H%M%S", time.localtime())

        return f"image_artifact_{fmt_time}_{entropy}.{self.format}"
```

- `format = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `height = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `mime_type` property
- `width = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/artifacts/image_artifact.py

```
def __attrs_post_init__(self) -> None:
    # Generating the name string requires attributes set by child classes.
    # This waits until all attributes are available before generating a name.
    if self.name == self.id:
        self.name = self.make_name()
```

make_name()

Source Code in griptape/artifacts/image_artifact.py

```
def make_name(self) -> str:
    entropy = "".join(random.choices(string.ascii_lowercase + string.digits, k=4))
    fmt_time = time.strftime("%y%m%d%H%M%S", time.localtime())

    return f"image_artifact_{fmt_time}_{entropy}.{self.format}"
```

to_bytes(**kwargs)

Source Code in griptape/artifacts/image_artifact.py

```
def to_bytes(self, **kwargs) -> bytes:
    return self.value
```

to_text()

Source Code in griptape/artifacts/image_artifact.py

```
def to_text(self) -> str:
    return f"Image, format: {self.format}, size: {len(self.value)} bytes"
```

ImageUrlArtifact

Bases: *ImageUrlArtifact* “)

Attributes

Name	Type	Description
<a href="./#griptape.artifacts.UrlArtifact.value" title="value = field(metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.ImageUrlArtifact.value)"><code class="language-python">	*str*	The url.

Source Code in griptape/artifacts/image_url_artifact.py

```
class ImageUrlArtifact(UrlArtifact):
    """Stores a url to an image.

    Attributes:
        value: The url.
    """
```

InfoArtifact

Bases: *BaseArtifact* “)

Attributes

Name	Type	Description
<a href="./#griptape.artifacts.InfoArtifact.value" title="value = field(converter=str, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.InfoArtifact.value)"><code class="language-python">	*str*	The info to convey.

Source Code in griptape/artifacts/info_artifact.py

```
@define
class InfoArtifact(BaseArtifact):
    """Represents helpful info that can be conveyed to the LLM.

    For example, "No results found" or "Please try again.".

    Attributes:
        value: The info to convey.
    """

    value: str = field(converter=str, metadata={"serializable": True})

    def to_text(self) -> str:
        return self.value
```

- `value = field(converter=str, metadata={'serializable': True})` class-attribute instance-attribute

to_text()

Source Code in griptape/artifacts/info_artifact.py

```
def to_text(self) -> str:
    return self.value
```

JsonArtifact

Bases: *BaseArtifact* “)

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">Json</code>	The JSON data. Values will automatically be converted to a JSON-compatible format.

Source Code in griptape/artifacts/json_artifact.py

```
@define
class JsonArtifact(BaseArtifact):
    """Stores JSON data.

    Attributes:
        value: The JSON data. Values will automatically be converted to a JSON-
↪compatible format.
    """

    value: Json = field(converter=lambda value: JsonArtifact.value_to_json(value),␣
↪metadata={"serializable": True})

    @classmethod
    def value_to_json(cls, value: Any) -> Json:
        if isinstance(value, str):
            return json.loads(value)
        return json.loads(json.dumps(value))

    def to_text(self) -> str:
        return json.dumps(self.value)
```

- `value = field(converter=lambda value: JsonArtifact.value_to_json(value), metadata={'serializable': True})` class-attribute instance-attribute

to_text()

Source Code in griptape/artifacts/json_artifact.py

```
def to_text(self) -> str:
    return json.dumps(self.value)
```

value_to_json(value)classmethod

Source Code in griptape/artifacts/json_artifact.py

```
@classmethod
def value_to_json(cls, value: Any) -> Json:
    if isinstance(value, str):
        return json.loads(value)
    return json.loads(json.dumps(value))
```

ListArtifact

Bases: *BaseArtifact* “) , *Generic[T_co*

Source Code in griptape/artifacts/list_artifact.py

```
@define
class ListArtifact(BaseArtifact, Generic[T_co]):
    value: Sequence[T_co] = field(factory=list, metadata={"serializable": True})
    item_separator: str = field(default="\n\n", kw_only=True, metadata={"serializable
↪": True})
    validate_uniform_types: bool = field(default=False, kw_only=True, metadata={
↪"serializable": True})

    @value.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_value(self, _: Attribute, value: list[T_co]) -> None:
        if self.validate_uniform_types and len(value) > 0:
            first_type = type(value[0])

            if not all(isinstance(v, first_type) for v in value):
                raise ValueError("list elements in value are not the same type")

    @property
    def child_type(self) -> Optional[type]:
        if self.value:
            return type(self.value[0])
        return None

    def __getitem__(self, key: int) -> T_co:
        return self.value[key]

    def __bool__(self) -> bool:
        return len(self) > 0

    def __add__(self, other: BaseArtifact) -> ListArtifact[T_co]:
        return ListArtifact(self.value + other.value)

    def __iter__(self) -> Iterator[T_co]:
        return iter(self.value)

    def to_text(self) -> str:
        return self.item_separator.join([v.to_text() for v in self.value])

    def is_type(self, target_type: type) -> bool:
```

(continues on next page)

(continued from previous page)

```
        if self.value:
            return isinstance(self.value[0], target_type)
        return False

    def has_items(self) -> bool:
        return len(self) > 0
```

- `child_type` property
- `item_separator = field(default='\n\n', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `validate_uniform_types = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `value = field(factory=list, metadata={'serializable': True})` class-attribute instance-attribute

add(other)

Source Code in griptape/artifacts/list_artifact.py

```
def __add__(self, other: BaseArtifact) -> ListArtifact[T_co]:
    return ListArtifact(self.value + other.value)
```

bool()

Source Code in griptape/artifacts/list_artifact.py

```
def __bool__(self) -> bool:
    return len(self) > 0
```

getitem(key)

Source Code in griptape/artifacts/list_artifact.py

```
def __getitem__(self, key: int) -> T_co:
    return self.value[key]
```

iter()

Source Code in griptape/artifacts/list_artifact.py

```
def __iter__(self) -> Iterator[T_co]:
    return iter(self.value)
```

has_items()

Source Code in griptape/artifacts/list_artifact.py

```
def has_items(self) -> bool:
    return len(self) > 0
```

is_type(target_type)

Source Code in griptape/artifacts/list_artifact.py

```
def is_type(self, target_type: type) -> bool:
    if self.value:
        return isinstance(self.value[0], target_type)
    return False
```

to_text()

Source Code in griptape/artifacts/list_artifact.py

```
def to_text(self) -> str:
    return self.item_separator.join([v.to_text() for v in self.value])
```

validate*value(*, value)

Source Code in griptape/artifacts/list_artifact.py

```
@value.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_value(self, _: Attribute, value: list[T_co]) -> None:
    if self.validate_uniform_types and len(value) > 0:
        first_type = type(value[0])

        if not all(isinstance(v, first_type) for v in value):
            raise ValueError("list elements in value are not the same type")
```

ModelArtifact

Bases: *GenericArtifact[BaseModel* “)

Attributes

Name	Type	Description
<a href="./#griptape.artifacts.ModelArtifact.value" title="value = field(metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.ModelArtifact.value)"><code class="language-python">	*BaseModel*	The pydantic model to store.

Source Code in griptape/artifacts/model_artifact.py

```
@define
class ModelArtifact(GenericArtifact[BaseModel]):
    """Stores Pydantic models as Artifacts.
```

(continues on next page)

(continued from previous page)

```
    Required since Pydantic models require a custom serialization method.

    Attributes:
        value: The pydantic model to store.
    """

    # We must explicitly define the type rather than rely on the parent T since
    # generic type information is lost at runtime.
    value: BaseModel = field(metadata={"serializable": True})

    def to_text(self) -> str:
        return self.value.model_dump_json()
```

- `value = field(metadata={'serializable': True})` class-attribute instance-attribute

to_text()

Source Code in griptape/artifacts/model_artifact.py

```
def to_text(self) -> str:
    return self.value.model_dump_json()
```

TextArtifact

Bases: *BaseArtifact* “)

Source Code in griptape/artifacts/text_artifact.py

```
@define
class TextArtifact(BaseArtifact):
    value: str = field(converter=str, metadata={"serializable": True})
    embedding: Optional[list[float]] = field(default=None, kw_only=True)

    def __add__(self, other: BaseArtifact) -> TextArtifact:
        return TextArtifact(self.value + other.value)

    def __bool__(self) -> bool:
        return bool(self.value.strip())

    def to_text(self) -> str:
        return self.value

    def generate_embedding(self, driver: BaseEmbeddingDriver) -> list[float]:
        embedding = driver.embed(str(self.value))

        if self.embedding is None:
            self.embedding = []
        self.embedding.clear()
        self.embedding.extend(embedding)

        return self.embedding
```

(continues on next page)

(continued from previous page)

```
    def token_count(self, tokenizer: BaseTokenizer) -> int:
        return tokenizer.count_tokens(str(self.value))
```

- `embedding = field(default=None, kw_only=True)` class-attribute instance-attribute
- `value = field(converter=str, metadata={'serializable': True})` class-attribute instance-attribute

add(other)

Source Code in griptape/artifacts/text_artifact.py

```
def __add__(self, other: BaseArtifact) -> TextArtifact:
    return TextArtifact(self.value + other.value)
```

bool()

Source Code in griptape/artifacts/text_artifact.py

```
def __bool__(self) -> bool:
    return bool(self.value.strip())
```

generate_embedding(driver)

Source Code in griptape/artifacts/text_artifact.py

```
def generate_embedding(self, driver: BaseEmbeddingDriver) -> list[float]:
    embedding = driver.embed(str(self.value))

    if self.embedding is None:
        self.embedding = []
    self.embedding.clear()
    self.embedding.extend(embedding)

    return self.embedding
```

to_text()

Source Code in griptape/artifacts/text_artifact.py

```
def to_text(self) -> str:
    return self.value
```

token_count(tokenizer)

Source Code in griptape/artifacts/text_artifact.py

```
def token_count(self, tokenizer: BaseTokenizer) -> int:
    return tokenizer.count_tokens(str(self.value))
```

UrlArtifact

Bases: *BaseArtifact* “)

Attributes

Name	Type	Description
<a href="./#griptape.artifacts.UrlArtifact.value" title="value = field(metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.UrlArtifact.value)"><code class="language-python">	*str*	The url.

Source Code in griptape/artifacts/url_artifact.py

```python
@define
class UrlArtifact(BaseArtifact):
    """Stores a url.

    Attributes:
        value: The url.
    """

    value: str = field(metadata={"serializable": True})

    def to_bytes(self, *, headers: dict | None = None, **kwargs: dict) -> bytes:
        """Fetches the content of the URL and returns it as bytes.

        Args:
            headers: Optional headers to include in the request.
            **kwargs: Additional keyword arguments, not used.

        Returns:
            bytes: The content of the URL as bytes.

        """
        response = requests.get(self.value, headers=headers)
        response.raise_for_status()

        return response.content

    def to_text(self) -> str:
        """Returns the URL as is."""
        return self.value
```

- `value = field(metadata={'serializable': True})` class-attribute instance-attribute

to_bytes(*, headers=None, **kwargs)

Fetches the content of the URL and returns it as bytes.

Parameters

Name	Type	Description	Default
headers	*dict I None*	Optional headers to include in the request. 	*None*
***kwargs*	*dict*	Additional keyword arguments, not used. 	*{}*

Returns

Name	Type	Description
bytes	*bytes*	The content of the URL as bytes.

Source Code in griptape/artifacts/url_artifact.py

```
def to_bytes(self, *, headers: dict | None = None, **kwargs: dict) -> bytes:
    """Fetches the content of the URL and returns it as bytes.

    Args:
        headers: Optional headers to include in the request.
        **kwargs: Additional keyword arguments, not used.

    Returns:
        bytes: The content of the URL as bytes.

    """
    response = requests.get(self.value, headers=headers)
    response.raise_for_status()

    return response.content
```

to_text()

Source Code in griptape/artifacts/url_artifact.py

```
def to_text(self) -> str:
    """Returns the URL as is."""
    return self.value
```

action_artifact

Bases: *BaseArtifact* “)

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">ToolAction</code>	The Action to take. Currently only supports ToolAction.

Source Code in griptape/artifacts/action_artifact.py

```
@define()
class ActionArtifact(BaseArtifact):
    """Represents the LLM taking an action to use a Tool.

    Attributes:
        value: The Action to take. Currently only supports ToolAction.
    """

    value: ToolAction = field(metadata={"serializable": True})

    def to_text(self) -> str:
        return str(self.value)
```

- `value = field(metadata={'serializable': True})` class-attribute instance-attribute

to_text()

Source Code in griptape/artifacts/action_artifact.py

```
def to_text(self) -> str:
    return str(self.value)
```

audio_artifact

Bases: *BlobArtifact* “)

Attributes

Name	Type	Description
<a href="audio_artifact.mdx#griptape.artifacts.audio_artifact.AudioArtifact.format" title="format = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.audio_artifact.AudioArtifact.format)"><code class="language-python">	*str*	The audio format, e.g. "wav" or "mp3".

Source Code in griptape/artifacts/audio_artifact.py

```python
@define
class AudioArtifact(BlobArtifact):
    """Stores audio data.

    Attributes:
        format: The audio format, e.g. "wav" or "mp3".
    """

    format: str = field(kw_only=True, metadata={"serializable": True})

    def __attrs_post_init__(self) -> None:
        # Generating the name string requires attributes set by child classes.
        # This waits until all attributes are available before generating a name.
        if self.name == self.id:
            self.name = self.make_name()

    @property
    def mime_type(self) -> str:
        return f"audio/{self.format}"

    def to_bytes(self, **kwargs) -> bytes:
        return self.value

    def to_text(self) -> str:
        return f"Audio, format: {self.format}, size: {len(self.value)} bytes"

    def make_name(self) -> str:
        entropy = "".join(random.choices(string.ascii_lowercase + string.digits, k=4))
        fmt_time = time.strftime("%y%m%d%H%M%S", time.localtime())

        return f"audio_artifact_{fmt_time}_{entropy}.{self.format}"
```

- `format = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `mime_type` property

attrs_post_init()

Source Code in griptape/artifacts/audio_artifact.py

```
def __attrs_post_init__(self) -> None:
    # Generating the name string requires attributes set by child classes.
    # This waits until all attributes are available before generating a name.
    if self.name == self.id:
        self.name = self.make_name()
```

make_name()

Source Code in griptape/artifacts/audio_artifact.py

```
def make_name(self) -> str:
    entropy = "".join(random.choices(string.ascii_lowercase + string.digits, k=4))
    fmt_time = time.strftime("%y%m%d%H%M%S", time.localtime())

    return f"audio_artifact_{fmt_time}_{entropy}.{self.format}"
```

to_bytes(**kwargs)

Source Code in griptape/artifacts/audio_artifact.py

```
def to_bytes(self, **kwargs) -> bytes:
    return self.value
```

to_text()

Source Code in griptape/artifacts/audio_artifact.py

```
def to_text(self) -> str:
    return f"Audio, format: {self.format}, size: {len(self.value)} bytes"
```

base_artifact

Bases: *SerializableMixin* “) , ABC

Attributes

Name	Type	Description
<a href="base_artifact.mdx#griptape.artifacts.base_artifact.BaseArtifact.id" title="id = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.base_artifact.BaseArtifact.id)"><code class="language-python">	*str*	The unique identifier of the Artifact. Defaults to a random UUID.
<a href="base_artifact.mdx#griptape.artifacts.base_artifact.BaseArtifact.reference" title="reference = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.base_artifact.BaseArtifact.reference)"><code class="language-python">	<a href="../common#griptape.common.Reference" title="Reference (griptape.common.Reference)"><code class="language-python">Optional[Reference]</code>	The optional Reference to the Artifact.
<a href="base_artifact.mdx#griptape.artifacts.base_artifact.BaseArtifact.meta" title="meta = field(factory=dict, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.base_artifact.BaseArtifact.meta)"><code class="language-python">	*dict[str, Any]*	The metadata associated with the Artifact. Defaults to an empty dictionary.

continues on next page

Table 18 – continued from previous page

Name	Type	Description
<a href="base_artifact.mdx#griptape.artifacts.base_artifact.BaseArtifact.name" title="name = field(default=Factory(lambda self: self.id, takes_self=True), kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.base_artifact.BaseArtifact.name)"><code class="language-python">	*str*	The name of the Artifact. Defaults to the id.
<a href="base_artifact.mdx#griptape.artifacts.base_artifact.BaseArtifact.value" title="value = field() class-attribute instance-attribute (griptape.artifacts.base_artifact.BaseArtifact.value)"><code class="language-python">	*Any*	The value of the Artifact.
<a href="base_artifact.mdx#griptape.artifacts.base_artifact.BaseArtifact.encoding" title="encoding = field(default='utf-8', kw_only=True) class-attribute instance-attribute (griptape.artifacts.base_artifact.BaseArtifact.encoding)"><code class="language-python">	*str*	The encoding to use when encoding/decoding the value.

continues on next page

Table 18 – continued from previous page

Name	Type	Description
<a href="base_artifact.mdx#griptape.artifacts.base_artifact.BaseArtifact.encoding_error_handler" title="encoding_error_handler = field(default='strict', kw_only=True) class-attribute instance-attribute (griptape.artifacts.base_artifact.BaseArtifact.encoding_error_handler)"><code class="language-python">	*str*	The error on_event to use when encoding/decoding the value.

Source Code in griptape/artifacts/base_artifact.py

```
@define
class BaseArtifact(SerializableMixin, ABC):
    """Serves as the base class for all Artifacts.

    Artifacts are used to encapsulate data and enhance it with metadata.

    Attributes:
        id: The unique identifier of the Artifact. Defaults to a random UUID.
        reference: The optional Reference to the Artifact.
        meta: The metadata associated with the Artifact. Defaults to an empty␣
↪dictionary.
        name: The name of the Artifact. Defaults to the id.
        value: The value of the Artifact.
        encoding: The encoding to use when encoding/decoding the value.
        encoding_error_handler: The error on_event to use when encoding/decoding the␣
↪value.
    """

    id: str = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata=
↪{"serializable": True})
    reference: Optional[Reference] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    meta: dict[str, Any] = field(factory=dict, kw_only=True, metadata={"serializable
↪": True})
    name: str = field(
        default=Factory(lambda self: self.id, takes_self=True),
        kw_only=True,
        metadata={"serializable": True},
    )
    value: Any = field()
    encoding_error_handler: str = field(default="strict", kw_only=True)
```

(continues on next page)

(continued from previous page)

```
    encoding: str = field(default="utf-8", kw_only=True)

    def __str__(self) -> str:
        return self.to_text()

    def __bool__(self) -> bool:
        return bool(self.value)

    def __len__(self) -> int:
        return len(self.value)

    def to_bytes(self, **kwargs) -> bytes:
        return self.to_text().encode(encoding=self.encoding, errors=self.encoding_
↪error_handler)

    @abstractmethod
    def to_text(self) -> str: ...
```

- `encoding = field(default='utf-8', kw_only=True)` class-attribute instance-attribute
- `encoding_error_handler = field(default='strict', kw_only=True)` class-attribute instance-attribute
- `id = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `meta = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `name = field(default=Factory(lambda self: self.id, takes_self=True), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `reference = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `value = field()` class-attribute instance-attribute

bool()

Source Code in griptape/artifacts/base_artifact.py

```
def __bool__(self) -> bool:
    return bool(self.value)
```

len()

Source Code in griptape/artifacts/base_artifact.py

```
def __len__(self) -> int:
    return len(self.value)
```

str()

Source Code in griptape/artifacts/base_artifact.py

```
def __str__(self) -> str:
    return self.to_text()
```

to_bytes(**kwargs)

Source Code in griptape/artifacts/base_artifact.py

```
def to_bytes(self, **kwargs) -> bytes:
    return self.to_text().encode(encoding=self.encoding, errors=self.encoding_error_
↪handler)
```

to_text()abstractmethod

Source Code in griptape/artifacts/base_artifact.py

```
@abstractmethod
def to_text(self) -> str: ...
```

blob_artifact

Bases: *BaseArtifact* “)

Attributes

Name	Type	Description
<a href="blob_artifact.mdx#griptape.artifacts.blob_artifact.BlobArtifact.value" title="value = field(converter=lambda value: value if isinstance(value, bytes) else str(value).encode(), metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.blob_artifact.BlobArtifact.value)"><code class="language-python">	*bytes*	The binary data.

Source Code in griptape/artifacts/blob_artifact.py

```
@define
class BlobArtifact(BaseArtifact):
```

(continues on next page)

(continued from previous page)

```
    """Stores arbitrary binary data.

    Attributes:
        value: The binary data.
    """

    value: bytes = field(
        converter=lambda value: value if isinstance(value, bytes) else str(value).
↪encode(),
        metadata={"serializable": True},
    )

    @property
    def base64(self) -> str:
        return base64.b64encode(self.value).decode(self.encoding)

    @property
    def mime_type(self) -> str:
        return "application/octet-stream"

    def to_bytes(self, **kwargs) -> bytes:
        return self.value

    def to_text(self) -> str:
        return self.value.decode(encoding=self.encoding, errors=self.encoding_error_
↪handler)
```

- `base64` property
- `mime_type` property
- `value = field(converter=lambda value: value if isinstance(value, bytes) else str(value).encode(), metadata={'serializable': True})` class-attribute instance-attribute

to_bytes(**kwargs)

Source Code in griptape/artifacts/blob_artifact.py

```
def to_bytes(self, **kwargs) -> bytes:
    return self.value
```

to_text()

Source Code in griptape/artifacts/blob_artifact.py

```
def to_text(self) -> str:
    return self.value.decode(encoding=self.encoding, errors=self.encoding_error_
↪handler)
```

boolean_artifact

Bases: *BaseArtifact* “)

Attributes

Name	Type	Description
<a href="boolean_artifact.mdx#griptape.artifacts.boolean_artifact.BooleanArtifact.value" title="value = field(converter=bool, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.boolean_artifact.BooleanArtifact.value)"><code class="language-python">	*bool*	The boolean value.

Source Code in griptape/artifacts/boolean_artifact.py

```
@define
class BooleanArtifact(BaseArtifact):
    """Stores a boolean value.

    Attributes:
        value: The boolean value.
    """

    value: bool = field(converter=bool, metadata={"serializable": True})

    @classmethod
    def parse_bool(cls, value: Union[str, bool]) -> BooleanArtifact:
        """Convert a string literal or bool to a BooleanArtifact. The string must be␣
↪either "true" or "false"."""
        if value is not None:
            if isinstance(value, str):
                if value.lower() == "true":
                    return BooleanArtifact(value=True)
                if value.lower() == "false":
                    return BooleanArtifact(value=False)
            elif isinstance(value, bool):
                return BooleanArtifact(value)
        raise ValueError(f"Cannot convert {value} to BooleanArtifact")

    def __add__(self, other: BaseArtifact) -> BooleanArtifact:
        raise ValueError("Cannot add BooleanArtifact with other artifacts")
```

(continues on next page)

(continued from previous page)

```
    def __eq__(self, value: object) -> bool:
        return self.value == value

    def to_text(self) -> str:
        return str(self.value).lower()
```

- `value = field(converter=bool, metadata={'serializable': True})` class-attribute instance-attribute

add(other)

Source Code in griptape/artifacts/boolean_artifact.py

```
def __add__(self, other: BaseArtifact) -> BooleanArtifact:
    raise ValueError("Cannot add BooleanArtifact with other artifacts")
```

eq(value)

Source Code in griptape/artifacts/boolean_artifact.py

```
def __eq__(self, value: object) -> bool:
    return self.value == value
```

parse_bool(value)classmethod

Source Code in griptape/artifacts/boolean_artifact.py

```
@classmethod
def parse_bool(cls, value: Union[str, bool]) -> BooleanArtifact:
    """Convert a string literal or bool to a BooleanArtifact. The string must be␣
↪either "true" or "false"."""
    if value is not None:
        if isinstance(value, str):
            if value.lower() == "true":
                return BooleanArtifact(value=True)
            if value.lower() == "false":
                return BooleanArtifact(value=False)
        elif isinstance(value, bool):
            return BooleanArtifact(value)
    raise ValueError(f"Cannot convert {value} to BooleanArtifact")
```

to_text()

Source Code in griptape/artifacts/boolean_artifact.py

```
def to_text(self) -> str:
    return str(self.value).lower()
```

error_artifact

Bases: *BaseArtifact* “)

Attributes

Name	Type	Description
<a href="error_artifact.mdx#griptape.artifacts.error_artifact.ErrorArtifact.value" title="value = field(converter=str, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.error_artifact.ErrorArtifact.value)"><code class="language-python">	*str*	The error message.
<a href="error_artifact.mdx#griptape.artifacts.error_artifact.ErrorArtifact.exception" title="exception = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.artifacts.error_artifact.ErrorArtifact.exception)"><code class="language-python">	*Optional[Exception]*	The exception that caused the error. Defaults to None.

Source Code in griptape/artifacts/error_artifact.py

```
@define
class ErrorArtifact(BaseArtifact):
    """Represents an error that may want to be conveyed to the LLM.

    Attributes:
        value: The error message.
        exception: The exception that caused the error. Defaults to None.
    """

    value: str = field(converter=str, metadata={"serializable": True})
    exception: Optional[Exception] = field(default=None, kw_only=True, metadata={
↪"serializable": False})

    def to_text(self) -> str:
        return self.value
```

- `exception = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `value = field(converter=str, metadata={'serializable': True})` class-attribute instance-attribute

to_text()

Source Code in griptape/artifacts/error_artifact.py

```
def to_text(self) -> str:
    return self.value
```

generic_artifact

- `T = TypeVar('T')` module-attribute

Bases: *BaseArtifact* “) , *Generic[T*

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">T</code>	The value of the Artifact.

Source Code in griptape/artifacts/generic_artifact.py

```
@define
class GenericArtifact(BaseArtifact, Generic[T]):
    """Serves as an escape hatch for artifacts that dont fit into any other category.

    Attributes:
        value: The value of the Artifact.
    """

    value: T = field(metadata={"serializable": True})

    def to_text(self) -> str:
        return str(self.value)
```

- `value = field(metadata={'serializable': True})` class-attribute instance-attribute

to_text()

Source Code in griptape/artifacts/generic_artifact.py

```
def to_text(self) -> str:
    return str(self.value)
```

image_artifact

Bases: *BlobArtifact* “)

Attributes

Name	Type	Description
<a href="image_artifact.mdx#griptape.artifacts.image_artifact.ImageArtifact.format" title="format = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.image_artifact.ImageArtifact.format)"><code class="language-python">	*str*	The format of the image data. Used when building the MIME type.
<a href="image_artifact.mdx#griptape.artifacts.image_artifact.ImageArtifact.width" title="width = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.image_artifact.ImageArtifact.width)"><code class="language-python">	*int*	The width of the image.

continues on next page

Table 23 – continued from previous page

Name	Type	Description
<a href="image_artifact.mdx#griptape.artifacts.image_artifact.ImageArtifact.height" title="height = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.image_artifact.ImageArtifact.height)"><code class="language-python">	*int*	The height of the image

Source Code in griptape/artifacts/image_artifact.py

```python
@define
class ImageArtifact(BlobArtifact):
    """Stores image data.

    Attributes:
        format: The format of the image data. Used when building the MIME type.
        width: The width of the image.
        height: The height of the image
    """

    format: str = field(kw_only=True, metadata={"serializable": True})
    width: int = field(kw_only=True, metadata={"serializable": True})
    height: int = field(kw_only=True, metadata={"serializable": True})

    def __attrs_post_init__(self) -> None:
        # Generating the name string requires attributes set by child classes.
        # This waits until all attributes are available before generating a name.
        if self.name == self.id:
            self.name = self.make_name()

    @property
    def mime_type(self) -> str:
        return f"image/{self.format}"

    def to_bytes(self, **kwargs) -> bytes:
        return self.value

    def to_text(self) -> str:
        return f"Image, format: {self.format}, size: {len(self.value)} bytes"

    def make_name(self) -> str:
        entropy = "".join(random.choices(string.ascii_lowercase + string.digits, k=4))
        fmt_time = time.strftime("%y%m%d%H%M%S", time.localtime())
```

(continues on next page)

(continued from previous page)

```
        return f"image_artifact_{fmt_time}_{entropy}.{self.format}"
```

- `format = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `height = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `mime_type` property
- `width = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/artifacts/image_artifact.py

```
def __attrs_post_init__(self) -> None:
    # Generating the name string requires attributes set by child classes.
    # This waits until all attributes are available before generating a name.
    if self.name == self.id:
        self.name = self.make_name()
```

make_name()

Source Code in griptape/artifacts/image_artifact.py

```
def make_name(self) -> str:
    entropy = "".join(random.choices(string.ascii_lowercase + string.digits, k=4))
    fmt_time = time.strftime("%y%m%d%H%M%S", time.localtime())

    return f"image_artifact_{fmt_time}_{entropy}.{self.format}"
```

to_bytes(**kwargs)

Source Code in griptape/artifacts/image_artifact.py

```
def to_bytes(self, **kwargs) -> bytes:
    return self.value
```

to_text()

Source Code in griptape/artifacts/image_artifact.py

```
def to_text(self) -> str:
    return f"Image, format: {self.format}, size: {len(self.value)} bytes"
```

image_url_artifact

Bases: *ImageUrlArtifact* “)

Attributes

Name	Type	Description
<a href="./#griptape.artifacts.BaseArtifact.value" title="value = field() class-attribute instance-attribute (griptape.artifacts.image_url_artifact.ImageUrlArtifact.value)"><code class="language-python">	*str*	The url.

Source Code in griptape/artifacts/image_url_artifact.py

```
class ImageUrlArtifact(UrlArtifact):
    """Stores a url to an image.

    Attributes:
        value: The url.
    """
```

info_artifact

Bases: *BaseArtifact* “)

Attributes

Name	Type	Description
<a href="info_artifact.mdx#griptape.artifacts.info_artifact.InfoArtifact.value" title="value = field(converter=str, metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.info_artifact.InfoArtifact.value)"><code class="language-python">	*str*	The info to convey.

Source Code in griptape/artifacts/info_artifact.py

```
@define
class InfoArtifact(BaseArtifact):
    """Represents helpful info that can be conveyed to the LLM.
```

(continues on next page)

(continued from previous page)

```
    For example, "No results found" or "Please try again.".

    Attributes:
        value: The info to convey.
    """

    value: str = field(converter=str, metadata={"serializable": True})

    def to_text(self) -> str:
        return self.value
```

- `value = field(converter=str, metadata={'serializable': True})` class-attribute instance-attribute

to_text()

Source Code in griptape/artifacts/info_artifact.py

```
def to_text(self) -> str:
    return self.value
```

json_artifact

- `Json = Union[dict[str, 'Json'], list['Json'], str, int, float, bool, None]` module-attribute

Bases: *BaseArtifact* “)

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">Json</code>	The JSON data. Values will automatically be converted to a JSON-compatible format.

Source Code in griptape/artifacts/json_artifact.py

```
@define
class JsonArtifact(BaseArtifact):
    """Stores JSON data.

    Attributes:
        value: The JSON data. Values will automatically be converted to a JSON-
↪compatible format.
    """

    value: Json = field(converter=lambda value: JsonArtifact.value_to_json(value),␣
↪metadata={"serializable": True})

    @classmethod
    def value_to_json(cls, value: Any) -> Json:
        if isinstance(value, str):
            return json.loads(value)
        return json.loads(json.dumps(value))

    def to_text(self) -> str:
        return json.dumps(self.value)
```

- `value = field(converter=lambda value: JsonArtifact.value_to_json(value), metadata={'serializable': True})` class-attribute instance-attribute

to_text()

Source Code in griptape/artifacts/json_artifact.py

```
def to_text(self) -> str:
    return json.dumps(self.value)
```

value_to_json(value)classmethod

Source Code in griptape/artifacts/json_artifact.py

```
@classmethod
def value_to_json(cls, value: Any) -> Json:
    if isinstance(value, str):
        return json.loads(value)
    return json.loads(json.dumps(value))
```

list_artifact

- `T_co = TypeVar('T_co', bound=BaseArtifact, covariant=True)` module-attribute

Bases: *BaseArtifact* “) , *Generic[T_co*

Source Code in griptape/artifacts/list_artifact.py

```
@define
class ListArtifact(BaseArtifact, Generic[T_co]):
    value: Sequence[T_co] = field(factory=list, metadata={"serializable": True})
    item_separator: str = field(default="\n\n", kw_only=True, metadata={"serializable
```

(continues on next page)

(continued from previous page)

```
↪": True})
    validate_uniform_types: bool = field(default=False, kw_only=True, metadata={
↪"serializable": True})

    @value.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_value(self, _: Attribute, value: list[T_co]) -> None:
        if self.validate_uniform_types and len(value) > 0:
            first_type = type(value[0])

            if not all(isinstance(v, first_type) for v in value):
                raise ValueError("list elements in value are not the same type")

    @property
    def child_type(self) -> Optional[type]:
        if self.value:
            return type(self.value[0])
        return None

    def __getitem__(self, key: int) -> T_co:
        return self.value[key]

    def __bool__(self) -> bool:
        return len(self) > 0

    def __add__(self, other: BaseArtifact) -> ListArtifact[T_co]:
        return ListArtifact(self.value + other.value)

    def __iter__(self) -> Iterator[T_co]:
        return iter(self.value)

    def to_text(self) -> str:
        return self.item_separator.join([v.to_text() for v in self.value])

    def is_type(self, target_type: type) -> bool:
        if self.value:
            return isinstance(self.value[0], target_type)
        return False

    def has_items(self) -> bool:
        return len(self) > 0
```

- `child_type` property
- `item_separator = field(default='\n\n', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `validate_uniform_types = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `value = field(factory=list, metadata={'serializable': True})` class-attribute instance-attribute

add(other)

Source Code in griptape/artifacts/list_artifact.py

```
def __add__(self, other: BaseArtifact) -> ListArtifact[T_co]:
    return ListArtifact(self.value + other.value)
```

bool()

Source Code in griptape/artifacts/list_artifact.py

```
def __bool__(self) -> bool:
    return len(self) > 0
```

getitem(key)

Source Code in griptape/artifacts/list_artifact.py

```
def __getitem__(self, key: int) -> T_co:
    return self.value[key]
```

iter()

Source Code in griptape/artifacts/list_artifact.py

```
def __iter__(self) -> Iterator[T_co]:
    return iter(self.value)
```

has_items()

Source Code in griptape/artifacts/list_artifact.py

```
def has_items(self) -> bool:
    return len(self) > 0
```

is_type(target_type)

Source Code in griptape/artifacts/list_artifact.py

```
def is_type(self, target_type: type) -> bool:
    if self.value:
        return isinstance(self.value[0], target_type)
    return False
```

to_text()

Source Code in griptape/artifacts/list_artifact.py

```
def to_text(self) -> str:
    return self.item_separator.join([v.to_text() for v in self.value])
```

validate*value(*, value)

Source Code in griptape/artifacts/list_artifact.py

```
@value.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_value(self, _: Attribute, value: list[T_co]) -> None:
    if self.validate_uniform_types and len(value) > 0:
        first_type = type(value[0])

        if not all(isinstance(v, first_type) for v in value):
            raise ValueError("list elements in value are not the same type")
```

model_artifact

Bases: *GenericArtifact[BaseModel* “)

Attributes

Name	Type	Description
<a href="model_artifact.mdx#griptape.artifacts.model_artifact.ModelArtifact.value" title="value = field(metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.model_artifact.ModelArtifact.value)"><code class="language-python">	*BaseModel*	The pydantic model to store.

Source Code in griptape/artifacts/model_artifact.py

```
@define
class ModelArtifact(GenericArtifact[BaseModel]):
    """Stores Pydantic models as Artifacts.

    Required since Pydantic models require a custom serialization method.

    Attributes:
        value: The pydantic model to store.
    """

    # We must explicitly define the type rather than rely on the parent T since
    # generic type information is lost at runtime.
    value: BaseModel = field(metadata={"serializable": True})

    def to_text(self) -> str:
        return self.value.model_dump_json()
```

- `value = field(metadata={'serializable': True})` class-attribute instance-attribute

to_text()

Source Code in griptape/artifacts/model_artifact.py

```
def to_text(self) -> str:
    return self.value.model_dump_json()
```

text_artifact

Bases: *BaseArtifact* “)

Source Code in griptape/artifacts/text_artifact.py

```
@define
class TextArtifact(BaseArtifact):
    value: str = field(converter=str, metadata={"serializable": True})
    embedding: Optional[list[float]] = field(default=None, kw_only=True)

    def __add__(self, other: BaseArtifact) -> TextArtifact:
        return TextArtifact(self.value + other.value)

    def __bool__(self) -> bool:
        return bool(self.value.strip())

    def to_text(self) -> str:
        return self.value

    def generate_embedding(self, driver: BaseEmbeddingDriver) -> list[float]:
        embedding = driver.embed(str(self.value))

        if self.embedding is None:
            self.embedding = []
        self.embedding.clear()
        self.embedding.extend(embedding)

        return self.embedding

    def token_count(self, tokenizer: BaseTokenizer) -> int:
        return tokenizer.count_tokens(str(self.value))
```

- `embedding = field(default=None, kw_only=True)` class-attribute instance-attribute
- `value = field(converter=str, metadata={'serializable': True})` class-attribute instance-attribute

add(other)

Source Code in griptape/artifacts/text_artifact.py

```
def __add__(self, other: BaseArtifact) -> TextArtifact:
    return TextArtifact(self.value + other.value)
```

bool()

Source Code in griptape/artifacts/text_artifact.py

```
def __bool__(self) -> bool:
    return bool(self.value.strip())
```

generate_embedding(driver)

Source Code in griptape/artifacts/text_artifact.py

```
def generate_embedding(self, driver: BaseEmbeddingDriver) -> list[float]:
    embedding = driver.embed(str(self.value))

    if self.embedding is None:
        self.embedding = []
    self.embedding.clear()
    self.embedding.extend(embedding)

    return self.embedding
```

to_text()

Source Code in griptape/artifacts/text_artifact.py

```
def to_text(self) -> str:
    return self.value
```

token_count(tokenizer)

Source Code in griptape/artifacts/text_artifact.py

```
def token_count(self, tokenizer: BaseTokenizer) -> int:
    return tokenizer.count_tokens(str(self.value))
```

url_artifact

Bases: *BaseArtifact* “)

Attributes

Name	Type	Description
<a href="url_artifact.mdx#griptape.artifacts.url_artifact.UrlArtifact.value" title="value = field(metadata={'serializable': True}) class-attribute instance-attribute (griptape.artifacts.url_artifact.UrlArtifact.value)"><code class="language-python">	*str*	The url.

Source Code in griptape/artifacts/url_artifact.py

```python
@define
class UrlArtifact(BaseArtifact):
    """Stores a url.

    Attributes:
        value: The url.
    """

    value: str = field(metadata={"serializable": True})

    def to_bytes(self, *, headers: dict | None = None, **kwargs: dict) -> bytes:
        """Fetches the content of the URL and returns it as bytes.

        Args:
            headers: Optional headers to include in the request.
            **kwargs: Additional keyword arguments, not used.

        Returns:
            bytes: The content of the URL as bytes.

        """
        response = requests.get(self.value, headers=headers)
        response.raise_for_status()

        return response.content

    def to_text(self) -> str:
        """Returns the URL as is."""
        return self.value
```

- `value = field(metadata={'serializable': True})` class-attribute instance-attribute

to_bytes(*, headers=None, **kwargs)

Fetches the content of the URL and returns it as bytes.

Parameters

Name	Type	Description	Default
headers	*dict I None*	Optional headers to include in the request. 	*None*
***kwargs*	*dict*	Additional keyword arguments, not used. 	*{}*

Returns

Name	Type	Description
bytes	*bytes*	The content of the URL as bytes.

Source Code in griptape/artifacts/url_artifact.py

```
def to_bytes(self, *, headers: dict | None = None, **kwargs: dict) -> bytes:
    """Fetches the content of the URL and returns it as bytes.

    Args:
        headers: Optional headers to include in the request.
        **kwargs: Additional keyword arguments, not used.

    Returns:
        bytes: The content of the URL as bytes.

    """
    response = requests.get(self.value, headers=headers)
    response.raise_for_status()

    return response.content
```

to_text()

Source Code in griptape/artifacts/url_artifact.py

```
def to_text(self) -> str:
    """Returns the URL as is."""
    return self.value
```

11.5.2 chunkers

- `__all__ = ['BaseChunker', 'ChunkSeparator', 'MarkdownChunker', 'PdfChunker', 'TextChunker']` module-attribute

Bases:

ABC

Source Code in griptape/chunkers/base_chunker.py

```
@define
class BaseChunker(ABC):
    DEFAULT_SEPARATORS = [ChunkSeparator(" ")]

    separators: list[ChunkSeparator] = field(
        default=Factory(lambda self: self.DEFAULT_SEPARATORS, takes_self=True),
        kw_only=True,
    )
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda: OpenAiTokenizer(model=OpenAiTokenizer.DEFAULT_OPENAI_
↪GPT_3_CHAT_MODEL)),
        kw_only=True,
    )
    max_tokens: int = field(
        default=Factory(lambda self: self.tokenizer.max_input_tokens, takes_
↪self=True),
        kw_only=True,
    )

    @max_tokens.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_max_tokens(self, _: Attribute, max_tokens: int) -> None:
        if max_tokens < 0:
            raise ValueError("max_tokens must be 0 or greater.")

    def chunk(self, text: TextArtifact | ListArtifact | str) -> list[TextArtifact]:
        text_to_chunk = text if isinstance(text, str) else text.to_text()
        reference = None if isinstance(text, str) else text.reference

        return [TextArtifact(c, reference=reference) for c in self._chunk_
↪recursively(text_to_chunk)]

    def _chunk_recursively(self, chunk: str, current_separator:␣
↪Optional[ChunkSeparator] = None) -> list[str]:
        token_count = self.tokenizer.count_tokens(chunk)
        half_token_count = token_count // 2

        if token_count <= self.max_tokens:
            return [chunk]
        # If a separator is provided, only use separators after it.
        separators = (
            self.separators[self.separators.index(current_separator) :] if current_
↪separator else self.separators
        )
```

(continues on next page)

(continued from previous page)

```
        # Loop through available separators to find the best split.
        for separator in separators:
            # Split the chunk into subchunks using the current separator.
            subchunks = chunk.strip().split(separator.value)

            # We should not operate on the filtered, non-empty subchunks because the␣
↪joins will be incorrect.
            # However, we only want to process chunks that have multiple non-empty␣
↪subchunks.
            # Therefore, we use the non-empty subchunks to decide if we should␣
↪proceed, but we operate on the original subchunks.
            non_empty_subchunks = list(filter(None, subchunks))

            if len(non_empty_subchunks) > 1:
                # Find what combination of subchunks results in the most balanced␣
↪split of the chunk.
                midpoint_index = self.__find_midpoint_index(separator, subchunks,␣
↪half_token_count)

                # Create the two subchunks based on the best separator.
                first_subchunk, second_subchunk = self.__get_subchunks(separator,␣
↪subchunks, midpoint_index)

                # Continue recursively chunking the subchunks.
                first_subchunk_rec = self._chunk_recursively(first_subchunk.strip(),␣
↪separator)
                second_subchunk_rec = self._chunk_recursively(second_subchunk.strip(),
↪ separator)

                # Return the concatenated results of the subchunks if both are non-
↪empty.
                if first_subchunk_rec and second_subchunk_rec:
                    return first_subchunk_rec + second_subchunk_rec
                # If only one subchunk is non-empty, return it.
                if first_subchunk_rec:
                    return first_subchunk_rec
                if second_subchunk_rec:
                    return second_subchunk_rec
                return []
        # If none of the separators result in a balanced split, split the chunk in␣
↪half.
        midpoint = len(chunk) // 2
        return self._chunk_recursively(chunk[:midpoint]) + self._chunk_
↪recursively(chunk[midpoint:])

    def __get_subchunks(self, separator: ChunkSeparator, subchunks: list[str],␣
↪balance_index: int) -> tuple[str, str]:
        # Create the two subchunks based on the best separator
        if separator.is_prefix:
            first_subchunk = separator.value.join(subchunks[: balance_index + 1])
            # We need to manually prepend the separator since join doesnt add it to␣
↪the first element.
```

(continues on next page)

(continued from previous page)

```
            second_subchunk = separator.value + separator.value.
↪join(subchunks[balance_index + 1 :])
        else:
            # We need to manually append the separator since join doesnt add it to␣
↪the last element.
            first_subchunk = separator.value.join(subchunks[: balance_index + 1]) +␣
↪separator.value
            second_subchunk = separator.value.join(subchunks[balance_index + 1 :])

        return first_subchunk, second_subchunk

    def __find_midpoint_index(self, separator: ChunkSeparator, subchunks: list[str],␣
↪half_token_count: int) -> int:
        midpoint_index = -1
        best_midpoint_distance = float("inf")

        for index, _ in enumerate(subchunks):
            subchunk_tokens_count = self.tokenizer.count_tokens(separator.value.
↪join(subchunks[: index + 1]))

            midpoint_distance = abs(subchunk_tokens_count - half_token_count)
            if midpoint_distance < best_midpoint_distance:
                midpoint_index = index
                best_midpoint_distance = midpoint_distance

        return midpoint_index
```

- `DEFAULT_SEPARATORS = [ChunkSeparator(' ')]` class-attribute instance-attribute
- `max_tokens = field(default=Factory(lambda self: self.tokenizer.max_input_tokens, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `separators = field(default=Factory(lambda self: self.DEFAULT_SEPARATORS, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda: OpenAiTokenizer(model=OpenAiTokenizer.DEFAULT_OPENAI_GPT_3_CHAT_MODEL)), kw_only=True)` class-attribute instance-attribute

__find_midpoint_index(separator, subchunks, half_token_count)

Source Code in griptape/chunkers/base_chunker.py

```
def __find_midpoint_index(self, separator: ChunkSeparator, subchunks: list[str], half_
↪token_count: int) -> int:
    midpoint_index = -1
    best_midpoint_distance = float("inf")

    for index, _ in enumerate(subchunks):
        subchunk_tokens_count = self.tokenizer.count_tokens(separator.value.
↪join(subchunks[: index + 1]))

        midpoint_distance = abs(subchunk_tokens_count - half_token_count)
        if midpoint_distance < best_midpoint_distance:
            midpoint_index = index
```

(continues on next page)

(continued from previous page)

```
            best_midpoint_distance = midpoint_distance

    return midpoint_index
```

__get_subchunks(separator, subchunks, balance_index)

Source Code in griptape/chunkers/base_chunker.py

```
def __get_subchunks(self, separator: ChunkSeparator, subchunks: list[str], balance_
↪index: int) -> tuple[str, str]:
    # Create the two subchunks based on the best separator
    if separator.is_prefix:
        first_subchunk = separator.value.join(subchunks[: balance_index + 1])
        # We need to manually prepend the separator since join doesnt add it to the␣
↪first element.
        second_subchunk = separator.value + separator.value.join(subchunks[balance_
↪index + 1 :])
    else:
        # We need to manually append the separator since join doesnt add it to the␣
↪last element.
        first_subchunk = separator.value.join(subchunks[: balance_index + 1]) +␣
↪separator.value
        second_subchunk = separator.value.join(subchunks[balance_index + 1 :])

    return first_subchunk, second_subchunk
```

_chunk_recursively(chunk, current_separator=None)

Source Code in griptape/chunkers/base_chunker.py

```
def _chunk_recursively(self, chunk: str, current_separator: Optional[ChunkSeparator]␣
↪= None) -> list[str]:
    token_count = self.tokenizer.count_tokens(chunk)
    half_token_count = token_count // 2

    if token_count <= self.max_tokens:
        return [chunk]
    # If a separator is provided, only use separators after it.
    separators = (
        self.separators[self.separators.index(current_separator) :] if current_
↪separator else self.separators
    )

    # Loop through available separators to find the best split.
    for separator in separators:
        # Split the chunk into subchunks using the current separator.
        subchunks = chunk.strip().split(separator.value)

        # We should not operate on the filtered, non-empty subchunks because the␣
↪joins will be incorrect.
        # However, we only want to process chunks that have multiple non-empty␣
↪subchunks.
```

(continues on next page)

(continued from previous page)

```
        # Therefore, we use the non-empty subchunks to decide if we should proceed,␣
↪but we operate on the original subchunks.
        non_empty_subchunks = list(filter(None, subchunks))

        if len(non_empty_subchunks) > 1:
            # Find what combination of subchunks results in the most balanced split␣
↪of the chunk.
            midpoint_index = self.__find_midpoint_index(separator, subchunks, half_
↪token_count)

            # Create the two subchunks based on the best separator.
            first_subchunk, second_subchunk = self.__get_subchunks(separator,␣
↪subchunks, midpoint_index)

            # Continue recursively chunking the subchunks.
            first_subchunk_rec = self._chunk_recursively(first_subchunk.strip(),␣
↪separator)
            second_subchunk_rec = self._chunk_recursively(second_subchunk.strip(),␣
↪separator)

            # Return the concatenated results of the subchunks if both are non-empty.
            if first_subchunk_rec and second_subchunk_rec:
                return first_subchunk_rec + second_subchunk_rec
            # If only one subchunk is non-empty, return it.
            if first_subchunk_rec:
                return first_subchunk_rec
            if second_subchunk_rec:
                return second_subchunk_rec
            return []
    # If none of the separators result in a balanced split, split the chunk in half.
    midpoint = len(chunk) // 2
    return self._chunk_recursively(chunk[:midpoint]) + self._chunk_
↪recursively(chunk[midpoint:])
```

chunk(text)

Source Code in griptape/chunkers/base_chunker.py

```
def chunk(self, text: TextArtifact | ListArtifact | str) -> list[TextArtifact]:
    text_to_chunk = text if isinstance(text, str) else text.to_text()
    reference = None if isinstance(text, str) else text.reference

    return [TextArtifact(c, reference=reference) for c in self._chunk_
↪recursively(text_to_chunk)]
```

validate*max_tokens*(, max_tokens)

Source Code in griptape/chunkers/base_chunker.py

```
@max_tokens.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_max_tokens(self, _: Attribute, max_tokens: int) -> None:
```

(continues on next page)

(continued from previous page)

```
    if max_tokens < 0:
        raise ValueError("max_tokens must be 0 or greater.")
```

ChunkSeparatordataclass

Source Code in griptape/chunkers/chunk_separator.py

```
@dataclass
class ChunkSeparator:
    value: str
    is_prefix: bool = False
```

- `is_prefix = False` class-attribute instance-attribute
- `value` instance-attribute

init(value, is_prefix=False)

MarkdownChunker

Bases: *BaseChunker* “)

Source Code in griptape/chunkers/markdown_chunker.py

```
class MarkdownChunker(BaseChunker):
    DEFAULT_SEPARATORS = [
        ChunkSeparator("##", is_prefix=True),
        ChunkSeparator("###", is_prefix=True),
        ChunkSeparator("####", is_prefix=True),
        ChunkSeparator("#####", is_prefix=True),
        ChunkSeparator("######", is_prefix=True),
        ChunkSeparator("\n\n"),
        ChunkSeparator(". "),
        ChunkSeparator("! "),
        ChunkSeparator("? "),
        ChunkSeparator(" "),
    ]
```

- `DEFAULT_SEPARATORS = [ChunkSeparator('##', is_prefix=True), ChunkSeparator('###', is_prefix=True), ChunkSeparator('####', is_prefix=True), ChunkSeparator('#####', is_prefix=True), ChunkSeparator('######', is_prefix=True), ChunkSeparator('\n\n'), ChunkSeparator('. '), ChunkSeparator('! '), ChunkSeparator('? '), ChunkSeparator(' ')]` class-attribute instance-attribute

PdfChunker

Bases: *BaseChunker* “)

Source Code in griptape/chunkers/pdf_chunker.py

```
class PdfChunker(BaseChunker):
    DEFAULT_SEPARATORS = [
        ChunkSeparator("\n\n"),
        ChunkSeparator(". "),
```

(continues on next page)

(continued from previous page)

```
        ChunkSeparator("! "),
        ChunkSeparator("? "),
        ChunkSeparator(" "),
    ]
```

- `DEFAULT_SEPARATORS = [ChunkSeparator('\n\n'), ChunkSeparator('. '), ChunkSeparator('! '), ChunkSeparator('? '), ChunkSeparator(' ')]` class-attribute instance-attribute

TextChunker

Bases: *BaseChunker* “)

Source Code in griptape/chunkers/text_chunker.py

```
class TextChunker(BaseChunker):
    DEFAULT_SEPARATORS = [
        ChunkSeparator("\n\n"),
        ChunkSeparator("\n"),
        ChunkSeparator(". "),
        ChunkSeparator("! "),
        ChunkSeparator("? "),
        ChunkSeparator(" "),
    ]
```

- `DEFAULT_SEPARATORS = [ChunkSeparator('\n\n'), ChunkSeparator('\n'), ChunkSeparator('. '), ChunkSeparator('! '), ChunkSeparator('? '), ChunkSeparator(' ')]` class-attribute instance-attribute

base_chunker

Bases:

ABC

Source Code in griptape/chunkers/base_chunker.py

```
@define
class BaseChunker(ABC):
    DEFAULT_SEPARATORS = [ChunkSeparator(" ")]

    separators: list[ChunkSeparator] = field(
        default=Factory(lambda self: self.DEFAULT_SEPARATORS, takes_self=True),
        kw_only=True,
    )
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda: OpenAiTokenizer(model=OpenAiTokenizer.DEFAULT_OPENAI_
↪GPT_3_CHAT_MODEL)),
        kw_only=True,
    )
    max_tokens: int = field(
        default=Factory(lambda self: self.tokenizer.max_input_tokens, takes_
↪self=True),
        kw_only=True,
```

(continues on next page)

(continued from previous page)

```
    )

    @max_tokens.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_max_tokens(self, _: Attribute, max_tokens: int) -> None:
        if max_tokens < 0:
            raise ValueError("max_tokens must be 0 or greater.")

    def chunk(self, text: TextArtifact | ListArtifact | str) -> list[TextArtifact]:
        text_to_chunk = text if isinstance(text, str) else text.to_text()
        reference = None if isinstance(text, str) else text.reference

        return [TextArtifact(c, reference=reference) for c in self._chunk_
↪recursively(text_to_chunk)]

    def _chunk_recursively(self, chunk: str, current_separator:␣
↪Optional[ChunkSeparator] = None) -> list[str]:
        token_count = self.tokenizer.count_tokens(chunk)
        half_token_count = token_count // 2

        if token_count <= self.max_tokens:
            return [chunk]
        # If a separator is provided, only use separators after it.
        separators = (
            self.separators[self.separators.index(current_separator) :] if current_
↪separator else self.separators
        )

        # Loop through available separators to find the best split.
        for separator in separators:
            # Split the chunk into subchunks using the current separator.
            subchunks = chunk.strip().split(separator.value)

            # We should not operate on the filtered, non-empty subchunks because the␣
↪joins will be incorrect.
            # However, we only want to process chunks that have multiple non-empty␣
↪subchunks.
            # Therefore, we use the non-empty subchunks to decide if we should␣
↪proceed, but we operate on the original subchunks.
            non_empty_subchunks = list(filter(None, subchunks))

            if len(non_empty_subchunks) > 1:
                # Find what combination of subchunks results in the most balanced␣
↪split of the chunk.
                midpoint_index = self.__find_midpoint_index(separator, subchunks,␣
↪half_token_count)

                # Create the two subchunks based on the best separator.
                first_subchunk, second_subchunk = self.__get_subchunks(separator,␣
↪subchunks, midpoint_index)

                # Continue recursively chunking the subchunks.
                first_subchunk_rec = self._chunk_recursively(first_subchunk.strip(),␣
```

(continues on next page)

(continued from previous page)

```
↪separator)
                second_subchunk_rec = self._chunk_recursively(second_subchunk.strip(),
↪ separator)

                # Return the concatenated results of the subchunks if both are non-
↪empty.
                if first_subchunk_rec and second_subchunk_rec:
                    return first_subchunk_rec + second_subchunk_rec
                # If only one subchunk is non-empty, return it.
                if first_subchunk_rec:
                    return first_subchunk_rec
                if second_subchunk_rec:
                    return second_subchunk_rec
                return []
        # If none of the separators result in a balanced split, split the chunk in␣
↪half.
        midpoint = len(chunk) // 2
        return self._chunk_recursively(chunk[:midpoint]) + self._chunk_
↪recursively(chunk[midpoint:])

    def __get_subchunks(self, separator: ChunkSeparator, subchunks: list[str],␣
↪balance_index: int) -> tuple[str, str]:
        # Create the two subchunks based on the best separator
        if separator.is_prefix:
            first_subchunk = separator.value.join(subchunks[: balance_index + 1])
            # We need to manually prepend the separator since join doesnt add it to␣
↪the first element.
            second_subchunk = separator.value + separator.value.
↪join(subchunks[balance_index + 1 :])
        else:
            # We need to manually append the separator since join doesnt add it to␣
↪the last element.
            first_subchunk = separator.value.join(subchunks[: balance_index + 1]) +␣
↪separator.value
            second_subchunk = separator.value.join(subchunks[balance_index + 1 :])

        return first_subchunk, second_subchunk

    def __find_midpoint_index(self, separator: ChunkSeparator, subchunks: list[str],␣
↪half_token_count: int) -> int:
        midpoint_index = -1
        best_midpoint_distance = float("inf")

        for index, _ in enumerate(subchunks):
            subchunk_tokens_count = self.tokenizer.count_tokens(separator.value.
↪join(subchunks[: index + 1]))

            midpoint_distance = abs(subchunk_tokens_count - half_token_count)
            if midpoint_distance < best_midpoint_distance:
                midpoint_index = index
                best_midpoint_distance = midpoint_distance
```

(continues on next page)

(continued from previous page)

```
        return midpoint_index
```

- `DEFAULT_SEPARATORS = [ChunkSeparator(' ')]` class-attribute instance-attribute
- `max_tokens = field(default=Factory(lambda self: self.tokenizer.max_input_tokens, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `separators = field(default=Factory(lambda self: self.DEFAULT_SEPARATORS, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda: OpenAiTokenizer(model=OpenAiTokenizer.DEFAULT_OPENAI_GPT_3_CHAT_MODEL)), kw_only=True)` class-attribute instance-attribute

__find_midpoint_index(separator, subchunks, half_token_count)

Source Code in griptape/chunkers/base_chunker.py

```
def __find_midpoint_index(self, separator: ChunkSeparator, subchunks: list[str], half_
↪token_count: int) -> int:
    midpoint_index = -1
    best_midpoint_distance = float("inf")

    for index, _ in enumerate(subchunks):
        subchunk_tokens_count = self.tokenizer.count_tokens(separator.value.
↪join(subchunks[: index + 1]))

        midpoint_distance = abs(subchunk_tokens_count - half_token_count)
        if midpoint_distance < best_midpoint_distance:
            midpoint_index = index
            best_midpoint_distance = midpoint_distance

    return midpoint_index
```

__get_subchunks(separator, subchunks, balance_index)

Source Code in griptape/chunkers/base_chunker.py

```
def __get_subchunks(self, separator: ChunkSeparator, subchunks: list[str], balance_
↪index: int) -> tuple[str, str]:
    # Create the two subchunks based on the best separator
    if separator.is_prefix:
        first_subchunk = separator.value.join(subchunks[: balance_index + 1])
        # We need to manually prepend the separator since join doesnt add it to the␣
↪first element.
        second_subchunk = separator.value + separator.value.join(subchunks[balance_
↪index + 1 :])
    else:
        # We need to manually append the separator since join doesnt add it to the␣
↪last element.
        first_subchunk = separator.value.join(subchunks[: balance_index + 1]) +␣
↪separator.value
        second_subchunk = separator.value.join(subchunks[balance_index + 1 :])

    return first_subchunk, second_subchunk
```

_chunk_recursively(chunk, current_separator=None)

Source Code in griptape/chunkers/base_chunker.py

```
def _chunk_recursively(self, chunk: str, current_separator: Optional[ChunkSeparator]␣
↪= None) -> list[str]:
    token_count = self.tokenizer.count_tokens(chunk)
    half_token_count = token_count // 2

    if token_count <= self.max_tokens:
        return [chunk]
    # If a separator is provided, only use separators after it.
    separators = (
        self.separators[self.separators.index(current_separator) :] if current_
↪separator else self.separators
    )

    # Loop through available separators to find the best split.
    for separator in separators:
        # Split the chunk into subchunks using the current separator.
        subchunks = chunk.strip().split(separator.value)

        # We should not operate on the filtered, non-empty subchunks because the␣
↪joins will be incorrect.
        # However, we only want to process chunks that have multiple non-empty␣
↪subchunks.
        # Therefore, we use the non-empty subchunks to decide if we should proceed,␣
↪but we operate on the original subchunks.
        non_empty_subchunks = list(filter(None, subchunks))

        if len(non_empty_subchunks) > 1:
            # Find what combination of subchunks results in the most balanced split␣
↪of the chunk.
            midpoint_index = self.__find_midpoint_index(separator, subchunks, half_
↪token_count)

            # Create the two subchunks based on the best separator.
            first_subchunk, second_subchunk = self.__get_subchunks(separator,␣
↪subchunks, midpoint_index)

            # Continue recursively chunking the subchunks.
            first_subchunk_rec = self._chunk_recursively(first_subchunk.strip(),␣
↪separator)
            second_subchunk_rec = self._chunk_recursively(second_subchunk.strip(),␣
↪separator)

            # Return the concatenated results of the subchunks if both are non-empty.
            if first_subchunk_rec and second_subchunk_rec:
                return first_subchunk_rec + second_subchunk_rec
            # If only one subchunk is non-empty, return it.
            if first_subchunk_rec:
                return first_subchunk_rec
            if second_subchunk_rec:
                return second_subchunk_rec
```

(continues on next page)

(continued from previous page)

```
            return []
    # If none of the separators result in a balanced split, split the chunk in half.
    midpoint = len(chunk) // 2
    return self._chunk_recursively(chunk[:midpoint]) + self._chunk_
↪recursively(chunk[midpoint:])
```

chunk(text)

Source Code in griptape/chunkers/base_chunker.py

```
def chunk(self, text: TextArtifact | ListArtifact | str) -> list[TextArtifact]:
    text_to_chunk = text if isinstance(text, str) else text.to_text()
    reference = None if isinstance(text, str) else text.reference

    return [TextArtifact(c, reference=reference) for c in self._chunk_
↪recursively(text_to_chunk)]
```

validate*max_tokens(*, max_tokens)

Source Code in griptape/chunkers/base_chunker.py

```
@max_tokens.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_max_tokens(self, _: Attribute, max_tokens: int) -> None:
    if max_tokens < 0:
        raise ValueError("max_tokens must be 0 or greater.")
```

chunk_separator

Source Code in griptape/chunkers/chunk_separator.py

```
@dataclass
class ChunkSeparator:
    value: str
    is_prefix: bool = False
```

- `is_prefix = False` class-attribute instance-attribute
- `value` instance-attribute

init(value, is_prefix=False)

markdown_chunker

Bases: *BaseChunker* “)

Source Code in griptape/chunkers/markdown_chunker.py

```
class MarkdownChunker(BaseChunker):
    DEFAULT_SEPARATORS = [
        ChunkSeparator("##", is_prefix=True),
        ChunkSeparator("###", is_prefix=True),
        ChunkSeparator("####", is_prefix=True),
```

(continues on next page)

(continued from previous page)

```
        ChunkSeparator("#####", is_prefix=True),
        ChunkSeparator("######", is_prefix=True),
        ChunkSeparator("\n\n"),
        ChunkSeparator(". "),
        ChunkSeparator("! "),
        ChunkSeparator("? "),
        ChunkSeparator(" "),
    ]
```

- `DEFAULT_SEPARATORS = [ChunkSeparator('##', is_prefix=True), ChunkSeparator('###', is_prefix=True), ChunkSeparator('####', is_prefix=True), ChunkSeparator('#####', is_prefix=True), ChunkSeparator('######', is_prefix=True), ChunkSeparator('\n\n'), ChunkSeparator('. '), ChunkSeparator('! '), ChunkSeparator('? '), ChunkSeparator(' ')]` class-attribute instance-attribute

pdf_chunker

Bases: *BaseChunker* “)

Source Code in griptape/chunkers/pdf_chunker.py

```
class PdfChunker(BaseChunker):
    DEFAULT_SEPARATORS = [
        ChunkSeparator("\n\n"),
        ChunkSeparator(". "),
        ChunkSeparator("! "),
        ChunkSeparator("? "),
        ChunkSeparator(" "),
    ]
```

- `DEFAULT_SEPARATORS = [ChunkSeparator('\n\n'), ChunkSeparator('. '), ChunkSeparator('! '), ChunkSeparator('? '), ChunkSeparator(' ')]` class-attribute instance-attribute

text_chunker

Bases: *BaseChunker* “)

Source Code in griptape/chunkers/text_chunker.py

```
class TextChunker(BaseChunker):
    DEFAULT_SEPARATORS = [
        ChunkSeparator("\n\n"),
        ChunkSeparator("\n"),
        ChunkSeparator(". "),
        ChunkSeparator("! "),
        ChunkSeparator("? "),
        ChunkSeparator(" "),
    ]
```

- `DEFAULT_SEPARATORS = [ChunkSeparator('\n\n'), ChunkSeparator('\n'), ChunkSeparator('. '), ChunkSeparator('! '), ChunkSeparator('? '), ChunkSeparator(' ')]` class-attribute instance-attribute

11.5.3 common

- `__all__ = ['ActionCallDeltaMessageContent', 'ActionCallMessageContent', 'ActionResultMessageContent', 'AudioDeltaMessageContent', 'AudioMessageContent', 'BaseAction', 'BaseDeltaMessageContent', 'BaseMessage', 'BaseMessageContent', 'DeltaMessage', 'GenericMessageContent', 'ImageMessageContent', 'Message', 'Observable', 'PromptStack', 'Reference', 'TextDeltaMessageContent', 'TextMessageContent', 'ToolAction', 'observable']` module-attribute

Bases: *BaseDeltaMessageContent* “)

Source Code in griptape/common/prompt_stack/contents/action_call_delta_message_content.py

```
@define
class ActionCallDeltaMessageContent(BaseDeltaMessageContent):
    tag: Optional[str] = field(default=None, metadata={"serializable": True})
    name: Optional[str] = field(default=None, metadata={"serializable": True})
    path: Optional[str] = field(default=None, metadata={"serializable": True})
    partial_input: Optional[str] = field(default=None, metadata={"serializable": True}
↪)

    def __str__(self) -> str:
        parts = []

        if self.name:
            parts.append(self.name)
            if self.path:
                parts.append(f".{self.path}")
                if self.tag:
                    parts.append(f" ({self.tag})")

        if self.partial_input:
            if parts:
                parts.append(f" {self.partial_input}")
            else:
                parts.append(self.partial_input)

        return "".join(parts)
```

- `name = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `partial_input = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `path = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `tag = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

str()

Source Code in griptape/common/prompt_stack/contents/action_call_delta_message_content.py

```
def __str__(self) -> str:
    parts = []
```

(continues on next page)

(continued from previous page)

```
    if self.name:
        parts.append(self.name)
        if self.path:
            parts.append(f".{self.path}")
            if self.tag:
                parts.append(f" ({self.tag})")

    if self.partial_input:
        if parts:
            parts.append(f" {self.partial_input}")
        else:
            parts.append(self.partial_input)

    return "".join(parts)
```

ActionCallMessageContent

Bases: *BaseMessageContent* “)

Source Code in griptape/common/prompt_stack/contents/action_call_message_content.py

```
@define
class ActionCallMessageContent(BaseMessageContent):
    artifact: ActionArtifact = field(metadata={"serializable": True})

    @classmethod
    def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪ActionCallMessageContent:
        action_call_deltas = [delta for delta in deltas if isinstance(delta,␣
↪ActionCallDeltaMessageContent)]

        tag = None
        name = None
        path = None
        json_input = ""

        for delta in action_call_deltas:
            if delta.tag is not None:
                tag = delta.tag
            if delta.name is not None:
                name = delta.name
            if delta.path is not None:
                path = delta.path
            if delta.partial_input is not None:
                json_input += delta.partial_input

        if tag is not None and name is not None and path is not None:
            try:
                parsed_input = json.loads(json_input or "{}")
            except json.JSONDecodeError as exc:
                raise ValueError("Invalid JSON input for ToolAction") from exc
            action = ToolAction(tag=tag, name=name, path=path, input=parsed_input)
```

(continues on next page)

(continued from previous page)

```
        else:
            raise ValueError("Missing required fields for ToolAction")

        artifact = ActionArtifact(value=action)

        return cls(artifact=artifact)
```

- `artifact = field(metadata={'serializable': True})` class-attribute instance-attribute

from_deltas(deltas)classmethod

Source Code in griptape/common/prompt_stack/contents/action_call_message_content.py

```
@classmethod
def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪ActionCallMessageContent:
    action_call_deltas = [delta for delta in deltas if isinstance(delta,␣
↪ActionCallDeltaMessageContent)]

    tag = None
    name = None
    path = None
    json_input = ""

    for delta in action_call_deltas:
        if delta.tag is not None:
            tag = delta.tag
        if delta.name is not None:
            name = delta.name
        if delta.path is not None:
            path = delta.path
        if delta.partial_input is not None:
            json_input += delta.partial_input

    if tag is not None and name is not None and path is not None:
        try:
            parsed_input = json.loads(json_input or "{}")
        except json.JSONDecodeError as exc:
            raise ValueError("Invalid JSON input for ToolAction") from exc
        action = ToolAction(tag=tag, name=name, path=path, input=parsed_input)
    else:
        raise ValueError("Missing required fields for ToolAction")

    artifact = ActionArtifact(value=action)

    return cls(artifact=artifact)
```

ActionResultMessageContent

Bases: *BaseMessageContent* “)

Source Code in griptape/common/prompt_stack/contents/action_result_message_content.py

```
@define
class ActionResultMessageContent(BaseMessageContent):
    artifact: BaseArtifact = field(metadata={"serializable": True})
    action: ToolAction = field(metadata={"serializable": True})

    @classmethod
    def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪ActionResultMessageContent:
        raise NotImplementedError
```

- `action = field(metadata={'serializable': True})` class-attribute instance-attribute
- `artifact = field(metadata={'serializable': True})` class-attribute instance-attribute

from_deltas(deltas)classmethod

Source Code in griptape/common/prompt_stack/contents/action_result_message_content.py

```
@classmethod
def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪ActionResultMessageContent:
    raise NotImplementedError
```

AudioDeltaMessageContent

Bases: *BaseDeltaMessageContent* “)

Attributes

Name	Type	Description
<a href="./#griptape.common.AudioDeltaMessageContent.id" title="id = field(default=None, metadata={'serializable': True}) class-attribute instance-attribute (griptape.common.AudioDeltaMessageContent.id)"><code class="language-python">	*Optional[str]*	The ID of the audio data.

continues on next page

Table 31 – continued from previous page

Name	Type	Description
<a href="./#griptape.common.AudioDeltaMessageContent.data" title="data = field(default=None, metadata={'serializable': True}) class-attribute instance-attribute (griptape.common.AudioDeltaMessageContent.data)"><code class="language-python">	*Optional[str]*	Base64 encoded audio data.
<a href="./#griptape.common.AudioDeltaMessageContent.transcript" title="transcript = field(default=None, metadata={'serializable': True}) class-attribute instance-attribute (griptape.common.AudioDeltaMessageContent.transcript)"><code class="language-python">	*Optional[str]*	The transcript of the audio data.
<a href="./#griptape.common.AudioDeltaMessageContent.expires_at" title="expires_at = field(default=None, metadata={'serializable': True}) class-attribute instance-attribute (griptape.common.AudioDeltaMessageContent.expires_at)"><code class="language-python">	*Optional[int]*	The Unix timestamp (in seconds) for when this audio data will no longer be accessible.

Source Code in griptape/common/prompt_stack/contents/audio_delta_message_content.py

```
@define(kw_only=True)
class AudioDeltaMessageContent(BaseDeltaMessageContent):
    """A delta message content for audio data.

    Attributes:
        id: The ID of the audio data.
```

(continues on next page)

(continued from previous page)

```
        data: Base64 encoded audio data.
        transcript: The transcript of the audio data.
        expires_at: The Unix timestamp (in seconds) for when this audio data will no␣
↪longer be accessible.
    """

    id: Optional[str] = field(default=None, metadata={"serializable": True})
    data: Optional[str] = field(default=None, metadata={"serializable": True})
    transcript: Optional[str] = field(default=None, metadata={"serializable": True})
    expires_at: Optional[int] = field(default=None, metadata={"serializable": True})
```

- `data = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `expires_at = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `id = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `transcript = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

AudioMessageContent

Bases: *BaseMessageContent* “)

Source Code in griptape/common/prompt_stack/contents/audio_message_content.py

```
@define
class AudioMessageContent(BaseMessageContent):
    artifact: AudioArtifact = field(metadata={"serializable": True})

    @classmethod
    def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪AudioMessageContent:
        audio_deltas = [delta for delta in deltas if isinstance(delta,␣
↪AudioDeltaMessageContent)]
        audio_data = [delta.data for delta in audio_deltas if delta.data is not None]
        transcript_data = [delta.transcript for delta in audio_deltas if delta.
↪transcript is not None]
        expires_at = next(delta.expires_at for delta in audio_deltas if delta.expires_
↪at is not None)
        audio_id = next(delta.id for delta in audio_deltas if delta.id is not None)

        audio_transcript = "".join(data for data in transcript_data)

        artifact = AudioArtifact(
            value=b"".join(base64.b64decode(data) for data in audio_data),
            format="wav",
            meta={
                "audio_id": audio_id,
                "expires_at": expires_at,
                "transcript": audio_transcript,
            },
```

(continues on next page)

(continued from previous page)

```
        )

        return cls(artifact=artifact)
```

- `artifact = field(metadata={'serializable': True})` class-attribute instance-attribute

from_deltas(deltas)classmethod

Source Code in griptape/common/prompt_stack/contents/audio_message_content.py

```
@classmethod
def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪AudioMessageContent:
    audio_deltas = [delta for delta in deltas if isinstance(delta,␣
↪AudioDeltaMessageContent)]
    audio_data = [delta.data for delta in audio_deltas if delta.data is not None]
    transcript_data = [delta.transcript for delta in audio_deltas if delta.transcript␣
↪is not None]
    expires_at = next(delta.expires_at for delta in audio_deltas if delta.expires_at␣
↪is not None)
    audio_id = next(delta.id for delta in audio_deltas if delta.id is not None)

    audio_transcript = "".join(data for data in transcript_data)

    artifact = AudioArtifact(
        value=b"".join(base64.b64decode(data) for data in audio_data),
        format="wav",
        meta={
            "audio_id": audio_id,
            "expires_at": expires_at,
            "transcript": audio_transcript,
        },
    )

    return cls(artifact=artifact)
```

BaseAction

Bases: *SerializableMixin* “) , ABC

Source Code in griptape/common/actions/base_action.py

```
class BaseAction(SerializableMixin, ABC): ...
```

BaseDeltaMessageContent

Bases:

ABC, *SerializableMixin* “)

Source Code in griptape/common/prompt_stack/contents/base_delta_message_content.py

```
@define
class BaseDeltaMessageContent(ABC, SerializableMixin):
    index: int = field(kw_only=True, default=0, metadata={"serializable": True})
```

- `index = field(kw_only=True, default=0, metadata={'serializable': True})` class-attribute instance-attribute

BaseMessage

Bases:

ABC, *SerializableMixin* “)

Source Code in griptape/common/prompt_stack/messages/base_message.py

```
@define
class BaseMessage(ABC, SerializableMixin):
    @define
    class Usage(SerializableMixin):
        input_tokens: Optional[float] = field(kw_only=True, default=None, metadata={
↪"serializable": True})
        output_tokens: Optional[float] = field(kw_only=True, default=None, metadata={
↪"serializable": True})

        @property
        def total_tokens(self) -> float:
            return (self.input_tokens or 0) + (self.output_tokens or 0)

        def __add__(self, other: BaseMessage.Usage) -> BaseMessage.Usage:
            return BaseMessage.Usage(
                input_tokens=(self.input_tokens or 0) + (other.input_tokens or 0),
                output_tokens=(self.output_tokens or 0) + (other.output_tokens or 0),
            )

    USER_ROLE = "user"
    ASSISTANT_ROLE = "assistant"
    SYSTEM_ROLE = "system"

    content: list[Union[BaseMessageContent, BaseDeltaMessageContent]] =␣
↪field(metadata={"serializable": True})
    role: str = field(kw_only=True, metadata={"serializable": True})
    usage: Usage = field(kw_only=True, default=Factory(lambda: BaseMessage.Usage()),␣
↪metadata={"serializable": True})

    def is_system(self) -> bool:
        return self.role == self.SYSTEM_ROLE

    def is_user(self) -> bool:
        return self.role == self.USER_ROLE

    def is_assistant(self) -> bool:
        return self.role == self.ASSISTANT_ROLE
```

- `ASSISTANT_ROLE = 'assistant'` class-attribute instance-attribute

- `SYSTEM_ROLE = 'system'` class-attribute instance-attribute
- `USER_ROLE = 'user'` class-attribute instance-attribute
- `content = field(metadata={'serializable': True})` class-attribute instance-attribute
- `role = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `usage = field(kw_only=True, default=Factory(lambda: BaseMessage.Usage()), metadata={'serializable': True})` class-attribute instance-attribute

Usage

Bases: *SerializableMixin* “)

Source Code in griptape/common/prompt_stack/messages/base_message.py

```
@define
class Usage(SerializableMixin):
    input_tokens: Optional[float] = field(kw_only=True, default=None, metadata={
↪"serializable": True})
    output_tokens: Optional[float] = field(kw_only=True, default=None, metadata={
↪"serializable": True})

    @property
    def total_tokens(self) -> float:
        return (self.input_tokens or 0) + (self.output_tokens or 0)

    def __add__(self, other: BaseMessage.Usage) -> BaseMessage.Usage:
        return BaseMessage.Usage(
            input_tokens=(self.input_tokens or 0) + (other.input_tokens or 0),
            output_tokens=(self.output_tokens or 0) + (other.output_tokens or 0),
        )
```

- `input_tokens = field(kw_only=True, default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `output_tokens = field(kw_only=True, default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `total_tokens` property

is_assistant()

Source Code in griptape/common/prompt_stack/messages/base_message.py

```
def is_assistant(self) -> bool:
    return self.role == self.ASSISTANT_ROLE
```

is_system()

Source Code in griptape/common/prompt_stack/messages/base_message.py

```
def is_system(self) -> bool:
    return self.role == self.SYSTEM_ROLE
```

is_user()

Source Code in griptape/common/prompt_stack/messages/base_message.py

```
def is_user(self) -> bool:
    return self.role == self.USER_ROLE
```

BaseMessageContent

Bases:

ABC, *SerializableMixin* “)

Source Code in griptape/common/prompt_stack/contents/base_message_content.py

```
@define
class BaseMessageContent(ABC, SerializableMixin):
    artifact: BaseArtifact = field(metadata={"serializable": True})

    def __str__(self) -> str:
        return self.artifact.to_text()

    def __bool__(self) -> bool:
        return bool(self.artifact)

    def __len__(self) -> int:
        return len(self.artifact)

    def to_text(self) -> str:
        return str(self.artifact)

    @classmethod
    @abstractmethod
    def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪BaseMessageContent: ...
```

- `artifact = field(metadata={'serializable': True})` class-attribute instance-attribute

bool()

Source Code in griptape/common/prompt_stack/contents/base_message_content.py

```
def __bool__(self) -> bool:
    return bool(self.artifact)
```

len()

Source Code in griptape/common/prompt_stack/contents/base_message_content.py

```
def __len__(self) -> int:
    return len(self.artifact)
```

str()

Source Code in griptape/common/prompt_stack/contents/base_message_content.py

```
def __str__(self) -> str:
    return self.artifact.to_text()
```

from_deltas(deltas)abstractmethodclassmethod

Source Code in griptape/common/prompt_stack/contents/base_message_content.py

```
@classmethod
@abstractmethod
def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪BaseMessageContent: ...
```

to_text()

Source Code in griptape/common/prompt_stack/contents/base_message_content.py

```
def to_text(self) -> str:
    return str(self.artifact)
```

DeltaMessage

Bases: *BaseMessage* “)

Source Code in griptape/common/prompt_stack/messages/delta_message.py

```
@define
class DeltaMessage(BaseMessage):
    role: Optional[str] = field(kw_only=True, default=None, metadata={"serializable":␣
↪True})
    content: Optional[BaseDeltaMessageContent] = field(kw_only=True, default=None,␣
↪metadata={"serializable": True})
```

- `content = field(kw_only=True, default=None, metadata={'serializable': True})`
  class-attribute instance-attribute
- `role = field(kw_only=True, default=None, metadata={'serializable': True})`
  class-attribute instance-attribute

GenericMessageContent

Bases: *BaseMessageContent* “)

Source Code in griptape/common/prompt_stack/contents/generic_message_content.py

```
@define
class GenericMessageContent(BaseMessageContent):
    artifact: GenericArtifact = field(metadata={"serializable": True})

    @classmethod
    def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪GenericMessageContent:
        raise NotImplementedError
```

- `artifact = field(metadata={'serializable': True})` class-attribute instance-attribute

from_deltas(deltas)classmethod

Source Code in griptape/common/prompt_stack/contents/generic_message_content.py

```
@classmethod
def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪GenericMessageContent:
    raise NotImplementedError
```

ImageMessageContent

Bases: *BaseMessageContent* “)

Source Code in griptape/common/prompt_stack/contents/image_message_content.py

```
@define
class ImageMessageContent(BaseMessageContent):
    artifact: Union[ImageArtifact, ImageUrlArtifact] = field(metadata={"serializable
↪": True})

    @classmethod
    def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪ImageMessageContent:
        raise NotImplementedError
```

- `artifact = field(metadata={'serializable': True})` class-attribute instance-attribute

from_deltas(deltas)classmethod

Source Code in griptape/common/prompt_stack/contents/image_message_content.py

```
@classmethod
def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪ImageMessageContent:
    raise NotImplementedError
```

Message

Bases: *BaseMessage* “)

Source Code in griptape/common/prompt_stack/messages/message.py

```
@define
class Message(BaseMessage):
    def __init__(self, content: str | list[BaseMessageContent], **kwargs: Any) ->␣
↪None:
        if isinstance(content, str):
            content = [TextMessageContent(TextArtifact(value=content))]
        self.__attrs_init__(content, **kwargs)  # pyright:␣
↪ignore[reportAttributeAccessIssue]

    content: list[BaseMessageContent] = field(metadata={"serializable": True})
```

(continues on next page)

(continued from previous page)

```
    @property
    def value(self) -> Any:
        return self.to_artifact().value

    def __str__(self) -> str:
        return self.to_text()

    def has_all_content_type(self, content_type: type[T]) -> bool:
        return all(isinstance(content, content_type) for content in self.content)

    def has_any_content_type(self, content_type: type[T]) -> bool:
        return any(isinstance(content, content_type) for content in self.content)

    def get_content_type(self, content_type: type[T]) -> list[T]:
        return [content for content in self.content if isinstance(content, content_
↪type)]

    def is_text(self) -> bool:
        return all(isinstance(content, TextMessageContent) for content in self.
↪content)

    def to_text(self) -> str:
        return "".join(
            [content.artifact.to_text() for content in self.content if␣
↪isinstance(content, TextMessageContent)],
        )

    def to_artifact(self, meta: Optional[dict] = None) -> BaseArtifact:
        if meta is None:
            meta = {}
        if len(self.content) == 1:
            artifact = self.content[0].artifact
        else:
            artifact = ListArtifact([content.artifact for content in self.content])

        artifact.meta.update(meta)
        return artifact
```

- `content = field(metadata={'serializable': True})` class-attribute instance-attribute
- `value` property

init(content, **kwargs)

Source Code in griptape/common/prompt_stack/messages/message.py

```
def __init__(self, content: str | list[BaseMessageContent], **kwargs: Any) -> None:
    if isinstance(content, str):
        content = [TextMessageContent(TextArtifact(value=content))]
    self.__attrs_init__(content, **kwargs)  # pyright:␣
↪ignore[reportAttributeAccessIssue]
```

str()

Source Code in griptape/common/prompt_stack/messages/message.py

```
def __str__(self) -> str:
    return self.to_text()
```

get_content_type(content_type)

Source Code in griptape/common/prompt_stack/messages/message.py

```
def get_content_type(self, content_type: type[T]) -> list[T]:
    return [content for content in self.content if isinstance(content, content_type)]
```

has_all_content_type(content_type)

Source Code in griptape/common/prompt_stack/messages/message.py

```
def has_all_content_type(self, content_type: type[T]) -> bool:
    return all(isinstance(content, content_type) for content in self.content)
```

has_any_content_type(content_type)

Source Code in griptape/common/prompt_stack/messages/message.py

```
def has_any_content_type(self, content_type: type[T]) -> bool:
    return any(isinstance(content, content_type) for content in self.content)
```

is_text()

Source Code in griptape/common/prompt_stack/messages/message.py

```
def is_text(self) -> bool:
    return all(isinstance(content, TextMessageContent) for content in self.content)
```

to_artifact(meta=None)

Source Code in griptape/common/prompt_stack/messages/message.py

```
def to_artifact(self, meta: Optional[dict] = None) -> BaseArtifact:
    if meta is None:
        meta = {}
    if len(self.content) == 1:
        artifact = self.content[0].artifact
    else:
        artifact = ListArtifact([content.artifact for content in self.content])

    artifact.meta.update(meta)
    return artifact
```

to_text()

Source Code in griptape/common/prompt_stack/messages/message.py

```
def to_text(self) -> str:
    return "".join(
        [content.artifact.to_text() for content in self.content if isinstance(content,
↪ TextMessageContent)],
    )
```

Observable

Source Code in griptape/common/observable.py

```
class Observable:
    @define
    class Call:
        func: Callable = field(kw_only=True)
        instance: Optional[Any] = field(default=None, kw_only=True)
        args: tuple[Any, ...] = field(default=Factory(tuple), kw_only=True)
        kwargs: dict[str, Any] = field(default=Factory(dict), kw_only=True)
        decorator_args: tuple[Any, ...] = field(default=Factory(tuple), kw_only=True)
        decorator_kwargs: dict[str, Any] = field(default=Factory(dict), kw_only=True)

        def __call__(self) -> Any:
            return self.func(*self.args, **self.kwargs)

        @property
        def tags(self) -> Optional[list[str]]:
            return self.decorator_kwargs.get("tags")
```

Call

Source Code in griptape/common/observable.py

```
@define
class Call:
    func: Callable = field(kw_only=True)
    instance: Optional[Any] = field(default=None, kw_only=True)
    args: tuple[Any, ...] = field(default=Factory(tuple), kw_only=True)
    kwargs: dict[str, Any] = field(default=Factory(dict), kw_only=True)
    decorator_args: tuple[Any, ...] = field(default=Factory(tuple), kw_only=True)
    decorator_kwargs: dict[str, Any] = field(default=Factory(dict), kw_only=True)

    def __call__(self) -> Any:
        return self.func(*self.args, **self.kwargs)

    @property
    def tags(self) -> Optional[list[str]]:
        return self.decorator_kwargs.get("tags")
```

- `args = field(default=Factory(tuple), kw_only=True)` class-attribute instance-attribute
- `decorator_args = field(default=Factory(tuple), kw_only=True)` class-attribute instance-attribute

- `decorator_kwargs = field(default=Factory(dict), kw_only=True)` class-attribute instance-attribute
- `func = field(kw_only=True)` class-attribute instance-attribute
- `instance = field(default=None, kw_only=True)` class-attribute instance-attribute
- `kwargs = field(default=Factory(dict), kw_only=True)` class-attribute instance-attribute
- `tags` property

PromptStack

Bases: *SerializableMixin* “)

Source Code in griptape/common/prompt_stack/prompt_stack.py

```
@define
class PromptStack(SerializableMixin):
    messages: list[Message] = field(factory=list, kw_only=True, metadata={
↪"serializable": True})
    tools: list[BaseTool] = field(factory=list, kw_only=True)
    output_schema: Optional[Union[Schema, type[BaseModel]]] = field(default=None, kw_
↪only=True)

    @property
    def system_messages(self) -> list[Message]:
        return [message for message in self.messages if message.is_system()]

    @property
    def user_messages(self) -> list[Message]:
        return [message for message in self.messages if message.is_user()]

    @property
    def assistant_messages(self) -> list[Message]:
        return [message for message in self.messages if message.is_assistant()]

    def add_message(self, artifact: str | BaseArtifact, role: str) -> Message:
        content = self.__to_message_content(artifact)

        self.messages.append(Message(content=content, role=role))

        return self.messages[-1]

    def add_system_message(self, artifact: str | BaseArtifact) -> Message:
        return self.add_message(artifact, Message.SYSTEM_ROLE)

    def add_user_message(self, artifact: str | BaseArtifact) -> Message:
        return self.add_message(artifact, Message.USER_ROLE)

    def add_assistant_message(self, artifact: str | BaseArtifact) -> Message:
        return self.add_message(artifact, Message.ASSISTANT_ROLE)

    def to_output_json_schema(self, schema_id: str = "Output Format") -> dict:
        if self.output_schema is None:
            raise ValueError("Output schema is not set")
```

(continues on next page)

(continued from previous page)

```
        if isinstance(self.output_schema, Schema):
            json_schema = self.output_schema.json_schema(schema_id)
        elif isinstance(self.output_schema, type) and issubclass(self.output_schema,␣
↪BaseModel):
            json_schema = build_strict_schema(self.output_schema.model_json_schema(),␣
↪schema_id)
        else:
            raise ValueError(f"Unsupported output schema type: {type(self.output_
↪schema)}")

        return json_schema

    @classmethod
    def from_artifact(cls, artifact: BaseArtifact) -> PromptStack:
        prompt_stack = cls()
        prompt_stack.add_user_message(artifact)

        return prompt_stack

    def __to_message_content(self, artifact: str | BaseArtifact) ->␣
↪list[BaseMessageContent]:
        if isinstance(artifact, str):
            return [TextMessageContent(TextArtifact(artifact))]
        if isinstance(artifact, TextArtifact):
            return [TextMessageContent(artifact)]
        if isinstance(artifact, ImageArtifact):
            return [ImageMessageContent(artifact)]
        if isinstance(artifact, ImageUrlArtifact):
            return [ImageMessageContent(artifact)]
        if isinstance(artifact, AudioArtifact):
            return [AudioMessageContent(artifact)]
        if isinstance(artifact, GenericArtifact):
            return [GenericMessageContent(artifact)]
        if isinstance(artifact, ActionArtifact):
            action = artifact.value
            output = action.output
            if output is None:
                return [ActionCallMessageContent(artifact)]
            return [ActionResultMessageContent(output, action=action)]
        if isinstance(artifact, ListArtifact):
            processed_contents = [self.__to_message_content(artifact) for artifact in␣
↪artifact.value]

            return [sub_content for processed_content in processed_contents for sub_
↪content in processed_content]
        return [TextMessageContent(TextArtifact(artifact.to_text()))]
```

- `assistant_messages` property
- `messages = field(factory=list, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output_schema = field(default=None, kw_only=True)` class-attribute instance-attribute

- `system_messages` property
- `tools = field(factory=list, kw_only=True)` class-attribute instance-attribute
- `user_messages` property

__to_message_content(artifact)

Source Code in griptape/common/prompt_stack/prompt_stack.py

```
def __to_message_content(self, artifact: str | BaseArtifact) ->␣
↪list[BaseMessageContent]:
    if isinstance(artifact, str):
        return [TextMessageContent(TextArtifact(artifact))]
    if isinstance(artifact, TextArtifact):
        return [TextMessageContent(artifact)]
    if isinstance(artifact, ImageArtifact):
        return [ImageMessageContent(artifact)]
    if isinstance(artifact, ImageUrlArtifact):
        return [ImageMessageContent(artifact)]
    if isinstance(artifact, AudioArtifact):
        return [AudioMessageContent(artifact)]
    if isinstance(artifact, GenericArtifact):
        return [GenericMessageContent(artifact)]
    if isinstance(artifact, ActionArtifact):
        action = artifact.value
        output = action.output
        if output is None:
            return [ActionCallMessageContent(artifact)]
        return [ActionResultMessageContent(output, action=action)]
    if isinstance(artifact, ListArtifact):
        processed_contents = [self.__to_message_content(artifact) for artifact in␣
↪artifact.value]

        return [sub_content for processed_content in processed_contents for sub_
↪content in processed_content]
    return [TextMessageContent(TextArtifact(artifact.to_text()))]
```

add_assistant_message(artifact)

Source Code in griptape/common/prompt_stack/prompt_stack.py

```
def add_assistant_message(self, artifact: str | BaseArtifact) -> Message:
    return self.add_message(artifact, Message.ASSISTANT_ROLE)
```

add_message(artifact, role)

Source Code in griptape/common/prompt_stack/prompt_stack.py

```
def add_message(self, artifact: str | BaseArtifact, role: str) -> Message:
    content = self.__to_message_content(artifact)

    self.messages.append(Message(content=content, role=role))

    return self.messages[-1]
```

add_system_message(artifact)

Source Code in griptape/common/prompt_stack/prompt_stack.py

```
def add_system_message(self, artifact: str | BaseArtifact) -> Message:
    return self.add_message(artifact, Message.SYSTEM_ROLE)
```

add_user_message(artifact)

Source Code in griptape/common/prompt_stack/prompt_stack.py

```
def add_user_message(self, artifact: str | BaseArtifact) -> Message:
    return self.add_message(artifact, Message.USER_ROLE)
```

from_artifact(artifact)classmethod

Source Code in griptape/common/prompt_stack/prompt_stack.py

```
@classmethod
def from_artifact(cls, artifact: BaseArtifact) -> PromptStack:
    prompt_stack = cls()
    prompt_stack.add_user_message(artifact)

    return prompt_stack
```

to_output_json_schema(schema_id='Output Format')

Source Code in griptape/common/prompt_stack/prompt_stack.py

```
def to_output_json_schema(self, schema_id: str = "Output Format") -> dict:
    if self.output_schema is None:
        raise ValueError("Output schema is not set")

    if isinstance(self.output_schema, Schema):
        json_schema = self.output_schema.json_schema(schema_id)
    elif isinstance(self.output_schema, type) and issubclass(self.output_schema,␣
↪BaseModel):
        json_schema = build_strict_schema(self.output_schema.model_json_schema(),␣
↪schema_id)
    else:
        raise ValueError(f"Unsupported output schema type: {type(self.output_schema)}
↪")

    return json_schema
```

Reference

Bases: *SerializableMixin* “)

Source Code in griptape/common/reference.py

11.5.4 True})

- `authors = field(factory=list, metadata={'serializable': True})` class-attribute instance-attribute
- `id = field(default=Factory(lambda: uuid.uuid4().hex), metadata={'serializable': True})` class-attribute instance-attribute
- `source = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `title = field(metadata={'serializable': True})` class-attribute instance-attribute
- `url = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `year = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

Bases: *BaseDeltaMessageContent* “)

Source Code in griptape/common/prompt_stack/contents/text_delta_message_content.py

```
@define
class TextDeltaMessageContent(BaseDeltaMessageContent):
    text: str = field(metadata={"serializable": True})
```

- `text = field(metadata={'serializable': True})` class-attribute instance-attribute

Bases: *BaseMessageContent* “)

Source Code in griptape/common/prompt_stack/contents/text_message_content.py

```
@define
class TextMessageContent(BaseMessageContent):
    artifact: BaseArtifact = field(metadata={"serializable": True})

    @classmethod
    def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪TextMessageContent:
        text_deltas = [delta for delta in deltas if isinstance(delta,␣
↪TextDeltaMessageContent)]

        artifact = TextArtifact(value="".join(delta.text for delta in text_deltas))

        return cls(artifact=artifact)
```

- `artifact = field(metadata={'serializable': True})` class-attribute instance-attribute

from_deltas(deltas)classmethod

Source Code in griptape/common/prompt_stack/contents/text_message_content.py

```
@classmethod
def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) -> TextMessageContent:
    text_deltas = [delta for delta in deltas if isinstance(delta,␣
↪TextDeltaMessageContent)]

    artifact = TextArtifact(value="".join(delta.text for delta in text_deltas))
```

(continues on next page)

(continued from previous page)

```
    return cls(artifact=artifact)
```

ToolAction

Bases: *BaseAction* “)

Attributes

Name	Type	Description
<a href="./#griptape.common.ToolAction.tag" title="tag = field(metadata={'serializable': True}) class-attribute instance-attribute (griptape.common.ToolAction.tag)"><code class="language-python">	*str*	The tag (unique identifier) of the action.
<a href="./#griptape.common.ToolAction.name" title="name = field(metadata={'serializable': True}) class-attribute instance-attribute (griptape.common.ToolAction.name)"><code class="language-python">	*str*	The name (Tool name) of the action.
<a href="./#griptape.common.ToolAction.path" title="path = field(default=None, metadata={'serializable': True}) class-attribute instance-attribute (griptape.common.ToolAction.path)"><code class="language-python">	*Optional[str]*	The path (Tool activity name) of the action.

continues on next page

Table 32 – continued from previous page

Name	Type	Description
<a href="./#griptape.common.ToolAction.input" title="input = field(factory=dict, metadata={'serializable': True}) class-attribute instance-attribute (griptape.common.ToolAction.input)"><code class="language-python">	*dict*	The input (Tool params) of the action.
<a href="./#griptape.common.ToolAction.tool" title="tool = field(default=None) class-attribute instance-attribute (griptape.common.ToolAction.tool)"><code class="language-python">	<a href="../tools#griptape.tools.BaseTool" title="BaseTool (griptape.tools.BaseTool)"><code class="language-python">Optional[BaseTool]</code>	The matched Tool of the action.
<a href="./#griptape.common.ToolAction.output" title="output = field(default=None) class-attribute instance-attribute (griptape.common.ToolAction.output)"><code class="language-python">	<a href="../artifacts#griptape.artifacts.BaseArtifact" title="BaseArtifact (griptape.artifacts.BaseArtifact)"><code class="language-python">Optional[BaseArtifact]</cod	The output (Tool result) of the action.

Source Code in griptape/common/actions/tool_action.py

```
@define(kw_only=True)
class ToolAction(BaseAction):
    """Represents an instance of an LLM using a Tool.

    Attributes:
        tag: The tag (unique identifier) of the action.
        name: The name (Tool name) of the action.
        path: The path (Tool activity name) of the action.
        input: The input (Tool params) of the action.
        tool: The matched Tool of the action.
        output: The output (Tool result) of the action.
    """

    tag: str = field(metadata={"serializable": True})
    name: str = field(metadata={"serializable": True})
    path: Optional[str] = field(default=None, metadata={"serializable": True})
    input: dict = field(factory=dict, metadata={"serializable": True})
```

(continues on next page)

(continued from previous page)

```
    tool: Optional[BaseTool] = field(default=None)
    output: Optional[BaseArtifact] = field(default=None)

    def __str__(self) -> str:
        return json.dumps(self.to_dict())

    def to_native_tool_name(self) -> str:
        parts = [self.name]

        if self.path is not None:
            parts.append(self.path)

        return "_".join(parts)

    @classmethod
    def from_native_tool_name(cls, native_tool_name: str) -> tuple[str,␣
↪Optional[str]]:
        parts = native_tool_name.split("_", 1)

        if len(parts) == 1:
            name, path = parts[0], None
        else:
            name, path = parts

        return name, path
```

- `input = field(factory=dict, metadata={'serializable': True})` class-attribute instance-attribute
- `name = field(metadata={'serializable': True})` class-attribute instance-attribute
- `output = field(default=None)` class-attribute instance-attribute
- `path = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `tag = field(metadata={'serializable': True})` class-attribute instance-attribute
- `tool = field(default=None)` class-attribute instance-attribute

str()

Source Code in griptape/common/actions/tool_action.py

```
def __str__(self) -> str:
    return json.dumps(self.to_dict())
```

from_native_tool_name(native_tool_name)classmethod

Source Code in griptape/common/actions/tool_action.py

```
@classmethod
def from_native_tool_name(cls, native_tool_name: str) -> tuple[str, Optional[str]]:
    parts = native_tool_name.split("_", 1)
```

(continues on next page)

(continued from previous page)

```
    if len(parts) == 1:
        name, path = parts[0], None
    else:
        name, path = parts

    return name, path
```

to_native_tool_name()

Source Code in griptape/common/actions/tool_action.py

```
def to_native_tool_name(self) -> str:
    parts = [self.name]

    if self.path is not None:
        parts.append(self.path)

    return "_".join(parts)
```

actions

base_action

Bases: *SerializableMixin* “) , ABC

Source Code in griptape/common/actions/base_action.py

```
class BaseAction(SerializableMixin, ABC): ...
```

tool_action

Bases: *BaseAction* “)

Attributes

Name	Type	Description
<a href="tool_action.mdx#griptape.common.actions.tool_action.ToolAction.tag" title="tag = field(metadata={'serializable': True}) class-attribute instance-attribute (griptape.common.actions.tool_action.ToolAction.tag)"><code class="language-python">	*str*	The tag (unique identifier) of the action.

continues on next page

Table 33 – continued from previous page

Name	Type	Description
<a href="tool_action.mdx#griptape.common.actions.tool_action.ToolAction.name" title="name = field(metadata={'serializable': True}) class-attribute instance-attribute (griptape.common.actions.tool_action.ToolAction.name)"><code class="language-python">	*str*	The name (Tool name) of the action.
<a href="tool_action.mdx#griptape.common.actions.tool_action.ToolAction.path" title="path = field(default=None, metadata={'serializable': True}) class-attribute instance-attribute (griptape.common.actions.tool_action.ToolAction.path)"><code class="language-python">	*Optional[str]*	The path (Tool activity name) of the action.
<a href="tool_action.mdx#griptape.common.actions.tool_action.ToolAction.input" title="input = field(factory=dict, metadata={'serializable': True}) class-attribute instance-attribute (griptape.common.actions.tool_action.ToolAction.input)"><code class="language-python">	*dict*	The input (Tool params) of the action.

continues on next page

Table 33 – continued from previous page

Name	Type	Description
<a href="tool_action.mdx#griptape.common.actions.tool_action.ToolAction.tool" title="tool = field(default=None) class-attribute instance-attribute (griptape.common.actions.tool_action.ToolAction.tool)"><code class="language-python">	<a href="../../tools#griptape.tools.BaseTool" title="BaseTool (griptape.tools.BaseTool)"><code class="language-python">Optional[BaseTool]</code>	The matched Tool of the action.
<a href="tool_action.mdx#griptape.common.actions.tool_action.ToolAction.output" title="output = field(default=None) class-attribute instance-attribute (griptape.common.actions.tool_action.ToolAction.output)"><code class="language-python">	<a href="../../artifacts#griptape.artifacts.BaseArtifact" title="BaseArtifact (griptape.artifacts.BaseArtifact)"><code class="language-python">Optional[BaseArtifact]</cod	The output (Tool result) of the action.

Source Code in griptape/common/actions/tool_action.py

```
@define(kw_only=True)
class ToolAction(BaseAction):
    """Represents an instance of an LLM using a Tool.

    Attributes:
        tag: The tag (unique identifier) of the action.
        name: The name (Tool name) of the action.
        path: The path (Tool activity name) of the action.
        input: The input (Tool params) of the action.
        tool: The matched Tool of the action.
        output: The output (Tool result) of the action.
    """

    tag: str = field(metadata={"serializable": True})
    name: str = field(metadata={"serializable": True})
    path: Optional[str] = field(default=None, metadata={"serializable": True})
    input: dict = field(factory=dict, metadata={"serializable": True})
    tool: Optional[BaseTool] = field(default=None)
    output: Optional[BaseArtifact] = field(default=None)
```

(continues on next page)

(continued from previous page)

```
    def __str__(self) -> str:
        return json.dumps(self.to_dict())

    def to_native_tool_name(self) -> str:
        parts = [self.name]

        if self.path is not None:
            parts.append(self.path)

        return "_".join(parts)

    @classmethod
    def from_native_tool_name(cls, native_tool_name: str) -> tuple[str,␣
↪Optional[str]]:
        parts = native_tool_name.split("_", 1)

        if len(parts) == 1:
            name, path = parts[0], None
        else:
            name, path = parts

        return name, path
```

- `input = field(factory=dict, metadata={'serializable': True})` class-attribute instance-attribute
- `name = field(metadata={'serializable': True})` class-attribute instance-attribute
- `output = field(default=None)` class-attribute instance-attribute
- `path = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `tag = field(metadata={'serializable': True})` class-attribute instance-attribute
- `tool = field(default=None)` class-attribute instance-attribute

str()

Source Code in griptape/common/actions/tool_action.py

```
def __str__(self) -> str:
    return json.dumps(self.to_dict())
```

from_native_tool_name(native_tool_name)classmethod

Source Code in griptape/common/actions/tool_action.py

```
@classmethod
def from_native_tool_name(cls, native_tool_name: str) -> tuple[str, Optional[str]]:
    parts = native_tool_name.split("_", 1)

    if len(parts) == 1:
```

(continues on next page)

(continued from previous page)

```
        name, path = parts[0], None
    else:
        name, path = parts

    return name, path
```

to_native_tool_name()

Source Code in griptape/common/actions/tool_action.py

```
def to_native_tool_name(self) -> str:
    parts = [self.name]

    if self.path is not None:
        parts.append(self.path)

    return "_".join(parts)
```

decorators

- `P = ParamSpec('P')` module-attribute
- `T = TypeVar('T')` module-attribute

Source Code in griptape/common/decorators.py

```
def observable(wrapped: Optional[Callable[P, T] | Any] = None, **dkwargs: Any) -> Any:
    if wrapped is None:
        return functools.partial(observable, **dkwargs)

    if not callable(wrapped):
        raise ValueError("Non-callable positional argument passed. Use kwargs to pass␣
↪arguments to the decorator.")

    @wrapt.decorator
    def wrapper(wrapped: Callable[P, T], instance: Any, args: Any, kwargs: Any) -> T:
        from griptape.common.observable import Observable
        from griptape.observability.observability import Observability

        return cast(
            "T",
            Observability.observe(
                Observable.Call(
                    func=wrapped,
                    instance=instance,
                    args=args,
                    kwargs=kwargs,
                    decorator_args=(),
                    decorator_kwargs=dkwargs,
                )
            ),
        )
```

(continues on next page)

(continued from previous page)

```
    return wrapper(wrapped)  # pyright: ignore[reportCallIssue]
```

observable

Source Code in griptape/common/observable.py

```
class Observable:
    @define
    class Call:
        func: Callable = field(kw_only=True)
        instance: Optional[Any] = field(default=None, kw_only=True)
        args: tuple[Any, ...] = field(default=Factory(tuple), kw_only=True)
        kwargs: dict[str, Any] = field(default=Factory(dict), kw_only=True)
        decorator_args: tuple[Any, ...] = field(default=Factory(tuple), kw_only=True)
        decorator_kwargs: dict[str, Any] = field(default=Factory(dict), kw_only=True)

        def __call__(self) -> Any:
            return self.func(*self.args, **self.kwargs)

        @property
        def tags(self) -> Optional[list[str]]:
            return self.decorator_kwargs.get("tags")
```

Call

Source Code in griptape/common/observable.py

```
@define
class Call:
    func: Callable = field(kw_only=True)
    instance: Optional[Any] = field(default=None, kw_only=True)
    args: tuple[Any, ...] = field(default=Factory(tuple), kw_only=True)
    kwargs: dict[str, Any] = field(default=Factory(dict), kw_only=True)
    decorator_args: tuple[Any, ...] = field(default=Factory(tuple), kw_only=True)
    decorator_kwargs: dict[str, Any] = field(default=Factory(dict), kw_only=True)

    def __call__(self) -> Any:
        return self.func(*self.args, **self.kwargs)

    @property
    def tags(self) -> Optional[list[str]]:
        return self.decorator_kwargs.get("tags")
```

- `args = field(default=Factory(tuple), kw_only=True)` class-attribute instance-attribute
- `decorator_args = field(default=Factory(tuple), kw_only=True)` class-attribute instance-attribute
- `decorator_kwargs = field(default=Factory(dict), kw_only=True)` class-attribute instance-attribute
- `func = field(kw_only=True)` class-attribute instance-attribute

- `instance = field(default=None, kw_only=True)` class-attribute instance-attribute
- `kwargs = field(default=Factory(dict), kw_only=True)` class-attribute instance-attribute
- `tags` property

prompt_stack

contents

action_call_delta_message_content

Bases: *BaseDeltaMessageContent* “)

Source Code in griptape/common/prompt_stack/contents/action_call_delta_message_content.py

```
@define
class ActionCallDeltaMessageContent(BaseDeltaMessageContent):
    tag: Optional[str] = field(default=None, metadata={"serializable": True})
    name: Optional[str] = field(default=None, metadata={"serializable": True})
    path: Optional[str] = field(default=None, metadata={"serializable": True})
    partial_input: Optional[str] = field(default=None, metadata={"serializable": True}
↪)

    def __str__(self) -> str:
        parts = []

        if self.name:
            parts.append(self.name)
            if self.path:
                parts.append(f".{self.path}")
                if self.tag:
                    parts.append(f" ({self.tag})")

        if self.partial_input:
            if parts:
                parts.append(f" {self.partial_input}")
            else:
                parts.append(self.partial_input)

        return "".join(parts)
```

- `name = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `partial_input = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `path = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `tag = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

str()

Source Code in griptape/common/prompt_stack/contents/action_call_delta_message_content.py

```
def __str__(self) -> str:
    parts = []

    if self.name:
        parts.append(self.name)
        if self.path:
            parts.append(f".{self.path}")
            if self.tag:
                parts.append(f" ({self.tag})")

    if self.partial_input:
        if parts:
            parts.append(f" {self.partial_input}")
        else:
            parts.append(self.partial_input)

    return "".join(parts)
```

action_call_message_content

Bases: *BaseMessageContent* “)

Source Code in griptape/common/prompt_stack/contents/action_call_message_content.py

```
@define
class ActionCallMessageContent(BaseMessageContent):
    artifact: ActionArtifact = field(metadata={"serializable": True})

    @classmethod
    def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪ActionCallMessageContent:
        action_call_deltas = [delta for delta in deltas if isinstance(delta,␣
↪ActionCallDeltaMessageContent)]

        tag = None
        name = None
        path = None
        json_input = ""

        for delta in action_call_deltas:
            if delta.tag is not None:
                tag = delta.tag
            if delta.name is not None:
                name = delta.name
            if delta.path is not None:
                path = delta.path
            if delta.partial_input is not None:
                json_input += delta.partial_input

        if tag is not None and name is not None and path is not None:
```

(continues on next page)

(continued from previous page)

```
            try:
                parsed_input = json.loads(json_input or "{}")
            except json.JSONDecodeError as exc:
                raise ValueError("Invalid JSON input for ToolAction") from exc
            action = ToolAction(tag=tag, name=name, path=path, input=parsed_input)
        else:
            raise ValueError("Missing required fields for ToolAction")

        artifact = ActionArtifact(value=action)

        return cls(artifact=artifact)
```

- `artifact = field(metadata={'serializable': True})` class-attribute instance-attribute

from_deltas(deltas)classmethod

Source Code in griptape/common/prompt_stack/contents/action_call_message_content.py

```
@classmethod
def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪ActionCallMessageContent:
    action_call_deltas = [delta for delta in deltas if isinstance(delta,␣
↪ActionCallDeltaMessageContent)]

    tag = None
    name = None
    path = None
    json_input = ""

    for delta in action_call_deltas:
        if delta.tag is not None:
            tag = delta.tag
        if delta.name is not None:
            name = delta.name
        if delta.path is not None:
            path = delta.path
        if delta.partial_input is not None:
            json_input += delta.partial_input

    if tag is not None and name is not None and path is not None:
        try:
            parsed_input = json.loads(json_input or "{}")
        except json.JSONDecodeError as exc:
            raise ValueError("Invalid JSON input for ToolAction") from exc
        action = ToolAction(tag=tag, name=name, path=path, input=parsed_input)
    else:
        raise ValueError("Missing required fields for ToolAction")

    artifact = ActionArtifact(value=action)

    return cls(artifact=artifact)
```

action_result_message_content

Bases: *BaseMessageContent* “)

Source Code in griptape/common/prompt_stack/contents/action_result_message_content.py

```
@define
class ActionResultMessageContent(BaseMessageContent):
    artifact: BaseArtifact = field(metadata={"serializable": True})
    action: ToolAction = field(metadata={"serializable": True})

    @classmethod
    def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪ActionResultMessageContent:
        raise NotImplementedError
```

- `action = field(metadata={'serializable': True})` class-attribute instance-attribute
- `artifact = field(metadata={'serializable': True})` class-attribute instance-attribute

from_deltas(deltas)classmethod

Source Code in griptape/common/prompt_stack/contents/action_result_message_content.py

```
@classmethod
def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪ActionResultMessageContent:
    raise NotImplementedError
```

audio_delta_message_content

Bases: *BaseDeltaMessageContent* “)

Attributes

Name	Type	Description
<a href="audio_delta_message_content.mdx#griptape.common.prompt_stack.contents.audio_delta_message_content.AudioDeltaMessageContent.id" title="id = field(default=None, metadata={'serializable': True}) class-attribute instance-attribute (griptape.common.prompt_stack.contents.audio_delta_message_content.AudioDeltaMessageContent.id)"><code class="language-python">	*Optional[str]*	The ID of the audio data.
<a href="audio_delta_message_content.mdx#griptape.common.prompt_stack.contents.audio_delta_message_content.AudioDeltaMessageContent.data" title="data = field(default=None, metadata={'serializable': True}) class-attribute instance-attribute (griptape.common.prompt_stack.contents.audio_delta_message_content.AudioDeltaMessageContent.data)"><code class="language-python">	*Optional[str]*	Base64 encoded audio data.

continues on next page

Table 34 – continued from previous page

Name	Type	Description
<a href="audio_delta_message_content.mdx#griptape.common.prompt_stack.contents.audio_delta_message_content.AudioDeltaMessageContent.transcript" title="transcript = field(default=None, metadata={'serializable': True})">class-attribute instance-attribute (griptape.common.prompt_stack.contents.audio_delta_message_content.AudioDeltaMessageContent.transcript)"><code class="language-python">	*Optional[str]*	The transcript of the audio data.
<a href="audio_delta_message_content.mdx#griptape.common.prompt_stack.contents.audio_delta_message_content.AudioDeltaMessageContent.expires_at" title="expires_at = field(default=None, metadata={'serializable': True})">class-attribute instance-attribute (griptape.common.prompt_stack.contents.audio_delta_message_content.AudioDeltaMessageContent.expires_at)"><code class="language-python">	*Optional[int]*	The Unix timestamp (in seconds) for when this audio data will no longer be accessible.

Source Code in griptape/common/prompt_stack/contents/audio_delta_message_content.py

```
@define(kw_only=True)
class AudioDeltaMessageContent(BaseDeltaMessageContent):
    """A delta message content for audio data.
```

(continues on next page)

(continued from previous page)

```
    Attributes:
        id: The ID of the audio data.
        data: Base64 encoded audio data.
        transcript: The transcript of the audio data.
        expires_at: The Unix timestamp (in seconds) for when this audio data will no␣
↪longer be accessible.
    """

    id: Optional[str] = field(default=None, metadata={"serializable": True})
    data: Optional[str] = field(default=None, metadata={"serializable": True})
    transcript: Optional[str] = field(default=None, metadata={"serializable": True})
    expires_at: Optional[int] = field(default=None, metadata={"serializable": True})
```

- `data = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `expires_at = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `id = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `transcript = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

audio_message_content

Bases: *BaseMessageContent* “)

Source Code in griptape/common/prompt_stack/contents/audio_message_content.py

```
@define
class AudioMessageContent(BaseMessageContent):
    artifact: AudioArtifact = field(metadata={"serializable": True})

    @classmethod
    def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪AudioMessageContent:
        audio_deltas = [delta for delta in deltas if isinstance(delta,␣
↪AudioDeltaMessageContent)]
        audio_data = [delta.data for delta in audio_deltas if delta.data is not None]
        transcript_data = [delta.transcript for delta in audio_deltas if delta.
↪transcript is not None]
        expires_at = next(delta.expires_at for delta in audio_deltas if delta.expires_
↪at is not None)
        audio_id = next(delta.id for delta in audio_deltas if delta.id is not None)

        audio_transcript = "".join(data for data in transcript_data)

        artifact = AudioArtifact(
            value=b"".join(base64.b64decode(data) for data in audio_data),
            format="wav",
            meta={
                "audio_id": audio_id,
```

(continues on next page)

(continued from previous page)

```
                "expires_at": expires_at,
                "transcript": audio_transcript,
            },
        )

        return cls(artifact=artifact)
```

- `artifact = field(metadata={'serializable': True})` class-attribute instance-attribute

from_deltas(deltas)classmethod

Source Code in griptape/common/prompt_stack/contents/audio_message_content.py

```
@classmethod
def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪AudioMessageContent:
    audio_deltas = [delta for delta in deltas if isinstance(delta,␣
↪AudioDeltaMessageContent)]
    audio_data = [delta.data for delta in audio_deltas if delta.data is not None]
    transcript_data = [delta.transcript for delta in audio_deltas if delta.transcript␣
↪is not None]
    expires_at = next(delta.expires_at for delta in audio_deltas if delta.expires_at␣
↪is not None)
    audio_id = next(delta.id for delta in audio_deltas if delta.id is not None)

    audio_transcript = "".join(data for data in transcript_data)

    artifact = AudioArtifact(
        value=b"".join(base64.b64decode(data) for data in audio_data),
        format="wav",
        meta={
            "audio_id": audio_id,
            "expires_at": expires_at,
            "transcript": audio_transcript,
        },
    )

    return cls(artifact=artifact)
```

base_delta_message_content

Bases:

ABC, *SerializableMixin* “)

Source Code in griptape/common/prompt_stack/contents/base_delta_message_content.py

```
@define
class BaseDeltaMessageContent(ABC, SerializableMixin):
    index: int = field(kw_only=True, default=0, metadata={"serializable": True})
```

- `index = field(kw_only=True, default=0, metadata={'serializable': True})` class-attribute instance-attribute

base_message_content

Bases:

ABC, *SerializableMixin* “)

Source Code in griptape/common/prompt_stack/contents/base_message_content.py

```
@define
class BaseMessageContent(ABC, SerializableMixin):
    artifact: BaseArtifact = field(metadata={"serializable": True})

    def __str__(self) -> str:
        return self.artifact.to_text()

    def __bool__(self) -> bool:
        return bool(self.artifact)

    def __len__(self) -> int:
        return len(self.artifact)

    def to_text(self) -> str:
        return str(self.artifact)

    @classmethod
    @abstractmethod
    def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪BaseMessageContent: ...
```

- `artifact = field(metadata={'serializable': True})` class-attribute instance-attribute

bool()

Source Code in griptape/common/prompt_stack/contents/base_message_content.py

```
def __bool__(self) -> bool:
    return bool(self.artifact)
```

len()

Source Code in griptape/common/prompt_stack/contents/base_message_content.py

```
def __len__(self) -> int:
    return len(self.artifact)
```

str()

Source Code in griptape/common/prompt_stack/contents/base_message_content.py

```
def __str__(self) -> str:
    return self.artifact.to_text()
```

from_deltas(deltas)abstractmethodclassmethod

Source Code in griptape/common/prompt_stack/contents/base_message_content.py

```
@classmethod
@abstractmethod
def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪BaseMessageContent: ...
```

to_text()

Source Code in griptape/common/prompt_stack/contents/base_message_content.py

```
def to_text(self) -> str:
    return str(self.artifact)
```

generic_message_content

Bases: *BaseMessageContent* “)

Source Code in griptape/common/prompt_stack/contents/generic_message_content.py

```
@define
class GenericMessageContent(BaseMessageContent):
    artifact: GenericArtifact = field(metadata={"serializable": True})

    @classmethod
    def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪GenericMessageContent:
        raise NotImplementedError
```

- `artifact = field(metadata={'serializable': True})` class-attribute instance-attribute

from_deltas(deltas)classmethod

Source Code in griptape/common/prompt_stack/contents/generic_message_content.py

```
@classmethod
def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪GenericMessageContent:
    raise NotImplementedError
```

image_message_content

Bases: *BaseMessageContent* “)

Source Code in griptape/common/prompt_stack/contents/image_message_content.py

```
@define
class ImageMessageContent(BaseMessageContent):
    artifact: Union[ImageArtifact, ImageUrlArtifact] = field(metadata={"serializable
↪": True})

    @classmethod
```

(continues on next page)

(continued from previous page)

```
    def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪ImageMessageContent:
        raise NotImplementedError
```

- `artifact = field(metadata={'serializable': True})` class-attribute instance-attribute

from_deltas(deltas)classmethod

Source Code in griptape/common/prompt_stack/contents/image_message_content.py

```
@classmethod
def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪ImageMessageContent:
    raise NotImplementedError
```

text_delta_message_content

Bases: *BaseDeltaMessageContent* “)

Source Code in griptape/common/prompt_stack/contents/text_delta_message_content.py

```
@define
class TextDeltaMessageContent(BaseDeltaMessageContent):
    text: str = field(metadata={"serializable": True})
```

- `text = field(metadata={'serializable': True})` class-attribute instance-attribute

text_message_content

Bases: *BaseMessageContent* “)

Source Code in griptape/common/prompt_stack/contents/text_message_content.py

```
@define
class TextMessageContent(BaseMessageContent):
    artifact: BaseArtifact = field(metadata={"serializable": True})

    @classmethod
    def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) ->␣
↪TextMessageContent:
        text_deltas = [delta for delta in deltas if isinstance(delta,␣
↪TextDeltaMessageContent)]

        artifact = TextArtifact(value="".join(delta.text for delta in text_deltas))

        return cls(artifact=artifact)
```

- `artifact = field(metadata={'serializable': True})` class-attribute instance-attribute

from_deltas(deltas)classmethod

Source Code in griptape/common/prompt_stack/contents/text_message_content.py

```
@classmethod
def from_deltas(cls, deltas: Sequence[BaseDeltaMessageContent]) -> TextMessageContent:
    text_deltas = [delta for delta in deltas if isinstance(delta,␣
↪TextDeltaMessageContent)]

    artifact = TextArtifact(value="".join(delta.text for delta in text_deltas))

    return cls(artifact=artifact)
```

messages

base_message

Bases:

ABC, *SerializableMixin* “)

Source Code in griptape/common/prompt_stack/messages/base_message.py

```
@define
class BaseMessage(ABC, SerializableMixin):
    @define
    class Usage(SerializableMixin):
        input_tokens: Optional[float] = field(kw_only=True, default=None, metadata={
↪"serializable": True})
        output_tokens: Optional[float] = field(kw_only=True, default=None, metadata={
↪"serializable": True})

        @property
        def total_tokens(self) -> float:
            return (self.input_tokens or 0) + (self.output_tokens or 0)

        def __add__(self, other: BaseMessage.Usage) -> BaseMessage.Usage:
            return BaseMessage.Usage(
                input_tokens=(self.input_tokens or 0) + (other.input_tokens or 0),
                output_tokens=(self.output_tokens or 0) + (other.output_tokens or 0),
            )

    USER_ROLE = "user"
    ASSISTANT_ROLE = "assistant"
    SYSTEM_ROLE = "system"

    content: list[Union[BaseMessageContent, BaseDeltaMessageContent]] =␣
↪field(metadata={"serializable": True})
    role: str = field(kw_only=True, metadata={"serializable": True})
    usage: Usage = field(kw_only=True, default=Factory(lambda: BaseMessage.Usage()),␣
↪metadata={"serializable": True})

    def is_system(self) -> bool:
        return self.role == self.SYSTEM_ROLE
```

(continues on next page)

(continued from previous page)

```
    def is_user(self) -> bool:
        return self.role == self.USER_ROLE

    def is_assistant(self) -> bool:
        return self.role == self.ASSISTANT_ROLE
```

- `ASSISTANT_ROLE = 'assistant'` class-attribute instance-attribute
- `SYSTEM_ROLE = 'system'` class-attribute instance-attribute
- `USER_ROLE = 'user'` class-attribute instance-attribute
- `content = field(metadata={'serializable': True})` class-attribute instance-attribute
- `role = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `usage = field(kw_only=True, default=Factory(lambda: BaseMessage.Usage()), metadata={'serializable': True})` class-attribute instance-attribute

Usage

Bases: *SerializableMixin* “)

Source Code in griptape/common/prompt_stack/messages/base_message.py

```
@define
class Usage(SerializableMixin):
    input_tokens: Optional[float] = field(kw_only=True, default=None, metadata={
↪"serializable": True})
    output_tokens: Optional[float] = field(kw_only=True, default=None, metadata={
↪"serializable": True})

    @property
    def total_tokens(self) -> float:
        return (self.input_tokens or 0) + (self.output_tokens or 0)

    def __add__(self, other: BaseMessage.Usage) -> BaseMessage.Usage:
        return BaseMessage.Usage(
            input_tokens=(self.input_tokens or 0) + (other.input_tokens or 0),
            output_tokens=(self.output_tokens or 0) + (other.output_tokens or 0),
        )
```

- `input_tokens = field(kw_only=True, default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `output_tokens = field(kw_only=True, default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `total_tokens` property

is_assistant()

Source Code in griptape/common/prompt_stack/messages/base_message.py

```
def is_assistant(self) -> bool:
    return self.role == self.ASSISTANT_ROLE
```

is_system()

Source Code in griptape/common/prompt_stack/messages/base_message.py

```
def is_system(self) -> bool:
    return self.role == self.SYSTEM_ROLE
```

is_user()

Source Code in griptape/common/prompt_stack/messages/base_message.py

```
def is_user(self) -> bool:
    return self.role == self.USER_ROLE
```

delta_message

Bases: *BaseMessage* “)

Source Code in griptape/common/prompt_stack/messages/delta_message.py

```
@define
class DeltaMessage(BaseMessage):
    role: Optional[str] = field(kw_only=True, default=None, metadata={"serializable":␣
↪True})
    content: Optional[BaseDeltaMessageContent] = field(kw_only=True, default=None,␣
↪metadata={"serializable": True})
```

- `content = field(kw_only=True, default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `role = field(kw_only=True, default=None, metadata={'serializable': True})` class-attribute instance-attribute

message

- `T = TypeVar('T', bound=BaseMessageContent)` module-attribute

Bases: *BaseMessage* “)

Source Code in griptape/common/prompt_stack/messages/message.py

```
@define
class Message(BaseMessage):
    def __init__(self, content: str | list[BaseMessageContent], **kwargs: Any) ->␣
↪None:
        if isinstance(content, str):
            content = [TextMessageContent(TextArtifact(value=content))]
        self.__attrs_init__(content, **kwargs)  # pyright:␣
```

(continues on next page)

(continued from previous page)

```
↪ignore[reportAttributeAccessIssue]

    content: list[BaseMessageContent] = field(metadata={"serializable": True})

    @property
    def value(self) -> Any:
        return self.to_artifact().value

    def __str__(self) -> str:
        return self.to_text()

    def has_all_content_type(self, content_type: type[T]) -> bool:
        return all(isinstance(content, content_type) for content in self.content)

    def has_any_content_type(self, content_type: type[T]) -> bool:
        return any(isinstance(content, content_type) for content in self.content)

    def get_content_type(self, content_type: type[T]) -> list[T]:
        return [content for content in self.content if isinstance(content, content_
↪type)]

    def is_text(self) -> bool:
        return all(isinstance(content, TextMessageContent) for content in self.
↪content)

    def to_text(self) -> str:
        return "".join(
            [content.artifact.to_text() for content in self.content if␣
↪isinstance(content, TextMessageContent)],
        )

    def to_artifact(self, meta: Optional[dict] = None) -> BaseArtifact:
        if meta is None:
            meta = {}
        if len(self.content) == 1:
            artifact = self.content[0].artifact
        else:
            artifact = ListArtifact([content.artifact for content in self.content])

        artifact.meta.update(meta)
        return artifact
```

- `content = field(metadata={'serializable': True})` class-attribute instance-attribute
- `value` property

init(content, **kwargs)

Source Code in griptape/common/prompt_stack/messages/message.py

```
def __init__(self, content: str | list[BaseMessageContent], **kwargs: Any) -> None:
    if isinstance(content, str):
        content = [TextMessageContent(TextArtifact(value=content))]
```

(continues on next page)

(continued from previous page)

```
    self.__attrs_init__(content, **kwargs)  # pyright:␣
↪ignore[reportAttributeAccessIssue]
```

str()

Source Code in griptape/common/prompt_stack/messages/message.py

```
def __str__(self) -> str:
    return self.to_text()
```

get_content_type(content_type)

Source Code in griptape/common/prompt_stack/messages/message.py

```
def get_content_type(self, content_type: type[T]) -> list[T]:
    return [content for content in self.content if isinstance(content, content_type)]
```

has_all_content_type(content_type)

Source Code in griptape/common/prompt_stack/messages/message.py

```
def has_all_content_type(self, content_type: type[T]) -> bool:
    return all(isinstance(content, content_type) for content in self.content)
```

has_any_content_type(content_type)

Source Code in griptape/common/prompt_stack/messages/message.py

```
def has_any_content_type(self, content_type: type[T]) -> bool:
    return any(isinstance(content, content_type) for content in self.content)
```

is_text()

Source Code in griptape/common/prompt_stack/messages/message.py

```
def is_text(self) -> bool:
    return all(isinstance(content, TextMessageContent) for content in self.content)
```

to_artifact(meta=None)

Source Code in griptape/common/prompt_stack/messages/message.py

```
def to_artifact(self, meta: Optional[dict] = None) -> BaseArtifact:
    if meta is None:
        meta = {}
    if len(self.content) == 1:
        artifact = self.content[0].artifact
    else:
        artifact = ListArtifact([content.artifact for content in self.content])
```

(continues on next page)

(continued from previous page)

```
    artifact.meta.update(meta)
    return artifact
```

to_text()

Source Code in griptape/common/prompt_stack/messages/message.py

```
def to_text(self) -> str:
    return "".join(
        [content.artifact.to_text() for content in self.content if isinstance(content,
↪ TextMessageContent)],
    )
```

prompt_stack

Bases: *SerializableMixin* “)

Source Code in griptape/common/prompt_stack/prompt_stack.py

```
@define
class PromptStack(SerializableMixin):
    messages: list[Message] = field(factory=list, kw_only=True, metadata={
↪"serializable": True})
    tools: list[BaseTool] = field(factory=list, kw_only=True)
    output_schema: Optional[Union[Schema, type[BaseModel]]] = field(default=None, kw_
↪only=True)

    @property
    def system_messages(self) -> list[Message]:
        return [message for message in self.messages if message.is_system()]

    @property
    def user_messages(self) -> list[Message]:
        return [message for message in self.messages if message.is_user()]

    @property
    def assistant_messages(self) -> list[Message]:
        return [message for message in self.messages if message.is_assistant()]

    def add_message(self, artifact: str | BaseArtifact, role: str) -> Message:
        content = self.__to_message_content(artifact)

        self.messages.append(Message(content=content, role=role))

        return self.messages[-1]

    def add_system_message(self, artifact: str | BaseArtifact) -> Message:
        return self.add_message(artifact, Message.SYSTEM_ROLE)

    def add_user_message(self, artifact: str | BaseArtifact) -> Message:
        return self.add_message(artifact, Message.USER_ROLE)
```

(continues on next page)

(continued from previous page)

```
    def add_assistant_message(self, artifact: str | BaseArtifact) -> Message:
        return self.add_message(artifact, Message.ASSISTANT_ROLE)

    def to_output_json_schema(self, schema_id: str = "Output Format") -> dict:
        if self.output_schema is None:
            raise ValueError("Output schema is not set")

        if isinstance(self.output_schema, Schema):
            json_schema = self.output_schema.json_schema(schema_id)
        elif isinstance(self.output_schema, type) and issubclass(self.output_schema,␣
↪BaseModel):
            json_schema = build_strict_schema(self.output_schema.model_json_schema(),␣
↪schema_id)
        else:
            raise ValueError(f"Unsupported output schema type: {type(self.output_
↪schema)}")

        return json_schema

    @classmethod
    def from_artifact(cls, artifact: BaseArtifact) -> PromptStack:
        prompt_stack = cls()
        prompt_stack.add_user_message(artifact)

        return prompt_stack

    def __to_message_content(self, artifact: str | BaseArtifact) ->␣
↪list[BaseMessageContent]:
        if isinstance(artifact, str):
            return [TextMessageContent(TextArtifact(artifact))]
        if isinstance(artifact, TextArtifact):
            return [TextMessageContent(artifact)]
        if isinstance(artifact, ImageArtifact):
            return [ImageMessageContent(artifact)]
        if isinstance(artifact, ImageUrlArtifact):
            return [ImageMessageContent(artifact)]
        if isinstance(artifact, AudioArtifact):
            return [AudioMessageContent(artifact)]
        if isinstance(artifact, GenericArtifact):
            return [GenericMessageContent(artifact)]
        if isinstance(artifact, ActionArtifact):
            action = artifact.value
            output = action.output
            if output is None:
                return [ActionCallMessageContent(artifact)]
            return [ActionResultMessageContent(output, action=action)]
        if isinstance(artifact, ListArtifact):
            processed_contents = [self.__to_message_content(artifact) for artifact in␣
↪artifact.value]

            return [sub_content for processed_content in processed_contents for sub_
↪content in processed_content]
```

(continues on next page)

(continued from previous page)

```
        return [TextMessageContent(TextArtifact(artifact.to_text()))]
```

- `assistant_messages` property
- `messages = field(factory=list, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output_schema = field(default=None, kw_only=True)` class-attribute instance-attribute
- `system_messages` property
- `tools = field(factory=list, kw_only=True)` class-attribute instance-attribute
- `user_messages` property

__to_message_content(artifact)

Source Code in griptape/common/prompt_stack/prompt_stack.py

```
def __to_message_content(self, artifact: str | BaseArtifact) ->␣
↪list[BaseMessageContent]:
    if isinstance(artifact, str):
        return [TextMessageContent(TextArtifact(artifact))]
    if isinstance(artifact, TextArtifact):
        return [TextMessageContent(artifact)]
    if isinstance(artifact, ImageArtifact):
        return [ImageMessageContent(artifact)]
    if isinstance(artifact, ImageUrlArtifact):
        return [ImageMessageContent(artifact)]
    if isinstance(artifact, AudioArtifact):
        return [AudioMessageContent(artifact)]
    if isinstance(artifact, GenericArtifact):
        return [GenericMessageContent(artifact)]
    if isinstance(artifact, ActionArtifact):
        action = artifact.value
        output = action.output
        if output is None:
            return [ActionCallMessageContent(artifact)]
        return [ActionResultMessageContent(output, action=action)]
    if isinstance(artifact, ListArtifact):
        processed_contents = [self.__to_message_content(artifact) for artifact in␣
↪artifact.value]

        return [sub_content for processed_content in processed_contents for sub_
↪content in processed_content]
    return [TextMessageContent(TextArtifact(artifact.to_text()))]
```

add_assistant_message(artifact)

Source Code in griptape/common/prompt_stack/prompt_stack.py

```
def add_assistant_message(self, artifact: str | BaseArtifact) -> Message:
    return self.add_message(artifact, Message.ASSISTANT_ROLE)
```

add_message(artifact, role)

Source Code in griptape/common/prompt_stack/prompt_stack.py

```
def add_message(self, artifact: str | BaseArtifact, role: str) -> Message:
    content = self.__to_message_content(artifact)

    self.messages.append(Message(content=content, role=role))

    return self.messages[-1]
```

add_system_message(artifact)

Source Code in griptape/common/prompt_stack/prompt_stack.py

```
def add_system_message(self, artifact: str | BaseArtifact) -> Message:
    return self.add_message(artifact, Message.SYSTEM_ROLE)
```

add_user_message(artifact)

Source Code in griptape/common/prompt_stack/prompt_stack.py

```
def add_user_message(self, artifact: str | BaseArtifact) -> Message:
    return self.add_message(artifact, Message.USER_ROLE)
```

from_artifact(artifact)classmethod

Source Code in griptape/common/prompt_stack/prompt_stack.py

```
@classmethod
def from_artifact(cls, artifact: BaseArtifact) -> PromptStack:
    prompt_stack = cls()
    prompt_stack.add_user_message(artifact)

    return prompt_stack
```

to_output_json_schema(schema_id='Output Format')

Source Code in griptape/common/prompt_stack/prompt_stack.py

```
def to_output_json_schema(self, schema_id: str = "Output Format") -> dict:
    if self.output_schema is None:
        raise ValueError("Output schema is not set")

    if isinstance(self.output_schema, Schema):
        json_schema = self.output_schema.json_schema(schema_id)
    elif isinstance(self.output_schema, type) and issubclass(self.output_schema,␣
↪BaseModel):
        json_schema = build_strict_schema(self.output_schema.model_json_schema(),␣
↪schema_id)
    else:
        raise ValueError(f"Unsupported output schema type: {type(self.output_schema)}
```

(continues on next page)

(continued from previous page)

```
↪")

    return json_schema
```

reference

Bases: *SerializableMixin* “)

Source Code in griptape/common/reference.py

True})

- `authors = field(factory=list, metadata={'serializable': True})` class-attribute instance-attribute
- `id = field(default=Factory(lambda: uuid.uuid4().hex), metadata={'serializable': True})` class-attribute instance-attribute
- `source = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `title = field(metadata={'serializable': True})` class-attribute instance-attribute
- `url = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `year = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

11.5.5 configs

- `Defaults = _DefaultsConfig()` module-attribute
- `__all__ = ['BaseConfig', 'Defaults']` module-attribute

Bases: *SerializableMixin* “) , ABC

Source Code in griptape/configs/base_config.py

```
@define(kw_only=True)
class BaseConfig(SerializableMixin, ABC): ...
```

base_config

Bases: *SerializableMixin* “) , ABC

Source Code in griptape/configs/base_config.py

```
@define(kw_only=True)
class BaseConfig(SerializableMixin, ABC): ...
```

defaults_config

- `Defaults = _DefaultsConfig()` module-attribute

Bases: *BaseConfig* “) , *SingletonMixin* ”)

Source Code in griptape/configs/defaults_config.py

```
@define(kw_only=True)
class _DefaultsConfig(BaseConfig, SingletonMixin):
    _logging_config: Optional[LoggingConfig] = field(default=None)
    _drivers_config: Optional[BaseDriversConfig] = field(default=None)

    @lazy_property()
    def logging_config(self) -> LoggingConfig:
        return LoggingConfig()

    @lazy_property()
    def drivers_config(self) -> BaseDriversConfig:
        from griptape.configs.drivers.openai_drivers_config import OpenAiDriversConfig

        return OpenAiDriversConfig()
```

- `_drivers_config = field(default=None)` class-attribute instance-attribute
- `_logging_config = field(default=None)` class-attribute instance-attribute

drivers_config()

Source Code in griptape/configs/defaults_config.py

```
@lazy_property()
def drivers_config(self) -> BaseDriversConfig:
    from griptape.configs.drivers.openai_drivers_config import OpenAiDriversConfig

    return OpenAiDriversConfig()
```

logging_config()

Source Code in griptape/configs/defaults_config.py

```
@lazy_property()
def logging_config(self) -> LoggingConfig:
    return LoggingConfig()
```

drivers

- `__all__ = ['AmazonBedrockDriversConfig', 'AnthropicDriversConfig', 'AzureOpenAiDriversConfig', 'BaseDriversConfig', 'CohereDriversConfig', 'DriversConfig', 'GoogleDriversConfig', 'OpenAiDriversConfig']` module-attribute

Bases: *AnthropicDriversConfig* “)

Source Code in griptape/configs/drivers/amazon_bedrock_drivers_config.py

```
@define
class AmazonBedrockDriversConfig(DriversConfig):
    session: boto3.Session = field(
        default=Factory(lambda: import_optional_dependency("boto3").Session()),
        kw_only=True,
        metadata={"serializable": False},
    )
```

(continues on next page)

(continued from previous page)

```
    @lazy_property()
    def prompt_driver(self) -> AmazonBedrockPromptDriver:
        return AmazonBedrockPromptDriver(session=self.session, model="anthropic.
↪claude-3-7-sonnet-20250219-v1:0")

    @lazy_property()
    def embedding_driver(self) -> AmazonBedrockTitanEmbeddingDriver:
        return AmazonBedrockTitanEmbeddingDriver(session=self.session, model="amazon.
↪titan-embed-text-v2:0")

    @lazy_property()
    def image_generation_driver(self) -> AmazonBedrockImageGenerationDriver:
        return AmazonBedrockImageGenerationDriver(
            session=self.session,
            model="amazon.titan-image-generator-v2:0",
            image_generation_model_driver=BedrockTitanImageGenerationModelDriver(),
        )

    @lazy_property()
    def vector_store_driver(self) -> LocalVectorStoreDriver:
        return LocalVectorStoreDriver(
            embedding_driver=AmazonBedrockTitanEmbeddingDriver(
                session=self.session, model="amazon.titan-embed-text-v2:0"
            )
        )
```

- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute

embedding_driver()

Source Code in griptape/configs/drivers/amazon_bedrock_drivers_config.py

```
@lazy_property()
def embedding_driver(self) -> AmazonBedrockTitanEmbeddingDriver:
    return AmazonBedrockTitanEmbeddingDriver(session=self.session, model="amazon.
↪titan-embed-text-v2:0")
```

image_generation_driver()

Source Code in griptape/configs/drivers/amazon_bedrock_drivers_config.py

```
@lazy_property()
def image_generation_driver(self) -> AmazonBedrockImageGenerationDriver:
    return AmazonBedrockImageGenerationDriver(
        session=self.session,
        model="amazon.titan-image-generator-v2:0",
        image_generation_model_driver=BedrockTitanImageGenerationModelDriver(),
    )
```

prompt_driver()

Source Code in griptape/configs/drivers/amazon_bedrock_drivers_config.py

```
@lazy_property()
def prompt_driver(self) -> AmazonBedrockPromptDriver:
    return AmazonBedrockPromptDriver(session=self.session, model="anthropic.claude-3-
↪7-sonnet-20250219-v1:0")
```

vector_store_driver()

Source Code in griptape/configs/drivers/amazon_bedrock_drivers_config.py

```
@lazy_property()
def vector_store_driver(self) -> LocalVectorStoreDriver:
    return LocalVectorStoreDriver(
        embedding_driver=AmazonBedrockTitanEmbeddingDriver(
            session=self.session, model="amazon.titan-embed-text-v2:0"
        )
    )
```

AnthropicDriversConfig

Bases: *AnthropicDriversConfig* “)

Source Code in griptape/configs/drivers/anthropic_drivers_config.py

```
@define
class AnthropicDriversConfig(DriversConfig):
    @lazy_property()
    def prompt_driver(self) -> AnthropicPromptDriver:
        return AnthropicPromptDriver(model="claude-3-7-sonnet-latest")
```

prompt_driver()

Source Code in griptape/configs/drivers/anthropic_drivers_config.py

```
@lazy_property()
def prompt_driver(self) -> AnthropicPromptDriver:
    return AnthropicPromptDriver(model="claude-3-7-sonnet-latest")
```

AzureOpenAiDriversConfig

Bases: *AnthropicDriversConfig* “)

Attributes

Name	Type	Description
<a href="./#griptape.configs.drivers.AzureOpenAiDriversConfig.azure_endpoint" title="azure_endpoint = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.configs.drivers.AzureOpenAiDriversConfig.azure_endpoint)"><code class="language-python">	*str*	The endpoint for the Azure OpenAI instance.
<a href="./#griptape.configs.drivers.AzureOpenAiDriversConfig.azure_ad_token" title="azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.configs.drivers.AzureOpenAiDriversConfig.azure_ad_token)"><code class="language-python">	*Optional[str]*	An optional Azure Active Directory token.

continues on next page

Table 35 – continued from previous page

Name	Type	Description
<a href="./#griptape.configs.drivers.AzureOpenAiDriversConfig.azure_ad_token_provider" title="azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.configs.drivers.AzureOpenAiDriversConfig.azure_ad_token_provider)"><code class="language-python">	*Optional[Callable[[], str]]*	An optional Azure Active Directory token provider.
<a href="./#griptape.configs.drivers.AzureOpenAiDriversConfig.api_key" title="api_key = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.configs.drivers.AzureOpenAiDriversConfig.api_key)"><code class="language-python">	*Optional[str]*	An optional Azure API key.
<a href="./#griptape.configs.drivers.AzureOpenAiDriversConfig.prompt_driver" title="prompt_driver() (griptape.configs.drivers.AzureOpenAiDriversConfig.prompt_driver)"><coc class="language-python">	<a href="../../drivers/prompt/openai.mdx#griptape.drivers.prompt.openai.AzureOpenAiChatPromptDriver" title="AzureOpenAiChatPromptDriver (griptape.drivers.prompt.openai.AzureOpenAiChatPromptDriver)"><code class="language-python">AzureOpenAiChatPromptDr	An Azure OpenAI Chat Prompt Driver.

continues on next page

Table 35 – continued from previous page

Name	Type	Description
<a href="./#griptape.configs.drivers.AzureOpenAiDriversConfig.image_generation_driver" title="image_generation_driver() (griptape.configs.drivers.AzureOpenAiDriversConfig.image_generation_driver)"><code class="language-python">	<a href="../../drivers/image_generation/openai.mdx#griptape.drivers.image_generation.openai.AzureOpenAiImageGenerationDriver" title="AzureOpenAiImageGenerationDriver (griptape.drivers.image_generation.openai.AzureOpenAiImageGenerationDriver)"><code class="language-python">AzureOpenAiImageGenerat	An Azure OpenAI Image Generation Driver.
<a href="./#griptape.configs.drivers.AzureOpenAiDriversConfig.embedding_driver" title="embedding_driver() (griptape.configs.drivers.AzureOpenAiDriversConfig.embedding_driver)"><code class="language-python">	<a href="../../drivers/embedding/openai.mdx#griptape.drivers.embedding.openai.AzureOpenAiEmbeddingDriver" title="AzureOpenAiEmbeddingDriver (griptape.drivers.embedding.openai.AzureOpenAiEmbeddingDriver)"><code class="language-python">AzureOpenAiEmbeddingDr	An Azure OpenAI Embedding Driver.
<a href="./#griptape.configs.drivers.AzureOpenAiDriversConfig.vector_store_driver" title="vector_store_driver() (griptape.configs.drivers.AzureOpenAiDriversConfig.vector_store_driver)"><code class="language-python">	<a href="../../drivers/vector/local.mdx#griptape.drivers.vector.local.LocalVectorStoreDriver" title="LocalVectorStoreDriver (griptape.drivers.vector.local.LocalVectorStoreDriver)"><code class="language-python">LocalVectorStoreDriver</co	A Local Vector Store Driver.

Source Code in griptape/configs/drivers/azure_openai_drivers_config.py

```
@define
class AzureOpenAiDriversConfig(DriversConfig):
    """Azure OpenAI Drivers Configuration.

    Attributes:
        azure_endpoint: The endpoint for the Azure OpenAI instance.
        azure_ad_token: An optional Azure Active Directory token.
        azure_ad_token_provider: An optional Azure Active Directory token provider.
        api_key: An optional Azure API key.
```

(continues on next page)

(continued from previous page)

```
        prompt_driver: An Azure OpenAI Chat Prompt Driver.
        image_generation_driver: An Azure OpenAI Image Generation Driver.
        embedding_driver: An Azure OpenAI Embedding Driver.
        vector_store_driver: A Local Vector Store Driver.
    """

    azure_endpoint: str = field(kw_only=True, metadata={"serializable": True})
    azure_ad_token: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": False})
    azure_ad_token_provider: Optional[Callable[[], str]] = field(
        kw_only=True,
        default=None,
        metadata={"serializable": False},
    )
    api_key: Optional[str] = field(kw_only=True, default=None, metadata={"serializable
↪": False})

    @lazy_property()
    def prompt_driver(self) -> AzureOpenAiChatPromptDriver:
        return AzureOpenAiChatPromptDriver(
            model="gpt-4.1",
            azure_endpoint=self.azure_endpoint,
            api_key=self.api_key,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )

    @lazy_property()
    def embedding_driver(self) -> AzureOpenAiEmbeddingDriver:
        return AzureOpenAiEmbeddingDriver(
            model="text-embedding-3-small",
            azure_endpoint=self.azure_endpoint,
            api_key=self.api_key,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )

    @lazy_property()
    def image_generation_driver(self) -> AzureOpenAiImageGenerationDriver:
        return AzureOpenAiImageGenerationDriver(
            model="dall-e-2",
            azure_endpoint=self.azure_endpoint,
            api_key=self.api_key,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
            image_size="512x512",
        )

    @lazy_property()
    def vector_store_driver(self) -> LocalVectorStoreDriver:
        return LocalVectorStoreDriver(
            embedding_driver=AzureOpenAiEmbeddingDriver(
```

(continues on next page)

(continued from previous page)

```
                model="text-embedding-3-small",
                azure_endpoint=self.azure_endpoint,
                api_key=self.api_key,
                azure_ad_token=self.azure_ad_token,
                azure_ad_token_provider=self.azure_ad_token_provider,
            )
        )

    @lazy_property()
    def text_to_speech_driver(self) -> AzureOpenAiTextToSpeechDriver:
        return AzureOpenAiTextToSpeechDriver(
            model="tts",
            azure_endpoint=self.azure_endpoint,
            api_key=self.api_key,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )
```

- `api_key = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

embedding_driver()

Source Code in griptape/configs/drivers/azure_openai_drivers_config.py

```
@lazy_property()
def embedding_driver(self) -> AzureOpenAiEmbeddingDriver:
    return AzureOpenAiEmbeddingDriver(
        model="text-embedding-3-small",
        azure_endpoint=self.azure_endpoint,
        api_key=self.api_key,
        azure_ad_token=self.azure_ad_token,
        azure_ad_token_provider=self.azure_ad_token_provider,
    )
```

image_generation_driver()

Source Code in griptape/configs/drivers/azure_openai_drivers_config.py

```
@lazy_property()
def image_generation_driver(self) -> AzureOpenAiImageGenerationDriver:
    return AzureOpenAiImageGenerationDriver(
        model="dall-e-2",
        azure_endpoint=self.azure_endpoint,
```

(continues on next page)

(continued from previous page)

```
        api_key=self.api_key,
        azure_ad_token=self.azure_ad_token,
        azure_ad_token_provider=self.azure_ad_token_provider,
        image_size="512x512",
    )
```

prompt_driver()

Source Code in griptape/configs/drivers/azure_openai_drivers_config.py

```
@lazy_property()
def prompt_driver(self) -> AzureOpenAiChatPromptDriver:
    return AzureOpenAiChatPromptDriver(
        model="gpt-4.1",
        azure_endpoint=self.azure_endpoint,
        api_key=self.api_key,
        azure_ad_token=self.azure_ad_token,
        azure_ad_token_provider=self.azure_ad_token_provider,
    )
```

text_to_speech_driver()

Source Code in griptape/configs/drivers/azure_openai_drivers_config.py

```
@lazy_property()
def text_to_speech_driver(self) -> AzureOpenAiTextToSpeechDriver:
    return AzureOpenAiTextToSpeechDriver(
        model="tts",
        azure_endpoint=self.azure_endpoint,
        api_key=self.api_key,
        azure_ad_token=self.azure_ad_token,
        azure_ad_token_provider=self.azure_ad_token_provider,
    )
```

vector_store_driver()

Source Code in griptape/configs/drivers/azure_openai_drivers_config.py

```
@lazy_property()
def vector_store_driver(self) -> LocalVectorStoreDriver:
    return LocalVectorStoreDriver(
        embedding_driver=AzureOpenAiEmbeddingDriver(
            model="text-embedding-3-small",
            azure_endpoint=self.azure_endpoint,
            api_key=self.api_key,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )
    )
```

BaseDriversConfig

Bases:

ABC, *SerializableMixin* “)

Source Code in griptape/configs/drivers/base_drivers_config.py

```
@define
class BaseDriversConfig(ABC, SerializableMixin):
    _prompt_driver: Optional[BasePromptDriver] = field(
        kw_only=True, default=None, metadata={"serializable": True}, alias="prompt_
↪driver"
    )
    _image_generation_driver: Optional[BaseImageGenerationDriver] = field(
        kw_only=True, default=None, metadata={"serializable": True}, alias="image_
↪generation_driver"
    )
    _embedding_driver: Optional[BaseEmbeddingDriver] = field(
        kw_only=True, default=None, metadata={"serializable": True}, alias="embedding_
↪driver"
    )
    _vector_store_driver: Optional[BaseVectorStoreDriver] = field(
        default=None, kw_only=True, metadata={"serializable": True}, alias="vector_
↪store_driver"
    )
    _conversation_memory_driver: Optional[BaseConversationMemoryDriver] = field(
        default=None, kw_only=True, metadata={"serializable": True}, alias=
↪"conversation_memory_driver"
    )
    _text_to_speech_driver: Optional[BaseTextToSpeechDriver] = field(
        default=None, kw_only=True, metadata={"serializable": True}, alias="text_to_
↪speech_driver"
    )
    _audio_transcription_driver: Optional[BaseAudioTranscriptionDriver] = field(
        default=None, kw_only=True, metadata={"serializable": True}, alias="audio_
↪transcription_driver"
    )
    _ruleset_driver: Optional[BaseRulesetDriver] = field(
        default=None, kw_only=True, metadata={"serializable": True}, alias="ruleset_
↪driver"
    )

    _last_drivers_config: Optional[BaseDriversConfig] = field(default=None)

    def __enter__(self) -> Self:
        from griptape.configs import Defaults

        self._last_drivers_config = Defaults.drivers_config

        Defaults.drivers_config = self

        return self

    def __exit__(self, type, value, traceback) -> None:  # noqa: ANN001, A002
```

(continues on next page)

(continued from previous page)

```
        from griptape.configs import Defaults

        if self._last_drivers_config is not None:
            Defaults.drivers_config = self._last_drivers_config

        self._last_drivers_config = None

    @lazy_property()
    @abstractmethod
    def prompt_driver(self) -> BasePromptDriver: ...

    @lazy_property()
    @abstractmethod
    def image_generation_driver(self) -> BaseImageGenerationDriver: ...

    @lazy_property()
    @abstractmethod
    def embedding_driver(self) -> BaseEmbeddingDriver: ...

    @lazy_property()
    @abstractmethod
    def vector_store_driver(self) -> BaseVectorStoreDriver: ...

    @lazy_property()
    @abstractmethod
    def conversation_memory_driver(self) -> BaseConversationMemoryDriver: ...

    @lazy_property()
    @abstractmethod
    def text_to_speech_driver(self) -> BaseTextToSpeechDriver: ...

    @lazy_property()
    @abstractmethod
    def audio_transcription_driver(self) -> BaseAudioTranscriptionDriver: ...

    @lazy_property()
    @abstractmethod
    def ruleset_driver(self) -> BaseRulesetDriver: ...
```

- `_audio_transcription_driver = field(default=None, kw_only=True, metadata={'serializable': True}, alias='audio_transcription_driver')` class-attribute instance-attribute
- `_conversation_memory_driver = field(default=None, kw_only=True, metadata={'serializable': True}, alias='conversation_memory_driver')` class-attribute instance-attribute
- `_embedding_driver = field(kw_only=True, default=None, metadata={'serializable': True}, alias='embedding_driver')` class-attribute instance-attribute
- `_image_generation_driver = field(kw_only=True, default=None, metadata={'serializable': True}, alias='image_generation_driver')` class-attribute instance-attribute
- `_last_drivers_config = field(default=None)` class-attribute instance-attribute

- `_prompt_driver = field(kw_only=True, default=None, metadata={'serializable': True}, alias='prompt_driver')` class-attribute instance-attribute
- `_ruleset_driver = field(default=None, kw_only=True, metadata={'serializable': True}, alias='ruleset_driver')` class-attribute instance-attribute
- `_text_to_speech_driver = field(default=None, kw_only=True, metadata={'serializable': True}, alias='text_to_speech_driver')` class-attribute instance-attribute
- `_vector_store_driver = field(default=None, kw_only=True, metadata={'serializable': True}, alias='vector_store_driver')` class-attribute instance-attribute

enter()

Source Code in griptape/configs/drivers/base_drivers_config.py

```
def __enter__(self) -> Self:
    from griptape.configs import Defaults

    self._last_drivers_config = Defaults.drivers_config

    Defaults.drivers_config = self

    return self
```

exit(type, value, traceback)

Source Code in griptape/configs/drivers/base_drivers_config.py

```
def __exit__(self, type, value, traceback) -> None:  # noqa: ANN001, A002
    from griptape.configs import Defaults

    if self._last_drivers_config is not None:
        Defaults.drivers_config = self._last_drivers_config

    self._last_drivers_config = None
```

audio_transcription_driver()abstractmethod

Source Code in griptape/configs/drivers/base_drivers_config.py

```
@lazy_property()
@abstractmethod
def audio_transcription_driver(self) -> BaseAudioTranscriptionDriver: ...
```

conversation_memory_driver()abstractmethod

Source Code in griptape/configs/drivers/base_drivers_config.py

```
@lazy_property()
@abstractmethod
def conversation_memory_driver(self) -> BaseConversationMemoryDriver: ...
```

embedding_driver()abstractmethod

Source Code in griptape/configs/drivers/base_drivers_config.py

```
@lazy_property()
@abstractmethod
def embedding_driver(self) -> BaseEmbeddingDriver: ...
```

image_generation_driver()abstractmethod

Source Code in griptape/configs/drivers/base_drivers_config.py

```
@lazy_property()
@abstractmethod
def image_generation_driver(self) -> BaseImageGenerationDriver: ...
```

prompt_driver()abstractmethod

Source Code in griptape/configs/drivers/base_drivers_config.py

```
@lazy_property()
@abstractmethod
def prompt_driver(self) -> BasePromptDriver: ...
```

ruleset_driver()abstractmethod

Source Code in griptape/configs/drivers/base_drivers_config.py

```
@lazy_property()
@abstractmethod
def ruleset_driver(self) -> BaseRulesetDriver: ...
```

text_to_speech_driver()abstractmethod

Source Code in griptape/configs/drivers/base_drivers_config.py

```
@lazy_property()
@abstractmethod
def text_to_speech_driver(self) -> BaseTextToSpeechDriver: ...
```

vector_store_driver()abstractmethod

Source Code in griptape/configs/drivers/base_drivers_config.py

```
@lazy_property()
@abstractmethod
def vector_store_driver(self) -> BaseVectorStoreDriver: ...
```

CohereDriversConfig

Bases: *AnthropicDriversConfig* “)

Source Code in griptape/configs/drivers/cohere_drivers_config.py

```
@define
class CohereDriversConfig(DriversConfig):
    api_key: str = field(metadata={"serializable": False}, kw_only=True)

    @lazy_property()
    def prompt_driver(self) -> CoherePromptDriver:
        return CoherePromptDriver(model="command-r", api_key=self.api_key)

    @lazy_property()
    def embedding_driver(self) -> CohereEmbeddingDriver:
        return CohereEmbeddingDriver(
            model="embed-english-v3.0",
            api_key=self.api_key,
            input_type="search_document",
        )

    @lazy_property()
    def vector_store_driver(self) -> LocalVectorStoreDriver:
        return LocalVectorStoreDriver(
            embedding_driver=CohereEmbeddingDriver(
                model="embed-english-v3.0",
                api_key=self.api_key,
                input_type="search_document",
            )
        )
```

- `api_key = field(metadata={'serializable': False}, kw_only=True)` class-attribute instance-attribute

embedding_driver()

Source Code in griptape/configs/drivers/cohere_drivers_config.py

```
@lazy_property()
def embedding_driver(self) -> CohereEmbeddingDriver:
    return CohereEmbeddingDriver(
        model="embed-english-v3.0",
        api_key=self.api_key,
        input_type="search_document",
    )
```

prompt_driver()

Source Code in griptape/configs/drivers/cohere_drivers_config.py

```
@lazy_property()
def prompt_driver(self) -> CoherePromptDriver:
    return CoherePromptDriver(model="command-r", api_key=self.api_key)
```

vector_store_driver()

Source Code in griptape/configs/drivers/cohere_drivers_config.py

```
@lazy_property()
def vector_store_driver(self) -> LocalVectorStoreDriver:
    return LocalVectorStoreDriver(
        embedding_driver=CohereEmbeddingDriver(
            model="embed-english-v3.0",
            api_key=self.api_key,
            input_type="search_document",
        )
    )
```

DriversConfig

Bases: *BaseDriversConfig* “)

Source Code in griptape/configs/drivers/drivers_config.py

```
@define
class DriversConfig(BaseDriversConfig):
    @lazy_property()
    def prompt_driver(self) -> BasePromptDriver:
        return DummyPromptDriver()

    @lazy_property()
    def image_generation_driver(self) -> BaseImageGenerationDriver:
        return DummyImageGenerationDriver()

    @lazy_property()
    def embedding_driver(self) -> BaseEmbeddingDriver:
        return DummyEmbeddingDriver()

    @lazy_property()
    def vector_store_driver(self) -> BaseVectorStoreDriver:
        return DummyVectorStoreDriver(embedding_driver=self.embedding_driver)

    @lazy_property()
    def conversation_memory_driver(self) -> BaseConversationMemoryDriver:
        return LocalConversationMemoryDriver()

    @lazy_property()
    def text_to_speech_driver(self) -> BaseTextToSpeechDriver:
        return DummyTextToSpeechDriver()

    @lazy_property()
    def audio_transcription_driver(self) -> BaseAudioTranscriptionDriver:
        return DummyAudioTranscriptionDriver()

    @lazy_property()
    def ruleset_driver(self) -> BaseRulesetDriver:
        return LocalRulesetDriver()
```

audio_transcription_driver()

Source Code in griptape/configs/drivers/drivers_config.py

```
@lazy_property()
def audio_transcription_driver(self) -> BaseAudioTranscriptionDriver:
    return DummyAudioTranscriptionDriver()
```

conversation_memory_driver()

Source Code in griptape/configs/drivers/drivers_config.py

```
@lazy_property()
def conversation_memory_driver(self) -> BaseConversationMemoryDriver:
    return LocalConversationMemoryDriver()
```

embedding_driver()

Source Code in griptape/configs/drivers/drivers_config.py

```
@lazy_property()
def embedding_driver(self) -> BaseEmbeddingDriver:
    return DummyEmbeddingDriver()
```

image_generation_driver()

Source Code in griptape/configs/drivers/drivers_config.py

```
@lazy_property()
def image_generation_driver(self) -> BaseImageGenerationDriver:
    return DummyImageGenerationDriver()
```

prompt_driver()

Source Code in griptape/configs/drivers/drivers_config.py

```
@lazy_property()
def prompt_driver(self) -> BasePromptDriver:
    return DummyPromptDriver()
```

ruleset_driver()

Source Code in griptape/configs/drivers/drivers_config.py

```
@lazy_property()
def ruleset_driver(self) -> BaseRulesetDriver:
    return LocalRulesetDriver()
```

text_to_speech_driver()

Source Code in griptape/configs/drivers/drivers_config.py

```
@lazy_property()
def text_to_speech_driver(self) -> BaseTextToSpeechDriver:
    return DummyTextToSpeechDriver()
```

vector_store_driver()

Source Code in griptape/configs/drivers/drivers_config.py

```
@lazy_property()
def vector_store_driver(self) -> BaseVectorStoreDriver:
    return DummyVectorStoreDriver(embedding_driver=self.embedding_driver)
```

GoogleDriversConfig

Bases: *AnthropicDriversConfig* “)

Source Code in griptape/configs/drivers/google_drivers_config.py

```
@define
class GoogleDriversConfig(DriversConfig):
    @lazy_property()
    def prompt_driver(self) -> GooglePromptDriver:
        return GooglePromptDriver(model="gemini-2.0-flash")

    @lazy_property()
    def embedding_driver(self) -> GoogleEmbeddingDriver:
        return GoogleEmbeddingDriver(model="models/embedding-004")

    @lazy_property()
    def vector_store_driver(self) -> LocalVectorStoreDriver:
        return LocalVectorStoreDriver(embedding_driver=GoogleEmbeddingDriver(model=
↪"models/embedding-004"))
```

embedding_driver()

Source Code in griptape/configs/drivers/google_drivers_config.py

```
@lazy_property()
def embedding_driver(self) -> GoogleEmbeddingDriver:
    return GoogleEmbeddingDriver(model="models/embedding-004")
```

prompt_driver()

Source Code in griptape/configs/drivers/google_drivers_config.py

```
@lazy_property()
def prompt_driver(self) -> GooglePromptDriver:
    return GooglePromptDriver(model="gemini-2.0-flash")
```

vector_store_driver()

Source Code in griptape/configs/drivers/google_drivers_config.py

```
@lazy_property()
def vector_store_driver(self) -> LocalVectorStoreDriver:
    return LocalVectorStoreDriver(embedding_driver=GoogleEmbeddingDriver(model=
↪"models/embedding-004"))
```

OpenAiDriversConfig

Bases: *AnthropicDriversConfig* “)

Source Code in griptape/configs/drivers/openai_drivers_config.py

```
@define
class OpenAiDriversConfig(DriversConfig):
    @lazy_property()
    def prompt_driver(self) -> OpenAiChatPromptDriver:
        return OpenAiChatPromptDriver(model="gpt-4.1")

    @lazy_property()
    def image_generation_driver(self) -> OpenAiImageGenerationDriver:
        return OpenAiImageGenerationDriver(model="dall-e-2", image_size="512x512")

    @lazy_property()
    def embedding_driver(self) -> OpenAiEmbeddingDriver:
        return OpenAiEmbeddingDriver(model="text-embedding-3-small")

    @lazy_property()
    def vector_store_driver(self) -> LocalVectorStoreDriver:
        return LocalVectorStoreDriver(embedding_driver=OpenAiEmbeddingDriver(model=
↪"text-embedding-3-small"))

    @lazy_property()
    def text_to_speech_driver(self) -> OpenAiTextToSpeechDriver:
        return OpenAiTextToSpeechDriver(model="tts-1")

    @lazy_property()
    def audio_transcription_driver(self) -> OpenAiAudioTranscriptionDriver:
        return OpenAiAudioTranscriptionDriver(model="whisper-1")
```

audio_transcription_driver()

Source Code in griptape/configs/drivers/openai_drivers_config.py

```
@lazy_property()
def audio_transcription_driver(self) -> OpenAiAudioTranscriptionDriver:
    return OpenAiAudioTranscriptionDriver(model="whisper-1")
```

embedding_driver()

Source Code in griptape/configs/drivers/openai_drivers_config.py

```
@lazy_property()
def embedding_driver(self) -> OpenAiEmbeddingDriver:
    return OpenAiEmbeddingDriver(model="text-embedding-3-small")
```

image_generation_driver()

Source Code in griptape/configs/drivers/openai_drivers_config.py

```
@lazy_property()
def image_generation_driver(self) -> OpenAiImageGenerationDriver:
    return OpenAiImageGenerationDriver(model="dall-e-2", image_size="512x512")
```

prompt_driver()

Source Code in griptape/configs/drivers/openai_drivers_config.py

```
@lazy_property()
def prompt_driver(self) -> OpenAiChatPromptDriver:
    return OpenAiChatPromptDriver(model="gpt-4.1")
```

text_to_speech_driver()

Source Code in griptape/configs/drivers/openai_drivers_config.py

```
@lazy_property()
def text_to_speech_driver(self) -> OpenAiTextToSpeechDriver:
    return OpenAiTextToSpeechDriver(model="tts-1")
```

vector_store_driver()

Source Code in griptape/configs/drivers/openai_drivers_config.py

```
@lazy_property()
def vector_store_driver(self) -> LocalVectorStoreDriver:
    return LocalVectorStoreDriver(embedding_driver=OpenAiEmbeddingDriver(model="text-
↪embedding-3-small"))
```

amazon_bedrock_drivers_config

Bases: *AnthropicDriversConfig* “)

Source Code in griptape/configs/drivers/amazon_bedrock_drivers_config.py

```
@define
class AmazonBedrockDriversConfig(DriversConfig):
    session: boto3.Session = field(
        default=Factory(lambda: import_optional_dependency("boto3").Session()),
        kw_only=True,
```

(continues on next page)

(continued from previous page)

```
        metadata={"serializable": False},
    )

    @lazy_property()
    def prompt_driver(self) -> AmazonBedrockPromptDriver:
        return AmazonBedrockPromptDriver(session=self.session, model="anthropic.
↪claude-3-7-sonnet-20250219-v1:0")

    @lazy_property()
    def embedding_driver(self) -> AmazonBedrockTitanEmbeddingDriver:
        return AmazonBedrockTitanEmbeddingDriver(session=self.session, model="amazon.
↪titan-embed-text-v2:0")

    @lazy_property()
    def image_generation_driver(self) -> AmazonBedrockImageGenerationDriver:
        return AmazonBedrockImageGenerationDriver(
            session=self.session,
            model="amazon.titan-image-generator-v2:0",
            image_generation_model_driver=BedrockTitanImageGenerationModelDriver(),
        )

    @lazy_property()
    def vector_store_driver(self) -> LocalVectorStoreDriver:
        return LocalVectorStoreDriver(
            embedding_driver=AmazonBedrockTitanEmbeddingDriver(
                session=self.session, model="amazon.titan-embed-text-v2:0"
            )
        )
```

- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute

embedding_driver()

Source Code in griptape/configs/drivers/amazon_bedrock_drivers_config.py

```
@lazy_property()
def embedding_driver(self) -> AmazonBedrockTitanEmbeddingDriver:
    return AmazonBedrockTitanEmbeddingDriver(session=self.session, model="amazon.
↪titan-embed-text-v2:0")
```

image_generation_driver()

Source Code in griptape/configs/drivers/amazon_bedrock_drivers_config.py

```
@lazy_property()
def image_generation_driver(self) -> AmazonBedrockImageGenerationDriver:
    return AmazonBedrockImageGenerationDriver(
        session=self.session,
        model="amazon.titan-image-generator-v2:0",
```

(continues on next page)

(continued from previous page)

```
        image_generation_model_driver=BedrockTitanImageGenerationModelDriver(),
    )
```

prompt_driver()

Source Code in griptape/configs/drivers/amazon_bedrock_drivers_config.py

```
@lazy_property()
def prompt_driver(self) -> AmazonBedrockPromptDriver:
    return AmazonBedrockPromptDriver(session=self.session, model="anthropic.claude-3-
↪7-sonnet-20250219-v1:0")
```

vector_store_driver()

Source Code in griptape/configs/drivers/amazon_bedrock_drivers_config.py

```
@lazy_property()
def vector_store_driver(self) -> LocalVectorStoreDriver:
    return LocalVectorStoreDriver(
        embedding_driver=AmazonBedrockTitanEmbeddingDriver(
            session=self.session, model="amazon.titan-embed-text-v2:0"
        )
    )
```

anthropic_drivers_config

Bases: *AnthropicDriversConfig* “)

Source Code in griptape/configs/drivers/anthropic_drivers_config.py

```
@define
class AnthropicDriversConfig(DriversConfig):
    @lazy_property()
    def prompt_driver(self) -> AnthropicPromptDriver:
        return AnthropicPromptDriver(model="claude-3-7-sonnet-latest")
```

prompt_driver()

Source Code in griptape/configs/drivers/anthropic_drivers_config.py

```
@lazy_property()
def prompt_driver(self) -> AnthropicPromptDriver:
    return AnthropicPromptDriver(model="claude-3-7-sonnet-latest")
```

azure_openai_drivers_config

Bases: *AnthropicDriversConfig* “)

Attributes

Name	Type	Description
<a href="azure_openai_drivers_config.mdx#griptape.configs.drivers.azure_openai_drivers_config.AzureOpenAiDriversConfig.azure_endpoint" title="azure_endpoint = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.configs.drivers.azure_openai_drivers_config.AzureOpenAiDriversConfig.azure_endpoint)"><code class="language-python">	*str*	The endpoint for the Azure OpenAI instance.
<a href="azure_openai_drivers_config.mdx#griptape.configs.drivers.azure_openai_drivers_config.AzureOpenAiDriversConfig.azure_ad_token" title="azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.configs.drivers.azure_openai_drivers_config.AzureOpenAiDriversConfig.azure_ad_token)"><code class="language-python">	*Optional[str]*	An optional Azure Active Directory token.

continues on next page

Table 36 – continued from previous page

Name	Type	Description
<a href="azure_openai_drivers_config.mdx#griptape.configs.drivers.azure_openai_drivers_config.AzureOpenAiDriversConfig.azure_ad_token_provider" title="azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.configs.drivers.azure_openai_drivers_config.AzureOpenAiDriversConfig.azure_ad_token_provider)"><code class="language-python">	*Optional[Callable[[], str]]*	An optional Azure Active Directory token provider.
<a href="azure_openai_drivers_config.mdx#griptape.configs.drivers.azure_openai_drivers_config.AzureOpenAiDriversConfig.api_key" title="api_key = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.configs.drivers.azure_openai_drivers_config.AzureOpenAiDriversConfig.api_key)"><code class="language-python">	*Optional[str]*	An optional Azure API key.

continues on next page

Table 36 – continued from previous page

Name	Type	Description
<a href="azure_openai_drivers_config.mdx#griptape.configs.drivers.azure_openai_drivers_config.AzureOpenAiDriversConfig.prompt_driver" title="prompt_driver() (griptape.configs.drivers.azure_openai_drivers_config.AzureOpenAiDriversConfig.prompt_driver)"><coc class="language-python">	<a href="../../drivers/prompt/openai.mdx#griptape.drivers.prompt.openai.AzureOpenAiChatPromptDriver" title="AzureOpenAiChatPromptDriver (griptape.drivers.prompt.openai.AzureOpenAiChatPromptDriver)"><code class="language-python">AzureOpenAiChatPromptDr	An Azure OpenAI Chat Prompt Driver.
<a href="azure_openai_drivers_config.mdx#griptape.configs.drivers.azure_openai_drivers_config.AzureOpenAiDriversConfig.image_generation_driver" title="image_generation_driver() (griptape.configs.drivers.azure_openai_drivers_config.AzureOpenAiDriversConfig.image_generation_driver)"><code class="language-python">	<a href="../../drivers/image_generation/openai.mdx#griptape.drivers.image_generation.openai.AzureOpenAiImageGenerationDriver" title="AzureOpenAiImageGenerationDriver (griptape.drivers.image_generation.openai.AzureOpenAiImageGenerationDriver)"><code class="language-python">AzureOpenAiImageGenerat	An Azure OpenAI Image Generation Driver.

continues on next page

Table 36 – continued from previous page

Name	Type	Description
<a href="azure_openai_drivers_config.mdx#griptape.configs.drivers.azure_openai_drivers_config.AzureOpenAiDriversConfig.embedding_driver" title="embedding_driver() (griptape.configs.drivers.azure_openai_drivers_config.AzureOpenAiDriversConfig.embedding_driver)"><code class="language-python">	<a href="../../drivers/embedding/openai.mdx#griptape.drivers.embedding.openai.AzureOpenAiEmbeddingDriver" title="AzureOpenAiEmbeddingDriver (griptape.drivers.embedding.openai.AzureOpenAiEmbeddingDriver)"><code class="language-python">AzureOpenAiEmbeddingDr	An Azure OpenAI Embedding Driver.
<a href="azure_openai_drivers_config.mdx#griptape.configs.drivers.azure_openai_drivers_config.AzureOpenAiDriversConfig.vector_store_driver" title="vector_store_driver() (griptape.configs.drivers.azure_openai_drivers_config.AzureOpenAiDriversConfig.vector_store_driver)"><code class="language-python">	<a href="../../drivers/vector/local.mdx#griptape.drivers.vector.local.LocalVectorStoreDriver" title="LocalVectorStoreDriver (griptape.drivers.vector.local.LocalVectorStoreDriver)"><code class="language-python">LocalVectorStoreDriver</co	A Local Vector Store Driver.

Source Code in griptape/configs/drivers/azure_openai_drivers_config.py

```
@define
class AzureOpenAiDriversConfig(DriversConfig):
    """Azure OpenAI Drivers Configuration.

    Attributes:
        azure_endpoint: The endpoint for the Azure OpenAI instance.
        azure_ad_token: An optional Azure Active Directory token.
        azure_ad_token_provider: An optional Azure Active Directory token provider.
        api_key: An optional Azure API key.
        prompt_driver: An Azure OpenAI Chat Prompt Driver.
        image_generation_driver: An Azure OpenAI Image Generation Driver.
        embedding_driver: An Azure OpenAI Embedding Driver.
```

(continues on next page)

(continued from previous page)

```
        vector_store_driver: A Local Vector Store Driver.
    """

    azure_endpoint: str = field(kw_only=True, metadata={"serializable": True})
    azure_ad_token: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": False})
    azure_ad_token_provider: Optional[Callable[[], str]] = field(
        kw_only=True,
        default=None,
        metadata={"serializable": False},
    )
    api_key: Optional[str] = field(kw_only=True, default=None, metadata={"serializable
↪": False})

    @lazy_property()
    def prompt_driver(self) -> AzureOpenAiChatPromptDriver:
        return AzureOpenAiChatPromptDriver(
            model="gpt-4.1",
            azure_endpoint=self.azure_endpoint,
            api_key=self.api_key,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )

    @lazy_property()
    def embedding_driver(self) -> AzureOpenAiEmbeddingDriver:
        return AzureOpenAiEmbeddingDriver(
            model="text-embedding-3-small",
            azure_endpoint=self.azure_endpoint,
            api_key=self.api_key,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )

    @lazy_property()
    def image_generation_driver(self) -> AzureOpenAiImageGenerationDriver:
        return AzureOpenAiImageGenerationDriver(
            model="dall-e-2",
            azure_endpoint=self.azure_endpoint,
            api_key=self.api_key,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
            image_size="512x512",
        )

    @lazy_property()
    def vector_store_driver(self) -> LocalVectorStoreDriver:
        return LocalVectorStoreDriver(
            embedding_driver=AzureOpenAiEmbeddingDriver(
                model="text-embedding-3-small",
                azure_endpoint=self.azure_endpoint,
                api_key=self.api_key,
```

(continues on next page)

(continued from previous page)

```
                azure_ad_token=self.azure_ad_token,
                azure_ad_token_provider=self.azure_ad_token_provider,
            )
        )

    @lazy_property()
    def text_to_speech_driver(self) -> AzureOpenAiTextToSpeechDriver:
        return AzureOpenAiTextToSpeechDriver(
            model="tts",
            azure_endpoint=self.azure_endpoint,
            api_key=self.api_key,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )
```

- `api_key = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

embedding_driver()

Source Code in griptape/configs/drivers/azure_openai_drivers_config.py

```
@lazy_property()
def embedding_driver(self) -> AzureOpenAiEmbeddingDriver:
    return AzureOpenAiEmbeddingDriver(
        model="text-embedding-3-small",
        azure_endpoint=self.azure_endpoint,
        api_key=self.api_key,
        azure_ad_token=self.azure_ad_token,
        azure_ad_token_provider=self.azure_ad_token_provider,
    )
```

image_generation_driver()

Source Code in griptape/configs/drivers/azure_openai_drivers_config.py

```
@lazy_property()
def image_generation_driver(self) -> AzureOpenAiImageGenerationDriver:
    return AzureOpenAiImageGenerationDriver(
        model="dall-e-2",
        azure_endpoint=self.azure_endpoint,
        api_key=self.api_key,
        azure_ad_token=self.azure_ad_token,
        azure_ad_token_provider=self.azure_ad_token_provider,
```

(continues on next page)

(continued from previous page)

```
        image_size="512x512",
    )
```

prompt_driver()

Source Code in griptape/configs/drivers/azure_openai_drivers_config.py

```
@lazy_property()
def prompt_driver(self) -> AzureOpenAiChatPromptDriver:
    return AzureOpenAiChatPromptDriver(
        model="gpt-4.1",
        azure_endpoint=self.azure_endpoint,
        api_key=self.api_key,
        azure_ad_token=self.azure_ad_token,
        azure_ad_token_provider=self.azure_ad_token_provider,
    )
```

text_to_speech_driver()

Source Code in griptape/configs/drivers/azure_openai_drivers_config.py

```
@lazy_property()
def text_to_speech_driver(self) -> AzureOpenAiTextToSpeechDriver:
    return AzureOpenAiTextToSpeechDriver(
        model="tts",
        azure_endpoint=self.azure_endpoint,
        api_key=self.api_key,
        azure_ad_token=self.azure_ad_token,
        azure_ad_token_provider=self.azure_ad_token_provider,
    )
```

vector_store_driver()

Source Code in griptape/configs/drivers/azure_openai_drivers_config.py

```
@lazy_property()
def vector_store_driver(self) -> LocalVectorStoreDriver:
    return LocalVectorStoreDriver(
        embedding_driver=AzureOpenAiEmbeddingDriver(
            model="text-embedding-3-small",
            azure_endpoint=self.azure_endpoint,
            api_key=self.api_key,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )
    )
```

base_drivers_config

Bases:

ABC, *SerializableMixin* “)

Source Code in griptape/configs/drivers/base_drivers_config.py

```
@define
class BaseDriversConfig(ABC, SerializableMixin):
    _prompt_driver: Optional[BasePromptDriver] = field(
        kw_only=True, default=None, metadata={"serializable": True}, alias="prompt_
↪driver"
    )
    _image_generation_driver: Optional[BaseImageGenerationDriver] = field(
        kw_only=True, default=None, metadata={"serializable": True}, alias="image_
↪generation_driver"
    )
    _embedding_driver: Optional[BaseEmbeddingDriver] = field(
        kw_only=True, default=None, metadata={"serializable": True}, alias="embedding_
↪driver"
    )
    _vector_store_driver: Optional[BaseVectorStoreDriver] = field(
        default=None, kw_only=True, metadata={"serializable": True}, alias="vector_
↪store_driver"
    )
    _conversation_memory_driver: Optional[BaseConversationMemoryDriver] = field(
        default=None, kw_only=True, metadata={"serializable": True}, alias=
↪"conversation_memory_driver"
    )
    _text_to_speech_driver: Optional[BaseTextToSpeechDriver] = field(
        default=None, kw_only=True, metadata={"serializable": True}, alias="text_to_
↪speech_driver"
    )
    _audio_transcription_driver: Optional[BaseAudioTranscriptionDriver] = field(
        default=None, kw_only=True, metadata={"serializable": True}, alias="audio_
↪transcription_driver"
    )
    _ruleset_driver: Optional[BaseRulesetDriver] = field(
        default=None, kw_only=True, metadata={"serializable": True}, alias="ruleset_
↪driver"
    )

    _last_drivers_config: Optional[BaseDriversConfig] = field(default=None)

    def __enter__(self) -> Self:
        from griptape.configs import Defaults

        self._last_drivers_config = Defaults.drivers_config

        Defaults.drivers_config = self

        return self

    def __exit__(self, type, value, traceback) -> None:  # noqa: ANN001, A002
```

(continues on next page)

(continued from previous page)

```
        from griptape.configs import Defaults

        if self._last_drivers_config is not None:
            Defaults.drivers_config = self._last_drivers_config

        self._last_drivers_config = None

    @lazy_property()
    @abstractmethod
    def prompt_driver(self) -> BasePromptDriver: ...

    @lazy_property()
    @abstractmethod
    def image_generation_driver(self) -> BaseImageGenerationDriver: ...

    @lazy_property()
    @abstractmethod
    def embedding_driver(self) -> BaseEmbeddingDriver: ...

    @lazy_property()
    @abstractmethod
    def vector_store_driver(self) -> BaseVectorStoreDriver: ...

    @lazy_property()
    @abstractmethod
    def conversation_memory_driver(self) -> BaseConversationMemoryDriver: ...

    @lazy_property()
    @abstractmethod
    def text_to_speech_driver(self) -> BaseTextToSpeechDriver: ...

    @lazy_property()
    @abstractmethod
    def audio_transcription_driver(self) -> BaseAudioTranscriptionDriver: ...

    @lazy_property()
    @abstractmethod
    def ruleset_driver(self) -> BaseRulesetDriver: ...
```

- `_audio_transcription_driver = field(default=None, kw_only=True, metadata={'serializable': True}, alias='audio_transcription_driver')` class-attribute instance-attribute
- `_conversation_memory_driver = field(default=None, kw_only=True, metadata={'serializable': True}, alias='conversation_memory_driver')` class-attribute instance-attribute
- `_embedding_driver = field(kw_only=True, default=None, metadata={'serializable': True}, alias='embedding_driver')` class-attribute instance-attribute
- `_image_generation_driver = field(kw_only=True, default=None, metadata={'serializable': True}, alias='image_generation_driver')` class-attribute instance-attribute
- `_last_drivers_config = field(default=None)` class-attribute instance-attribute

- `_prompt_driver = field(kw_only=True, default=None, metadata={'serializable': True}, alias='prompt_driver')` class-attribute instance-attribute
- `_ruleset_driver = field(default=None, kw_only=True, metadata={'serializable': True}, alias='ruleset_driver')` class-attribute instance-attribute
- `_text_to_speech_driver = field(default=None, kw_only=True, metadata={'serializable': True}, alias='text_to_speech_driver')` class-attribute instance-attribute
- `_vector_store_driver = field(default=None, kw_only=True, metadata={'serializable': True}, alias='vector_store_driver')` class-attribute instance-attribute

enter()

Source Code in griptape/configs/drivers/base_drivers_config.py

```
def __enter__(self) -> Self:
    from griptape.configs import Defaults

    self._last_drivers_config = Defaults.drivers_config

    Defaults.drivers_config = self

    return self
```

exit(type, value, traceback)

Source Code in griptape/configs/drivers/base_drivers_config.py

```
def __exit__(self, type, value, traceback) -> None:  # noqa: ANN001, A002
    from griptape.configs import Defaults

    if self._last_drivers_config is not None:
        Defaults.drivers_config = self._last_drivers_config

    self._last_drivers_config = None
```

audio_transcription_driver()abstractmethod

Source Code in griptape/configs/drivers/base_drivers_config.py

```
@lazy_property()
@abstractmethod
def audio_transcription_driver(self) -> BaseAudioTranscriptionDriver: ...
```

conversation_memory_driver()abstractmethod

Source Code in griptape/configs/drivers/base_drivers_config.py

```
@lazy_property()
@abstractmethod
def conversation_memory_driver(self) -> BaseConversationMemoryDriver: ...
```

embedding_driver()abstractmethod

Source Code in griptape/configs/drivers/base_drivers_config.py

```
@lazy_property()
@abstractmethod
def embedding_driver(self) -> BaseEmbeddingDriver: ...
```

image_generation_driver()abstractmethod

Source Code in griptape/configs/drivers/base_drivers_config.py

```
@lazy_property()
@abstractmethod
def image_generation_driver(self) -> BaseImageGenerationDriver: ...
```

prompt_driver()abstractmethod

Source Code in griptape/configs/drivers/base_drivers_config.py

```
@lazy_property()
@abstractmethod
def prompt_driver(self) -> BasePromptDriver: ...
```

ruleset_driver()abstractmethod

Source Code in griptape/configs/drivers/base_drivers_config.py

```
@lazy_property()
@abstractmethod
def ruleset_driver(self) -> BaseRulesetDriver: ...
```

text_to_speech_driver()abstractmethod

Source Code in griptape/configs/drivers/base_drivers_config.py

```
@lazy_property()
@abstractmethod
def text_to_speech_driver(self) -> BaseTextToSpeechDriver: ...
```

vector_store_driver()abstractmethod

Source Code in griptape/configs/drivers/base_drivers_config.py

```
@lazy_property()
@abstractmethod
def vector_store_driver(self) -> BaseVectorStoreDriver: ...
```

cohere_drivers_config

Bases: *AnthropicDriversConfig* “)

Source Code in griptape/configs/drivers/cohere_drivers_config.py

```
@define
class CohereDriversConfig(DriversConfig):
    api_key: str = field(metadata={"serializable": False}, kw_only=True)

    @lazy_property()
    def prompt_driver(self) -> CoherePromptDriver:
        return CoherePromptDriver(model="command-r", api_key=self.api_key)

    @lazy_property()
    def embedding_driver(self) -> CohereEmbeddingDriver:
        return CohereEmbeddingDriver(
            model="embed-english-v3.0",
            api_key=self.api_key,
            input_type="search_document",
        )

    @lazy_property()
    def vector_store_driver(self) -> LocalVectorStoreDriver:
        return LocalVectorStoreDriver(
            embedding_driver=CohereEmbeddingDriver(
                model="embed-english-v3.0",
                api_key=self.api_key,
                input_type="search_document",
            )
        )
```

- `api_key = field(metadata={'serializable': False}, kw_only=True)` class-attribute instance-attribute

embedding_driver()

Source Code in griptape/configs/drivers/cohere_drivers_config.py

```
@lazy_property()
def embedding_driver(self) -> CohereEmbeddingDriver:
    return CohereEmbeddingDriver(
        model="embed-english-v3.0",
        api_key=self.api_key,
        input_type="search_document",
    )
```

prompt_driver()

Source Code in griptape/configs/drivers/cohere_drivers_config.py

```
@lazy_property()
def prompt_driver(self) -> CoherePromptDriver:
    return CoherePromptDriver(model="command-r", api_key=self.api_key)
```

vector_store_driver()

Source Code in griptape/configs/drivers/cohere_drivers_config.py

```
@lazy_property()
def vector_store_driver(self) -> LocalVectorStoreDriver:
    return LocalVectorStoreDriver(
        embedding_driver=CohereEmbeddingDriver(
            model="embed-english-v3.0",
            api_key=self.api_key,
            input_type="search_document",
        )
    )
```

drivers_config

Bases: *BaseDriversConfig* “)

Source Code in griptape/configs/drivers/drivers_config.py

```
@define
class DriversConfig(BaseDriversConfig):
    @lazy_property()
    def prompt_driver(self) -> BasePromptDriver:
        return DummyPromptDriver()

    @lazy_property()
    def image_generation_driver(self) -> BaseImageGenerationDriver:
        return DummyImageGenerationDriver()

    @lazy_property()
    def embedding_driver(self) -> BaseEmbeddingDriver:
        return DummyEmbeddingDriver()

    @lazy_property()
    def vector_store_driver(self) -> BaseVectorStoreDriver:
        return DummyVectorStoreDriver(embedding_driver=self.embedding_driver)

    @lazy_property()
    def conversation_memory_driver(self) -> BaseConversationMemoryDriver:
        return LocalConversationMemoryDriver()

    @lazy_property()
    def text_to_speech_driver(self) -> BaseTextToSpeechDriver:
        return DummyTextToSpeechDriver()

    @lazy_property()
    def audio_transcription_driver(self) -> BaseAudioTranscriptionDriver:
        return DummyAudioTranscriptionDriver()

    @lazy_property()
    def ruleset_driver(self) -> BaseRulesetDriver:
        return LocalRulesetDriver()
```

audio_transcription_driver()

Source Code in griptape/configs/drivers/drivers_config.py

```
@lazy_property()
def audio_transcription_driver(self) -> BaseAudioTranscriptionDriver:
    return DummyAudioTranscriptionDriver()
```

conversation_memory_driver()

Source Code in griptape/configs/drivers/drivers_config.py

```
@lazy_property()
def conversation_memory_driver(self) -> BaseConversationMemoryDriver:
    return LocalConversationMemoryDriver()
```

embedding_driver()

Source Code in griptape/configs/drivers/drivers_config.py

```
@lazy_property()
def embedding_driver(self) -> BaseEmbeddingDriver:
    return DummyEmbeddingDriver()
```

image_generation_driver()

Source Code in griptape/configs/drivers/drivers_config.py

```
@lazy_property()
def image_generation_driver(self) -> BaseImageGenerationDriver:
    return DummyImageGenerationDriver()
```

prompt_driver()

Source Code in griptape/configs/drivers/drivers_config.py

```
@lazy_property()
def prompt_driver(self) -> BasePromptDriver:
    return DummyPromptDriver()
```

ruleset_driver()

Source Code in griptape/configs/drivers/drivers_config.py

```
@lazy_property()
def ruleset_driver(self) -> BaseRulesetDriver:
    return LocalRulesetDriver()
```

text_to_speech_driver()

Source Code in griptape/configs/drivers/drivers_config.py

```
@lazy_property()
def text_to_speech_driver(self) -> BaseTextToSpeechDriver:
    return DummyTextToSpeechDriver()
```

vector_store_driver()

Source Code in griptape/configs/drivers/drivers_config.py

```
@lazy_property()
def vector_store_driver(self) -> BaseVectorStoreDriver:
    return DummyVectorStoreDriver(embedding_driver=self.embedding_driver)
```

google_drivers_config

Bases: *AnthropicDriversConfig* “)

Source Code in griptape/configs/drivers/google_drivers_config.py

```
@define
class GoogleDriversConfig(DriversConfig):
    @lazy_property()
    def prompt_driver(self) -> GooglePromptDriver:
        return GooglePromptDriver(model="gemini-2.0-flash")

    @lazy_property()
    def embedding_driver(self) -> GoogleEmbeddingDriver:
        return GoogleEmbeddingDriver(model="models/embedding-004")

    @lazy_property()
    def vector_store_driver(self) -> LocalVectorStoreDriver:
        return LocalVectorStoreDriver(embedding_driver=GoogleEmbeddingDriver(model=
↪"models/embedding-004"))
```

embedding_driver()

Source Code in griptape/configs/drivers/google_drivers_config.py

```
@lazy_property()
def embedding_driver(self) -> GoogleEmbeddingDriver:
    return GoogleEmbeddingDriver(model="models/embedding-004")
```

prompt_driver()

Source Code in griptape/configs/drivers/google_drivers_config.py

```
@lazy_property()
def prompt_driver(self) -> GooglePromptDriver:
    return GooglePromptDriver(model="gemini-2.0-flash")
```

vector_store_driver()

Source Code in griptape/configs/drivers/google_drivers_config.py

```
@lazy_property()
def vector_store_driver(self) -> LocalVectorStoreDriver:
    return LocalVectorStoreDriver(embedding_driver=GoogleEmbeddingDriver(model=
↪"models/embedding-004"))
```

openai_drivers_config

Bases: *AnthropicDriversConfig* “)

Source Code in griptape/configs/drivers/openai_drivers_config.py

```
@define
class OpenAiDriversConfig(DriversConfig):
    @lazy_property()
    def prompt_driver(self) -> OpenAiChatPromptDriver:
        return OpenAiChatPromptDriver(model="gpt-4.1")

    @lazy_property()
    def image_generation_driver(self) -> OpenAiImageGenerationDriver:
        return OpenAiImageGenerationDriver(model="dall-e-2", image_size="512x512")

    @lazy_property()
    def embedding_driver(self) -> OpenAiEmbeddingDriver:
        return OpenAiEmbeddingDriver(model="text-embedding-3-small")

    @lazy_property()
    def vector_store_driver(self) -> LocalVectorStoreDriver:
        return LocalVectorStoreDriver(embedding_driver=OpenAiEmbeddingDriver(model=
↪"text-embedding-3-small"))

    @lazy_property()
    def text_to_speech_driver(self) -> OpenAiTextToSpeechDriver:
        return OpenAiTextToSpeechDriver(model="tts-1")

    @lazy_property()
    def audio_transcription_driver(self) -> OpenAiAudioTranscriptionDriver:
        return OpenAiAudioTranscriptionDriver(model="whisper-1")
```

audio_transcription_driver()

Source Code in griptape/configs/drivers/openai_drivers_config.py

```
@lazy_property()
def audio_transcription_driver(self) -> OpenAiAudioTranscriptionDriver:
    return OpenAiAudioTranscriptionDriver(model="whisper-1")
```

embedding_driver()

Source Code in griptape/configs/drivers/openai_drivers_config.py

```
@lazy_property()
def embedding_driver(self) -> OpenAiEmbeddingDriver:
    return OpenAiEmbeddingDriver(model="text-embedding-3-small")
```

image_generation_driver()

Source Code in griptape/configs/drivers/openai_drivers_config.py

```
@lazy_property()
def image_generation_driver(self) -> OpenAiImageGenerationDriver:
    return OpenAiImageGenerationDriver(model="dall-e-2", image_size="512x512")
```

prompt_driver()

Source Code in griptape/configs/drivers/openai_drivers_config.py

```
@lazy_property()
def prompt_driver(self) -> OpenAiChatPromptDriver:
    return OpenAiChatPromptDriver(model="gpt-4.1")
```

text_to_speech_driver()

Source Code in griptape/configs/drivers/openai_drivers_config.py

```
@lazy_property()
def text_to_speech_driver(self) -> OpenAiTextToSpeechDriver:
    return OpenAiTextToSpeechDriver(model="tts-1")
```

vector_store_driver()

Source Code in griptape/configs/drivers/openai_drivers_config.py

```
@lazy_property()
def vector_store_driver(self) -> LocalVectorStoreDriver:
    return LocalVectorStoreDriver(embedding_driver=OpenAiEmbeddingDriver(model="text-
↪embedding-3-small"))
```

logging

- `__all__ = ['JsonFormatter', 'LoggingConfig', 'NewlineLoggingFilter', 'TruncateLoggingFilter']` module-attribute

Bases:

Formatter

Source Code in griptape/configs/logging/json_formatter.py

```
@define
class JsonFormatter(logging.Formatter):
    indent: int = field(default=2, kw_only=True)

    def __attrs_pre_init__(self) -> None:
        super().__init__()

    def format(self, record: Any) -> str:
        if isinstance(record.msg, dict):
            record.msg = json.dumps(record.msg, indent=self.indent)
        elif isinstance(record.msg, (list, tuple)):
            record.msg = json.dumps(list(record.msg), indent=self.indent)

        return super().format(record)
```

- `indent = field(default=2, kw_only=True)` class-attribute instance-attribute

attrs_pre_init()

Source Code in griptape/configs/logging/json_formatter.py

```
def __attrs_pre_init__(self) -> None:
    super().__init__()
```

format(record)

Source Code in griptape/configs/logging/json_formatter.py

```
def format(self, record: Any) -> str:
    if isinstance(record.msg, dict):
        record.msg = json.dumps(record.msg, indent=self.indent)
    elif isinstance(record.msg, (list, tuple)):
        record.msg = json.dumps(list(record.msg), indent=self.indent)

    return super().format(record)
```

LoggingConfig

Source Code in griptape/configs/logging/logging_config.py

```
@define
class LoggingConfig:
    logger_name: str = field(default="griptape", kw_only=True)

    def __attrs_post_init__(self) -> None:
        logger = logging.getLogger(self.logger_name)
        logger.setLevel(logging.INFO)
        logger.propagate = False
        logger.addHandler(RichHandler(show_time=True, show_path=False))
```

- `logger_name = field(default='griptape', kw_only=True)` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/configs/logging/logging_config.py

```
def __attrs_post_init__(self) -> None:
    logger = logging.getLogger(self.logger_name)
    logger.setLevel(logging.INFO)
    logger.propagate = False
    logger.addHandler(RichHandler(show_time=True, show_path=False))
```

NewlineLoggingFilter

Bases:

Filter

Source Code in griptape/configs/logging/newline_logging_filter.py

```
@define
class NewlineLoggingFilter(logging.Filter):
    replace_str: str = field(default=" ", kw_only=True)

    def filter(self, record: Any) -> bool:
        record.msg = record.msg.replace("\n", self.replace_str)
        return True
```

- `replace_str = field(default=' ', kw_only=True)` class-attribute instance-attribute

filter(record)

Source Code in griptape/configs/logging/newline_logging_filter.py

```
def filter(self, record: Any) -> bool:
    record.msg = record.msg.replace("\n", self.replace_str)
    return True
```

TruncateLoggingFilter

Bases:

Filter

Source Code in griptape/configs/logging/truncate_logging_filter.py

```
@define
class TruncateLoggingFilter(logging.Filter):
    max_log_length: int = field(default=1000, kw_only=True)

    def filter(self, record: Any) -> bool:
        message = record.getMessage()

        if len(message) > self.max_log_length:
            record.msg = f"{message[: self.max_log_length]}... [{len(message) - self.
↪max_log_length} more characters]"
            record.args = ()
        return True
```

- `max_log_length = field(default=1000, kw_only=True)` class-attribute instance-attribute

filter(record)

Source Code in griptape/configs/logging/truncate_logging_filter.py

```
def filter(self, record: Any) -> bool:
    message = record.getMessage()

    if len(message) > self.max_log_length:
        record.msg = f"{message[: self.max_log_length]}... [{len(message) - self.max_
↪log_length} more characters]"
        record.args = ()
    return True
```

json_formatter

Bases:

Formatter

Source Code in griptape/configs/logging/json_formatter.py

```
@define
class JsonFormatter(logging.Formatter):
    indent: int = field(default=2, kw_only=True)

    def __attrs_pre_init__(self) -> None:
        super().__init__()

    def format(self, record: Any) -> str:
        if isinstance(record.msg, dict):
            record.msg = json.dumps(record.msg, indent=self.indent)
        elif isinstance(record.msg, (list, tuple)):
            record.msg = json.dumps(list(record.msg), indent=self.indent)

        return super().format(record)
```

- `indent = field(default=2, kw_only=True)` class-attribute instance-attribute

attrs_pre_init()

Source Code in griptape/configs/logging/json_formatter.py

```
def __attrs_pre_init__(self) -> None:
    super().__init__()
```

format(record)

Source Code in griptape/configs/logging/json_formatter.py

```
def format(self, record: Any) -> str:
    if isinstance(record.msg, dict):
        record.msg = json.dumps(record.msg, indent=self.indent)
```

(continues on next page)

(continued from previous page)

```
    elif isinstance(record.msg, (list, tuple)):
        record.msg = json.dumps(list(record.msg), indent=self.indent)

    return super().format(record)
```

logging_config

Source Code in griptape/configs/logging/logging_config.py

```
@define
class LoggingConfig:
    logger_name: str = field(default="griptape", kw_only=True)

    def __attrs_post_init__(self) -> None:
        logger = logging.getLogger(self.logger_name)
        logger.setLevel(logging.INFO)
        logger.propagate = False
        logger.addHandler(RichHandler(show_time=True, show_path=False))
```

- `logger_name = field(default='griptape', kw_only=True)` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/configs/logging/logging_config.py

```
def __attrs_post_init__(self) -> None:
    logger = logging.getLogger(self.logger_name)
    logger.setLevel(logging.INFO)
    logger.propagate = False
    logger.addHandler(RichHandler(show_time=True, show_path=False))
```

newline_logging_filter

Bases:

Filter

Source Code in griptape/configs/logging/newline_logging_filter.py

```
@define
class NewlineLoggingFilter(logging.Filter):
    replace_str: str = field(default=" ", kw_only=True)

    def filter(self, record: Any) -> bool:
        record.msg = record.msg.replace("\n", self.replace_str)
        return True
```

- `replace_str = field(default=' ', kw_only=True)` class-attribute instance-attribute

filter(record)

Source Code in griptape/configs/logging/newline_logging_filter.py

```
def filter(self, record: Any) -> bool:
    record.msg = record.msg.replace("\n", self.replace_str)
    return True
```

truncate_logging_filter

Bases:

Filter

Source Code in griptape/configs/logging/truncate_logging_filter.py

```
@define
class TruncateLoggingFilter(logging.Filter):
    max_log_length: int = field(default=1000, kw_only=True)

    def filter(self, record: Any) -> bool:
        message = record.getMessage()

        if len(message) > self.max_log_length:
            record.msg = f"{message[: self.max_log_length]}... [{len(message) - self.
↪max_log_length} more characters]"
            record.args = ()
        return True
```

- `max_log_length = field(default=1000, kw_only=True)` class-attribute instance-attribute

filter(record)

Source Code in griptape/configs/logging/truncate_logging_filter.py

```
def filter(self, record: Any) -> bool:
    message = record.getMessage()

    if len(message) > self.max_log_length:
        record.msg = f"{message[: self.max_log_length]}... [{len(message) - self.max_
↪log_length} more characters]"
        record.args = ()
    return True
```

11.5.6 drivers

- `__all__ = ['AmazonBedrockCohereEmbeddingDriver', 'AmazonBedrockImageGenerationDriver', 'AmazonBedrockPromptDriver', 'AmazonBedrockTitanEmbeddingDriver', 'AmazonDynamoDbConversationMemoryDriver', 'AmazonOpenSearchVectorStoreDriver', 'AmazonRedshiftSqlDriver', 'AmazonS3FileManagerDriver', 'AmazonSageMakerJumpstartEmbeddingDriver' 'AmazonSageMakerJumpstartPromptDriver', 'AmazonSqsEventListenerDriver', 'AnthropicPromptDriver', 'AstraDbVectorStoreDriver', 'AwsIotCoreEventListenerDriver', 'AzureMongoDbVectorStoreDriver', 'AzureOpenAiChatPromptDriver', 'AzureOpenAiEmbeddingDriver', 'AzureOpenAiImageGenerationDriver',`

```
'AzureOpenAiTextToSpeechDriver', 'BaseAssistantDriver', 'BaseAudioTranscriptionDriver',
'BaseConversationMemoryDriver', 'BaseDiffusionImageGenerationPipelineDriver',
'BaseEmbeddingDriver', 'BaseEventListenerDriver', 'BaseFileManagerDriver',
'BaseImageGenerationDriver', 'BaseImageGenerationModelDriver',
'BaseMultiModelImageGenerationDriver', 'BaseObservabilityDriver',
'BasePromptDriver', 'BaseRerankDriver', 'BaseRulesetDriver',
'BaseSqlDriver', 'BaseStructureRunDriver', 'BaseTextToSpeechDriver',
'BaseVectorStoreDriver', 'BaseWebScraperDriver', 'BaseWebSearchDriver',
'BedrockStableDiffusionImageGenerationModelDriver', 'BedrockTitanImageGenerationModelDriver',
'CohereEmbeddingDriver', 'CoherePromptDriver', 'CohereRerankDriver',
'DatadogObservabilityDriver', 'DuckDuckGoWebSearchDriver', 'DummyAudioTranscriptionDriver',
'DummyEmbeddingDriver', 'DummyImageGenerationDriver', 'DummyPromptDriver',
'DummyTextToSpeechDriver', 'DummyVectorStoreDriver', 'ElevenLabsTextToSpeechDriver',
'ExaWebSearchDriver', 'GoogleEmbeddingDriver', 'GooglePromptDriver',
'GoogleWebSearchDriver', 'GriptapeCloudAssistantDriver', 'GriptapeCloudConversationMemoryDriver'
'GriptapeCloudEventListenerDriver', 'GriptapeCloudFileManagerDriver',
'GriptapeCloudObservabilityDriver', 'GriptapeCloudPromptDriver',
'GriptapeCloudRulesetDriver', 'GriptapeCloudStructureRunDriver',
'GriptapeCloudVectorStoreDriver', 'GrokPromptDriver', 'HuggingFaceHubEmbeddingDriver',
'HuggingFaceHubPromptDriver', 'HuggingFacePipelineImageGenerationDriver',
'HuggingFacePipelinePromptDriver', 'LeonardoImageGenerationDriver',
'LocalConversationMemoryDriver', 'LocalFileManagerDriver', 'LocalRerankDriver',
'LocalRulesetDriver', 'LocalStructureRunDriver', 'LocalVectorStoreDriver',
'MarkdownifyWebScraperDriver', 'MarqoVectorStoreDriver', 'MongoDbAtlasVectorStoreDriver',
'NoOpObservabilityDriver', 'OllamaEmbeddingDriver', 'OllamaPromptDriver',
'OpenAiAssistantDriver', 'OpenAiAudioTranscriptionDriver', 'OpenAiChatPromptDriver',
'OpenAiEmbeddingDriver', 'OpenAiImageGenerationDriver', 'OpenAiTextToSpeechDriver',
'OpenSearchVectorStoreDriver', 'OpenTelemetryObservabilityDriver',
'PerplexityPromptDriver', 'PerplexityWebSearchDriver', 'PgAiKnowledgeBaseVectorStoreDriver',
'PgVectorVectorStoreDriver', 'PineconeVectorStoreDriver', 'ProxyWebScraperDriver',
'PusherEventListenerDriver', 'QdrantVectorStoreDriver', 'RedisConversationMemoryDriver',
'RedisVectorStoreDriver', 'SnowflakeSqlDriver', 'SqlDriver',
'StableDiffusion3ControlNetImageGenerationPipelineDriver', 'StableDiffusion3ImageGenerationPipel
'StableDiffusion3Img2ImgImageGenerationPipelineDriver', 'StableDiffusion3ControlNetImageGenerati
'BaseImageGenerationDriver', 'BaseMultiModelImageGenerationDriver',
'OpenAiImageGenerationDriver', 'LeonardoImageGenerationDriver',
'AmazonBedrockImageGenerationDriver', 'AzureOpenAiImageGenerationDriver',
'DummyImageGenerationDriver', 'HuggingFacePipelineImageGenerationDriver',
'GriptapeCloudImageGenerationDriver', 'BaseWebScraperDriver',
'TrafilaturaWebScraperDriver', 'MarkdownifyWebScraperDriver',
'ProxyWebScraperDriver', 'BaseWebSearchDriver', 'GoogleWebSearchDriver',
'DuckDuckGoWebSearchDriver', 'ExaWebSearchDriver', 'TavilyWebSearchDriver',
'TrafilaturaWebScraperDriver', 'VoyageAiEmbeddingDriver', 'WebhookEventListenerDriver']
```

module-attribute

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<code class="language-python">	*str*	Embedding model name. Defaults to DEFAULT_MODEL.
<code class="language-python">	*str*	Defaults to *search_query*. Prepends special tokens to differentiate each type from one another: *search_document* when you encode documents for embeddings that you store in a vector database. *search_query* when querying your vector DB to find relevant documents.
<code class="language-python">	*Session*	Optionally provide custom *boto3.Session*.

continues on next page

Table 37 – continued from previous page

Name	Type	Description
<code class="language-python">	<code class="language-python">BaseTokenizer</code>	Optionally provide custom *BedrockCohereTokenizer*.
<a href="./#griptape.drivers.AmazonBedrockCohereEmbeddingDriver.client" title="client() (griptape.drivers.AmazonBedrockCohereEmbeddingDriver.client)"><code class="language-python">	*BedrockRuntimeClient*	Optionally provide custom *bedrock-runtime* client.

Source Code in griptape/drivers/embedding/amazon_bedrock_cohere_embedding_driver.py

```
@define
class AmazonBedrockCohereEmbeddingDriver(BaseEmbeddingDriver):
    """Amazon Bedrock Cohere Embedding Driver.

    Attributes:
        model: Embedding model name. Defaults to DEFAULT_MODEL.
        input_type: Defaults to `search_query`. Prepends special tokens to␣
↪differentiate each type from one another:
            `search_document` when you encode documents for embeddings that you store␣
↪in a vector database.
            `search_query` when querying your vector DB to find relevant documents.
        session: Optionally provide custom `boto3.Session`.
        tokenizer: Optionally provide custom `BedrockCohereTokenizer`.
        client: Optionally provide custom `bedrock-runtime` client.
    """

    DEFAULT_MODEL = "cohere.embed-english-v3"

    model: str = field(default=DEFAULT_MODEL, kw_only=True)
```

(continues on next page)

(continued from previous page)

```
    input_type: str = field(default="search_query", kw_only=True)
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: AmazonBedrockTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
    _client: Optional[BedrockRuntimeClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> BedrockRuntimeClient:
        return self.session.client("bedrock-runtime")

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        payload = {"input_type": self.input_type, "texts": [chunk]}

        response = self.client.invoke_model(
            body=json.dumps(payload),
            modelId=self.model,
            accept="*/*",
            contentType="application/json",
        )
        response_body = json.loads(response.get("body").read())

        return response_body.get("embeddings")[0]
```

- `DEFAULT_MODEL = 'cohere.embed-english-v3'` class-attribute instance-attribute
- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `input_type = field(default='search_query', kw_only=True)` class-attribute instance-attribute
- `model = field(default=DEFAULT_MODEL, kw_only=True)` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: AmazonBedrockTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/amazon_bedrock_cohere_embedding_driver.py

```
@lazy_property()
def client(self) -> BedrockRuntimeClient:
    return self.session.client("bedrock-runtime")
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/amazon_bedrock_cohere_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    payload = {"input_type": self.input_type, "texts": [chunk]}

    response = self.client.invoke_model(
        body=json.dumps(payload),
        modelId=self.model,
        accept="*/*",
        contentType="application/json",
    )
    response_body = json.loads(response.get("body").read())

    return response_body.get("embeddings")[0]
```

AmazonBedrockImageGenerationDriver

Bases: *BaseMultiModelImageGenerationDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.BaseImageGenerationDriver.model" title="model = field(kw_only=True, metadata={'serializable': True})">class-attribute instance-attribute (griptape.drivers.AmazonBedrockImageGenerationDriver.model)"><cod class="language-python">	*str*	Bedrock model ID.

continues on next page

Table 38 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.AmazonBedrockImageGenerationDriver.session" title="session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True) class-attribute instance-attribute (griptape.drivers.AmazonBedrockImageGenerationDriver.session)"><code class="language-python">	*Session*	boto3 session.
<a href="./#griptape.drivers.AmazonBedrockImageGenerationDriver.client" title="client() (griptape.drivers.AmazonBedrockImageGenerationDriver.client)"><code class="language-python">	*BedrockRuntimeClient*	Bedrock runtime client.
<a href="./#griptape.drivers.AmazonBedrockImageGenerationDriver.image_width" title="image_width = field(default=512, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.AmazonBedrockImageGenerationDriver.image_width)"><code class="language-python">	*int*	Width of output images. Defaults to 512 and must be a multiple of 64.

continues on next page

Table 38 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.AmazonBedrockImageGenerationDriver.image_height" title="image_height = field(default=512, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.AmazonBedrockImageGenerationDriver.image_height)"><code class="language-python">	*int*	Height of output images. Defaults to 512 and must be a multiple of 64.
<a href="./#griptape.drivers.AmazonBedrockImageGenerationDriver.seed" title="seed = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.AmazonBedrockImageGenerationDriver.seed)"><code class="language-python">	*Optional[int]*	Optionally provide a consistent seed to generation requests, increasing consistency in output.

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
@define
class AmazonBedrockImageGenerationDriver(BaseMultiModelImageGenerationDriver):
    """Driver for image generation models provided by Amazon Bedrock.

    Attributes:
        model: Bedrock model ID.
        session: boto3 session.
        client: Bedrock runtime client.
        image_width: Width of output images. Defaults to 512 and must be a multiple␣
↪of 64.
        image_height: Height of output images. Defaults to 512 and must be a multiple␣
↪of 64.
        seed: Optionally provide a consistent seed to generation requests, increasing␣
↪consistency in output.
    """
```

(continues on next page)

(continued from previous page)

```
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    image_width: int = field(default=512, kw_only=True, metadata={"serializable":␣
↪True})
    image_height: int = field(default=512, kw_only=True, metadata={"serializable":␣
↪True})
    seed: Optional[int] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    _client: Optional[BedrockRuntimeClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> BedrockRuntimeClient:
        return self.session.client("bedrock-runtime")

    def try_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact:
        request = self.image_generation_model_driver.text_to_image_request_parameters(
            prompts,
            self.image_width,
            self.image_height,
            negative_prompts=negative_prompts,
            seed=self.seed,
        )

        image_bytes = self._make_request(request)

        return ImageArtifact(
            value=image_bytes,
            format="png",
            width=self.image_width,
            height=self.image_height,
            meta={"prompt": ", ".join(prompts), "model": self.model},
        )

    def try_image_variation(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        request = self.image_generation_model_driver.image_variation_request_
↪parameters(
            prompts,
            image=image,
            negative_prompts=negative_prompts,
            seed=self.seed,
        )

        image_bytes = self._make_request(request)
```

(continues on next page)

(continued from previous page)

```
        return ImageArtifact(
            value=image_bytes,
            format="png",
            width=image.width,
            height=image.height,
            meta={"prompt": ", ".join(prompts), "model": self.model},
        )

    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        request = self.image_generation_model_driver.image_inpainting_request_
↪parameters(
            prompts,
            image=image,
            mask=mask,
            negative_prompts=negative_prompts,
            seed=self.seed,
        )

        image_bytes = self._make_request(request)

        return ImageArtifact(
            value=image_bytes,
            format="png",
            width=image.width,
            height=image.height,
            meta={"prompt": ", ".join(prompts), "model": self.model},
        )

    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        request = self.image_generation_model_driver.image_outpainting_request_
↪parameters(
            prompts,
            image=image,
            mask=mask,
            negative_prompts=negative_prompts,
            seed=self.seed,
        )

        image_bytes = self._make_request(request)
```

(continues on next page)

(continued from previous page)

```
        return ImageArtifact(
            value=image_bytes,
            format="png",
            width=image.width,
            height=image.height,
            meta={"prompt": ", ".join(prompts), "model": self.model},
        )

    def _make_request(self, request: dict) -> bytes:
        response = self.client.invoke_model(
            body=json.dumps(request),
            modelId=self.model,
            accept="application/json",
            contentType="application/json",
        )

        response_body = json.loads(response.get("body").read())

        try:
            image_bytes = self.image_generation_model_driver.get_generated_
↪image(response_body)
        except Exception as e:
            raise ValueError(f"Inpainting generation failed: {e}") from e

        return image_bytes
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `image_height = field(default=512, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `image_width = field(default=512, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `seed = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute

_make_request(request)

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
def _make_request(self, request: dict) -> bytes:
    response = self.client.invoke_model(
        body=json.dumps(request),
        modelId=self.model,
        accept="application/json",
        contentType="application/json",
    )

    response_body = json.loads(response.get("body").read())
```

(continues on next page)

(continued from previous page)

```
    try:
        image_bytes = self.image_generation_model_driver.get_generated_image(response_
↪body)
    except Exception as e:
        raise ValueError(f"Inpainting generation failed: {e}") from e

    return image_bytes
```

client()

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
@lazy_property()
def client(self) -> BedrockRuntimeClient:
    return self.session.client("bedrock-runtime")
```

try_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    request = self.image_generation_model_driver.image_inpainting_request_parameters(
        prompts,
        image=image,
        mask=mask,
        negative_prompts=negative_prompts,
        seed=self.seed,
    )

    image_bytes = self._make_request(request)

    return ImageArtifact(
        value=image_bytes,
        format="png",
        width=image.width,
        height=image.height,
        meta={"prompt": ", ".join(prompts), "model": self.model},
    )
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
def try_image_outpainting(
    self,
```

(continues on next page)

(continued from previous page)

```
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    request = self.image_generation_model_driver.image_outpainting_request_parameters(
        prompts,
        image=image,
        mask=mask,
        negative_prompts=negative_prompts,
        seed=self.seed,
    )

    image_bytes = self._make_request(request)

    return ImageArtifact(
        value=image_bytes,
        format="png",
        width=image.width,
        height=image.height,
        meta={"prompt": ", ".join(prompts), "model": self.model},
    )
```

try_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
def try_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    request = self.image_generation_model_driver.image_variation_request_parameters(
        prompts,
        image=image,
        negative_prompts=negative_prompts,
        seed=self.seed,
    )

    image_bytes = self._make_request(request)

    return ImageArtifact(
        value=image_bytes,
        format="png",
        width=image.width,
        height=image.height,
        meta={"prompt": ", ".join(prompts), "model": self.model},
    )
```

try_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    request = self.image_generation_model_driver.text_to_image_request_parameters(
        prompts,
        self.image_width,
        self.image_height,
        negative_prompts=negative_prompts,
        seed=self.seed,
    )

    image_bytes = self._make_request(request)

    return ImageArtifact(
        value=image_bytes,
        format="png",
        width=self.image_width,
        height=self.image_height,
        meta={"prompt": ", ".join(prompts), "model": self.model},
    )
```

AmazonBedrockPromptDriver

Bases: *BasePromptDriver* “)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
@define
class AmazonBedrockPromptDriver(BasePromptDriver):
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    additional_model_request_fields: dict = field(default=Factory(dict), kw_only=True)
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: AmazonBedrockTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
    use_native_tools: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})
    structured_output_strategy: StructuredOutputStrategy = field(
        default="tool", kw_only=True, metadata={"serializable": True}
    )
    tool_choice: dict = field(default=Factory(lambda: {"auto": {}}), kw_only=True,␣
↪metadata={"serializable": True})
    _client: Optional[Any] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @structured_output_strategy.validator  # pyright:␣
↪ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
        if value == "native":
```

(continues on next page)

(continued from previous page)

```
            raise ValueError(f"{__class__.__name__} does not support `{value}` structured output strategy.")

        return value

    @lazy_property()
    def client(self) -> Any:
        return self.session.client("bedrock-runtime")

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        params = self._base_params(prompt_stack)
        logger.debug(params)
        response = self.client.converse(**params)
        logger.debug(response)

        usage = response["usage"]
        output_message = response["output"]["message"]

        message_content = output_message.get("content", [])

        # Move reasoning content to the beginning of the content to have it appear first in output
        reasoning_content_index = next(
            (i for i, content in enumerate(message_content) if "reasoningContent" in content),
            None,
        )
        if reasoning_content_index is not None:
            message_content.insert(0, message_content.pop(reasoning_content_index))

        return Message(
            content=[self.__to_prompt_stack_message_content(content) for content in message_content],
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(input_tokens=usage["inputTokens"], output_tokens=usage["outputTokens"]),
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        params = self._base_params(prompt_stack)
        logger.debug(params)
        response = self.client.converse_stream(**params)

        stream = response.get("stream")
        if stream is not None:
            for event in stream:
                logger.debug(event)
                if "contentBlockDelta" in event or "contentBlockStart" in event:
                    yield DeltaMessage(content=self.__to_prompt_stack_delta_message_content(event))
```

(continues on next page)

(continued from previous page)

```
                elif "metadata" in event:
                    usage = event["metadata"]["usage"]
                    yield DeltaMessage(
                        usage=DeltaMessage.Usage(
                            input_tokens=usage["inputTokens"],
                            output_tokens=usage["outputTokens"],
                        ),
                    )
        else:
            raise Exception("model response is empty")

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        system_messages = [{"text": message.to_text()} for message in prompt_stack.
↪system_messages]
        messages = self.__to_bedrock_messages([message for message in prompt_stack.
↪messages if not message.is_system()])

        params = {
            "modelId": self.model,
            "messages": messages,
            "system": system_messages,
            "inferenceConfig": {
                "temperature": self.temperature,
                **({"maxTokens": self.max_tokens} if self.max_tokens is not None else
↪{}),
            },
            "additionalModelRequestFields": self.additional_model_request_fields,
            **self.extra_params,
        }

        if prompt_stack.tools and self.use_native_tools:
            params["toolConfig"] = {
                "tools": [],
                "toolChoice": self.tool_choice,
            }

            if prompt_stack.output_schema is not None and self.structured_output_
↪strategy == "tool":
                params["toolConfig"]["toolChoice"] = {"any": {}}

            params["toolConfig"]["tools"] = self.__to_bedrock_tools(prompt_stack.
↪tools)

        return params

    def __to_bedrock_messages(self, messages: list[Message]) -> list[dict]:
        return [
            {
                "role": self.__to_bedrock_role(message),
                "content": [self.__to_bedrock_message_content(content) for content in␣
↪message.content],
            }
```

(continues on next page)

(continued from previous page)

```
            for message in messages
        ]

    def __to_bedrock_role(self, message: Message) -> str:
        if message.is_assistant():
            return "assistant"
        return "user"

    def __to_bedrock_tools(self, tools: list[BaseTool]) -> list[dict]:
        return [
            {
                "toolSpec": {
                    "name": tool.to_native_tool_name(activity),
                    "description": tool.activity_description(activity),
                    "inputSchema": {
                        "json": tool.to_activity_json_schema(activity, "http://json-
↪schema.org/draft-07/schema#"),
                    },
                },
            }
            for tool in tools
            for activity in tool.activities()
        ]

    def __to_bedrock_message_content(self, content: BaseMessageContent) -> dict:
        if isinstance(content, TextMessageContent):
            return {"text": content.artifact.to_text()}
        if isinstance(content, ImageMessageContent):
            artifact = content.artifact
            if isinstance(artifact, ImageArtifact):
                return {"image": {"format": artifact.format, "source": {"bytes":␣
↪artifact.value}}}
            if isinstance(artifact, ImageUrlArtifact):
                return {"image": {"format": "png", "source": {"s3Location": {"uri":␣
↪artifact.value}}}}
            raise ValueError(f"Unsupported image artifact type: {type(artifact)}")
        if isinstance(content, ActionCallMessageContent):
            action_call = content.artifact.value

            return {
                "toolUse": {
                    "toolUseId": action_call.tag,
                    "name": f"{action_call.name}_{action_call.path}",
                    "input": action_call.input,
                },
            }
        if isinstance(content, ActionResultMessageContent):
            artifact = content.artifact

            if isinstance(artifact, ListArtifact):
                message_content = [self.__to_bedrock_tool_use_content(artifact) for␣
↪artifact in artifact.value]
```

(continues on next page)

(continued from previous page)

```
            else:
                message_content = [self.__to_bedrock_tool_use_content(artifact)]

            return {
                "toolResult": {
                    "toolUseId": content.action.tag,
                    "content": message_content,
                    "status": "error" if isinstance(artifact, ErrorArtifact) else
↪"success",
                },
            }
        return content.artifact.value

    def __to_bedrock_tool_use_content(self, artifact: BaseArtifact) -> dict:
        if isinstance(artifact, ImageArtifact):
            return {"image": {"format": artifact.format, "source": {"bytes": artifact.
↪value}}}
        return {"text": artifact.to_text()}

    def __to_prompt_stack_message_content(self, content: dict) -> BaseMessageContent:
        if "text" in content:
            return TextMessageContent(TextArtifact(content["text"]))
        if "toolUse" in content:
            name, path = ToolAction.from_native_tool_name(content["toolUse"]["name"])
            return ActionCallMessageContent(
                artifact=ActionArtifact(
                    value=ToolAction(
                        tag=content["toolUse"]["toolUseId"],
                        name=name,
                        path=path,
                        input=content["toolUse"]["input"],
                    ),
                ),
            )
        if "reasoningContent" in content:
            return TextMessageContent(TextArtifact(content["reasoningContent"][
↪"reasoningText"]["text"]))
        raise ValueError(f"Unsupported message content type: {content}")

    def __to_prompt_stack_delta_message_content(self, event: dict) ->␣
↪BaseDeltaMessageContent:
        if "contentBlockStart" in event:
            content_block = event["contentBlockStart"]["start"]

            if "toolUse" in content_block:
                name, path = ToolAction.from_native_tool_name(content_block["toolUse
↪"]["name"])

                return ActionCallDeltaMessageContent(
                    index=event["contentBlockStart"]["contentBlockIndex"],
                    tag=content_block["toolUse"]["toolUseId"],
                    name=name,
```

(continues on next page)

(continued from previous page)

```
                    path=path,
                )
            if "text" in content_block:
                return TextDeltaMessageContent(
                    content_block["text"],
                    index=event["contentBlockStart"]["contentBlockIndex"],
                )
            raise ValueError(f"Unsupported message content type: {event}")
        if "contentBlockDelta" in event:
            content_block_delta = event["contentBlockDelta"]

            if "text" in content_block_delta["delta"]:
                return TextDeltaMessageContent(
                    content_block_delta["delta"]["text"],
                    index=content_block_delta["contentBlockIndex"],
                )
            if "toolUse" in content_block_delta["delta"]:
                return ActionCallDeltaMessageContent(
                    index=content_block_delta["contentBlockIndex"],
                    partial_input=content_block_delta["delta"]["toolUse"]["input"],
                )
            raise ValueError(f"Unsupported message content type: {event}")
        raise ValueError(f"Unsupported message content type: {event}")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `additional_model_request_fields = field(default=Factory(dict), kw_only=True)` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `structured_output_strategy = field(default='tool', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: AmazonBedrockTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `tool_choice = field(default=Factory(lambda: {'auto': {}}), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `use_native_tools = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__to_bedrock_message_content(content)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_bedrock_message_content(self, content: BaseMessageContent) -> dict:
    if isinstance(content, TextMessageContent):
        return {"text": content.artifact.to_text()}
    if isinstance(content, ImageMessageContent):
        artifact = content.artifact
        if isinstance(artifact, ImageArtifact):
```

(continues on next page)

(continued from previous page)

```
            return {"image": {"format": artifact.format, "source": {"bytes": artifact.
↪value}}}
        if isinstance(artifact, ImageUrlArtifact):
            return {"image": {"format": "png", "source": {"s3Location": {"uri":␣
↪artifact.value}}}}
        raise ValueError(f"Unsupported image artifact type: {type(artifact)}")
    if isinstance(content, ActionCallMessageContent):
        action_call = content.artifact.value

        return {
            "toolUse": {
                "toolUseId": action_call.tag,
                "name": f"{action_call.name}_{action_call.path}",
                "input": action_call.input,
            },
        }
    if isinstance(content, ActionResultMessageContent):
        artifact = content.artifact

        if isinstance(artifact, ListArtifact):
            message_content = [self.__to_bedrock_tool_use_content(artifact) for␣
↪artifact in artifact.value]
        else:
            message_content = [self.__to_bedrock_tool_use_content(artifact)]

        return {
            "toolResult": {
                "toolUseId": content.action.tag,
                "content": message_content,
                "status": "error" if isinstance(artifact, ErrorArtifact) else "success
↪",
            },
        }
    return content.artifact.value
```

__to_bedrock_messages(messages)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_bedrock_messages(self, messages: list[Message]) -> list[dict]:
    return [
        {
            "role": self.__to_bedrock_role(message),
            "content": [self.__to_bedrock_message_content(content) for content in␣
↪message.content],
        }
        for message in messages
    ]
```

__to_bedrock_role(message)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_bedrock_role(self, message: Message) -> str:
    if message.is_assistant():
        return "assistant"
    return "user"
```

__to_bedrock_tool_use_content(artifact)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_bedrock_tool_use_content(self, artifact: BaseArtifact) -> dict:
    if isinstance(artifact, ImageArtifact):
        return {"image": {"format": artifact.format, "source": {"bytes": artifact.
↪value}}}
    return {"text": artifact.to_text()}
```

__to_bedrock_tools(tools)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_bedrock_tools(self, tools: list[BaseTool]) -> list[dict]:
    return [
        {
            "toolSpec": {
                "name": tool.to_native_tool_name(activity),
                "description": tool.activity_description(activity),
                "inputSchema": {
                    "json": tool.to_activity_json_schema(activity, "http://json-
↪schema.org/draft-07/schema#"),
                },
            },
        }
        for tool in tools
        for activity in tool.activities()
    ]
```

__to_prompt_stack_delta_message_content(event)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_prompt_stack_delta_message_content(self, event: dict) ->␣
↪BaseDeltaMessageContent:
    if "contentBlockStart" in event:
        content_block = event["contentBlockStart"]["start"]

        if "toolUse" in content_block:
            name, path = ToolAction.from_native_tool_name(content_block["toolUse"][
↪"name"])

            return ActionCallDeltaMessageContent(
                index=event["contentBlockStart"]["contentBlockIndex"],
```

(continues on next page)

(continued from previous page)

```
                tag=content_block["toolUse"]["toolUseId"],
                name=name,
                path=path,
            )
        if "text" in content_block:
            return TextDeltaMessageContent(
                content_block["text"],
                index=event["contentBlockStart"]["contentBlockIndex"],
            )
        raise ValueError(f"Unsupported message content type: {event}")
    if "contentBlockDelta" in event:
        content_block_delta = event["contentBlockDelta"]

        if "text" in content_block_delta["delta"]:
            return TextDeltaMessageContent(
                content_block_delta["delta"]["text"],
                index=content_block_delta["contentBlockIndex"],
            )
        if "toolUse" in content_block_delta["delta"]:
            return ActionCallDeltaMessageContent(
                index=content_block_delta["contentBlockIndex"],
                partial_input=content_block_delta["delta"]["toolUse"]["input"],
            )
        raise ValueError(f"Unsupported message content type: {event}")
    raise ValueError(f"Unsupported message content type: {event}")
```

__to_prompt_stack_message_content(content)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_prompt_stack_message_content(self, content: dict) -> BaseMessageContent:
    if "text" in content:
        return TextMessageContent(TextArtifact(content["text"]))
    if "toolUse" in content:
        name, path = ToolAction.from_native_tool_name(content["toolUse"]["name"])
        return ActionCallMessageContent(
            artifact=ActionArtifact(
                value=ToolAction(
                    tag=content["toolUse"]["toolUseId"],
                    name=name,
                    path=path,
                    input=content["toolUse"]["input"],
                ),
            ),
        )
    if "reasoningContent" in content:
        return TextMessageContent(TextArtifact(content["reasoningContent"][
↪"reasoningText"]["text"]))
    raise ValueError(f"Unsupported message content type: {content}")
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    system_messages = [{"text": message.to_text()} for message in prompt_stack.system_
↪messages]
    messages = self.__to_bedrock_messages([message for message in prompt_stack.
↪messages if not message.is_system()])

    params = {
        "modelId": self.model,
        "messages": messages,
        "system": system_messages,
        "inferenceConfig": {
            "temperature": self.temperature,
            **({"maxTokens": self.max_tokens} if self.max_tokens is not None else {}),
        },
        "additionalModelRequestFields": self.additional_model_request_fields,
        **self.extra_params,
    }

    if prompt_stack.tools and self.use_native_tools:
        params["toolConfig"] = {
            "tools": [],
            "toolChoice": self.tool_choice,
        }

        if prompt_stack.output_schema is not None and self.structured_output_strategy␣
↪== "tool":
            params["toolConfig"]["toolChoice"] = {"any": {}}

        params["toolConfig"]["tools"] = self.__to_bedrock_tools(prompt_stack.tools)

    return params
```

client()

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
@lazy_property()
def client(self) -> Any:
    return self.session.client("bedrock-runtime")
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    params = self._base_params(prompt_stack)
    logger.debug(params)
    response = self.client.converse(**params)
    logger.debug(response)
```

(continues on next page)

(continued from previous page)

```
    usage = response["usage"]
    output_message = response["output"]["message"]

    message_content = output_message.get("content", [])

    # Move reasoning content to the beginning of the content to have it appear first␣
↪in output
    reasoning_content_index = next(
        (i for i, content in enumerate(message_content) if "reasoningContent" in␣
↪content),
        None,
    )
    if reasoning_content_index is not None:
        message_content.insert(0, message_content.pop(reasoning_content_index))

    return Message(
        content=[self.__to_prompt_stack_message_content(content) for content in␣
↪message_content],
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(input_tokens=usage["inputTokens"], output_tokens=usage[
↪"outputTokens"]),
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    params = self._base_params(prompt_stack)
    logger.debug(params)
    response = self.client.converse_stream(**params)

    stream = response.get("stream")
    if stream is not None:
        for event in stream:
            logger.debug(event)
            if "contentBlockDelta" in event or "contentBlockStart" in event:
                yield DeltaMessage(content=self.__to_prompt_stack_delta_message_
↪content(event))
            elif "metadata" in event:
                usage = event["metadata"]["usage"]
                yield DeltaMessage(
                    usage=DeltaMessage.Usage(
                        input_tokens=usage["inputTokens"],
                        output_tokens=usage["outputTokens"],
                    ),
                )
    else:
        raise Exception("model response is empty")
```

validate*structured_output_strategy(*, value)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
@structured_output_strategy.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
    if value == "native":
        raise ValueError(f"{__class__.__name__} does not support `{value}` structured␣
↪output strategy.")

    return value
```

AmazonBedrockTitanEmbeddingDriver

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<code class="language-python">	*str*	Embedding model name. Defaults to DEFAULT_MODEL.

continues on next page

Table 39 – continued from previous page

Name	Type	Description
<code class="language-python">	<code class="language-python">BaseTokenizer</code>	Optionally provide custom *BedrockTitanTokenizer*.
<a href="./#griptape.drivers.AmazonBedrockTitanEmbeddingDriver.session" title="session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True) class-attribute instance-attribute (griptape.drivers.AmazonBedrockTitanEmbeddingDriver.session)"><code class="language-python">	*Session*	Optionally provide custom *boto3.Session*.
<a href="./#griptape.drivers.AmazonBedrockTitanEmbeddingDriver.client" title="client() (griptape.drivers.AmazonBedrockTitanEmbeddingDriver.client)"><code class="language-python">	*BedrockRuntimeClient*	Optionally provide custom *bedrock-runtime* client.

Source Code in griptape/drivers/embedding/amazon_bedrock_titan_embedding_driver.py

```
@define
class AmazonBedrockTitanEmbeddingDriver(BaseEmbeddingDriver):
```

(continues on next page)

(continued from previous page)

```
    """Amazon Bedrock Titan Embedding Driver.

    Attributes:
        model: Embedding model name. Defaults to DEFAULT_MODEL.
        tokenizer: Optionally provide custom `BedrockTitanTokenizer`.
        session: Optionally provide custom `boto3.Session`.
        client: Optionally provide custom `bedrock-runtime` client.
    """

    DEFAULT_MODEL = "amazon.titan-embed-text-v1"

    model: str = field(default=DEFAULT_MODEL, kw_only=True, metadata={"serializable":␣
↪True})
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: AmazonBedrockTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
    _client: Optional[BedrockRuntimeClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> BedrockRuntimeClient:
        return self.session.client("bedrock-runtime")

    def try_embed_artifact(self, artifact: TextArtifact | ImageArtifact, **kwargs) ->␣
↪list[float]:
        if isinstance(artifact, TextArtifact):
            return self.try_embed_chunk(artifact.value)
        return self._invoke_model({"inputImage": base64.b64encode(artifact.value).
↪decode()})["embedding"]

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        return self._invoke_model(
            {
                "inputText": chunk,
            }
        )["embedding"]

    def _invoke_model(self, payload: dict) -> dict[str, Any]:
        response = self.client.invoke_model(
            body=json.dumps(payload),
            modelId=self.model,
            accept="application/json",
            contentType="application/json",
        )
        return json.loads(response.get("body").read())
```

- `DEFAULT_MODEL = 'amazon.titan-embed-text-v1'` class-attribute instance-attribute

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: AmazonBedrockTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute

_invoke_model(payload)

Source Code in griptape/drivers/embedding/amazon_bedrock_titan_embedding_driver.py

```
def _invoke_model(self, payload: dict) -> dict[str, Any]:
    response = self.client.invoke_model(
        body=json.dumps(payload),
        modelId=self.model,
        accept="application/json",
        contentType="application/json",
    )
    return json.loads(response.get("body").read())
```

client()

Source Code in griptape/drivers/embedding/amazon_bedrock_titan_embedding_driver.py

```
@lazy_property()
def client(self) -> BedrockRuntimeClient:
    return self.session.client("bedrock-runtime")
```

try_embed_artifact(artifact, **kwargs)

Source Code in griptape/drivers/embedding/amazon_bedrock_titan_embedding_driver.py

```
def try_embed_artifact(self, artifact: TextArtifact | ImageArtifact, **kwargs) ->␣
↪list[float]:
    if isinstance(artifact, TextArtifact):
        return self.try_embed_chunk(artifact.value)
    return self._invoke_model({"inputImage": base64.b64encode(artifact.value).
↪decode()})["embedding"]
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/amazon_bedrock_titan_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    return self._invoke_model(
        {
            "inputText": chunk,
        }
    )["embedding"]
```

AmazonDynamoDbConversationMemoryDriver

Bases: *BaseConversationMemoryDriver* “)

Source Code in griptape/drivers/memory/conversation/amazon_dynamodb_conversation_memory_driver.py

```
@define
class AmazonDynamoDbConversationMemoryDriver(BaseConversationMemoryDriver):
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    table_name: str = field(kw_only=True, metadata={"serializable": True})
    partition_key: str = field(kw_only=True, metadata={"serializable": True})
    value_attribute_key: str = field(kw_only=True, metadata={"serializable": True})
    partition_key_value: str = field(kw_only=True, metadata={"serializable": True})
    sort_key: Optional[str] = field(default=None, metadata={"serializable": True})
    sort_key_value: Optional[str | int] = field(default=None, metadata={"serializable
↪": True})
    _table: Optional[Table] = field(default=None, kw_only=True, alias="table",␣
↪metadata={"serializable": False})

    @lazy_property()
    def table(self) -> Table:
        return self.session.resource("dynamodb").Table(self.table_name)

    def store(self, runs: list[Run], metadata: dict) -> None:
        self.table.update_item(
            Key=self._get_key(),
            UpdateExpression="set #attr = :value",
            ExpressionAttributeNames={"#attr": self.value_attribute_key},
            ExpressionAttributeValues={
                ":value": json.dumps(self._to_params_dict(runs, metadata)),
            },
        )

    def load(self) -> tuple[list[Run], dict[str, Any]]:
        response = self.table.get_item(Key=self._get_key())

        if "Item" in response and self.value_attribute_key in response["Item"]:
            memory_dict = json.loads(str(response["Item"][self.value_attribute_key]))
            return self._from_params_dict(memory_dict)
        return [], {}

    def _get_key(self) -> dict[str, str | int]:
        key: dict[str, str | int] = {self.partition_key: self.partition_key_value}

        if self.sort_key is not None and self.sort_key_value is not None:
            key[self.sort_key] = self.sort_key_value

        return key
```

- `_table = field(default=None, kw_only=True, alias='table', metadata={'serializable': False})` class-attribute instance-attribute
- `partition_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

- `partition_key_value = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `sort_key = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `sort_key_value = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `table_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `value_attribute_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_get_key()

Source Code in griptape/drivers/memory/conversation/amazon_dynamodb_conversation_memory_driver.py

```
def _get_key(self) -> dict[str, str | int]:
    key: dict[str, str | int] = {self.partition_key: self.partition_key_value}

    if self.sort_key is not None and self.sort_key_value is not None:
        key[self.sort_key] = self.sort_key_value

    return key
```

load()

Source Code in griptape/drivers/memory/conversation/amazon_dynamodb_conversation_memory_driver.py

```
def load(self) -> tuple[list[Run], dict[str, Any]]:
    response = self.table.get_item(Key=self._get_key())

    if "Item" in response and self.value_attribute_key in response["Item"]:
        memory_dict = json.loads(str(response["Item"][self.value_attribute_key]))
        return self._from_params_dict(memory_dict)
    return [], {}
```

store(runs, metadata)

Source Code in griptape/drivers/memory/conversation/amazon_dynamodb_conversation_memory_driver.py

```
def store(self, runs: list[Run], metadata: dict) -> None:
    self.table.update_item(
        Key=self._get_key(),
        UpdateExpression="set #attr = :value",
        ExpressionAttributeNames={"#attr": self.value_attribute_key},
        ExpressionAttributeValues={
            ":value": json.dumps(self._to_params_dict(runs, metadata)),
        },
    )
```

table()

Source Code in griptape/drivers/memory/conversation/amazon_dynamodb_conversation_memory_driver.py

```
@lazy_property()
def table(self) -> Table:
    return self.session.resource("dynamodb").Table(self.table_name)
```

AmazonOpenSearchVectorStoreDriver

Bases: *AmazonOpenSearchVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.AmazonOpenSearchVectorStoreDriver.session" title="session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True) class-attribute instance-attribute (griptape.drivers.AmazonOpenSearchVectorStoreDriver.session)"><code class="language-python">	*Session*	The boto3 session to use.
<a href="./#griptape.drivers.AmazonOpenSearchVectorStoreDriver.service" title="service = field(default='es', kw_only=True) class-attribute instance-attribute (griptape.drivers.AmazonOpenSearchVectorStoreDriver.service)"><code class="language-python">	*str*	Service name for AWS Signature v4. Values can be 'es' or 'aoss' for for OpenSearch Serverless. Defaults to 'es'.

continues on next page

Table 40 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.AmazonOpenSearchVectorStoreDriver.http_auth" title="http_auth = field(default=Factory(lambda self: import_optional_dependency('opensearchpy').AWSV4SignerAuth(self.session.get_credentials(), self.session.region_name, self.service), takes_self=True)) class-attribute instance-attribute (griptape.drivers.AmazonOpenSearchVectorStoreDriver.http_auth)"><class="language-python">	*str I tuple[str, str]*	The HTTP authentication credentials to use. Defaults to using credentials in the boto3 session.
<a href="./#griptape.drivers.OpenSearchVectorStoreDriver.client" title="client() (griptape.drivers.AmazonOpenSearchVectorStoreDriver.client)"><coc class="language-python">	*OpenSearch*	An optional OpenSearch client to use. Defaults to a new client using the host, port, http_auth, use_ssl, and verify_certs attributes.

Source Code in griptape/drivers/vector/amazon_opensearch_vector_store_driver.py

```
@define
class AmazonOpenSearchVectorStoreDriver(OpenSearchVectorStoreDriver):
    """A Vector Store Driver for Amazon OpenSearch.

    Attributes:
        session: The boto3 session to use.
        service: Service name for AWS Signature v4. Values can be es or aoss for for␣
↪OpenSearch Serverless. Defaults to es.
        http_auth: The HTTP authentication credentials to use. Defaults to using␣
↪credentials in the boto3 session.
        client: An optional OpenSearch client to use. Defaults to a new client using␣
↪the host, port, http_auth, use_ssl, and verify_certs attributes.
    """

    session: Session = field(default=Factory(lambda: import_optional_dependency("boto3
↪").Session()), kw_only=True)
    service: str = field(default="es", kw_only=True)
    http_auth: str | tuple[str, str] = field(
        default=Factory(
```

(continues on next page)

(continued from previous page)

```
            lambda self: import_optional_dependency("opensearchpy").AWSV4SignerAuth(
                self.session.get_credentials(),
                self.session.region_name,
                self.service,
            ),
            takes_self=True,
        ),
    )

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        """Inserts or updates a vector in OpenSearch.

        If a vector with the given vector ID already exists, it is updated; otherwise,
↪ a new vector is inserted.
        Metadata associated with the vector can also be provided.
        """
        vector_id = vector_id or str_to_hash(str(vector))
        doc = {"vector": vector, "namespace": namespace, "metadata": meta}
        doc.update(kwargs)
        if self.service == "aoss":
            response = self.client.index(index=self.index_name, body=doc)
        else:
            response = self.client.index(index=self.index_name, id=vector_id,␣
↪body=doc)

        return response["_id"]
```

- `http_auth = field(default=Factory(lambda self: import_optional_dependency('opensearchpy').AWSV4SignerAuth(self.session.get_credentials(), self.session.region_name, self.service), takes_self=True))` class-attribute instance-attribute
- `service = field(default='es', kw_only=True)` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/amazon_opensearch_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
```

(continues on next page)

(continued from previous page)

```
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    """Inserts or updates a vector in OpenSearch.

    If a vector with the given vector ID already exists, it is updated; otherwise, a new vector is inserted.
    Metadata associated with the vector can also be provided.
    """
    vector_id = vector_id or str_to_hash(str(vector))
    doc = {"vector": vector, "namespace": namespace, "metadata": meta}
    doc.update(kwargs)
    if self.service == "aoss":
        response = self.client.index(index=self.index_name, body=doc)
    else:
        response = self.client.index(index=self.index_name, id=vector_id, body=doc)

    return response["_id"]
```

AmazonRedshiftSqlDriver

Bases: *BaseSqlDriver* “)

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@define
class AmazonRedshiftSqlDriver(BaseSqlDriver):
    database: str = field(kw_only=True)
    session: boto3.Session = field(kw_only=True)
    cluster_identifier: Optional[str] = field(default=None, kw_only=True)
    workgroup_name: Optional[str] = field(default=None, kw_only=True)
    db_user: Optional[str] = field(default=None, kw_only=True)
    database_credentials_secret_arn: Optional[str] = field(default=None, kw_only=True)
    wait_for_query_completion_sec: float = field(default=0.3, kw_only=True)
    _client: Optional[RedshiftDataAPIServiceClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> RedshiftDataAPIServiceClient:
        return self.session.client("redshift-data")

    @workgroup_name.validator  # pyright: ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_params(self, _: Attribute, workgroup_name: Optional[str]) -> None:
        if not self.cluster_identifier and not self.workgroup_name:
            raise ValueError("Provide a value for one of `cluster_identifier` or `workgroup_name`")
        if self.cluster_identifier and self.workgroup_name:
            raise ValueError("Provide a value for either `cluster_identifier` or `workgroup_name`, but not both")
```

(continues on next page)

(continued from previous page)

```
    @classmethod
    def _process_rows_from_records(cls, records: list) -> list[list]:
        return [[c[list(c.keys())[0]] for c in r] for r in records]

    @classmethod
    def _process_cells_from_rows_and_columns(cls, columns: list, rows: list[list]) ->␣
↪list[dict[str, Any]]:
        return [{column: r[idx] for idx, column in enumerate(columns)} for r in rows]

    @classmethod
    def _process_columns_from_column_metadata(cls, meta: dict) -> list:
        return [k["name"] for k in meta]

    @classmethod
    def _post_process(cls, meta: dict, records: list) -> list[dict[str, Any]]:
        columns = cls._process_columns_from_column_metadata(meta)
        rows = cls._process_rows_from_records(records)
        return cls._process_cells_from_rows_and_columns(columns, rows)

    def execute_query(self, query: str) -> Optional[list[BaseSqlDriver.RowResult]]:
        rows = self.execute_query_raw(query)
        if rows:
            return [BaseSqlDriver.RowResult(row) for row in rows]
        return None

    def execute_query_raw(self, query: str) -> Optional[list[dict[str,␣
↪Optional[Any]]]]:
        function_kwargs = {"Sql": query, "Database": self.database}
        if self.workgroup_name:
            function_kwargs["WorkgroupName"] = self.workgroup_name
        if self.cluster_identifier:
            function_kwargs["ClusterIdentifier"] = self.cluster_identifier
        if self.db_user:
            function_kwargs["DbUser"] = self.db_user
        if self.database_credentials_secret_arn:
            function_kwargs["SecretArn"] = self.database_credentials_secret_arn

        response = self.client.execute_statement(**function_kwargs)  # pyright:␣
↪ignore[reportArgumentType]
        response_id = response["Id"]

        statement = self.client.describe_statement(Id=response_id)

        while statement["Status"] in ["SUBMITTED", "PICKED", "STARTED"]:
            time.sleep(self.wait_for_query_completion_sec)
            statement = self.client.describe_statement(Id=response_id)

        if statement["Status"] == "FINISHED":
            statement_result = self.client.get_statement_result(Id=response_id)
            results = statement_result.get("Records", [])
```

(continues on next page)

(continued from previous page)

```
            while "NextToken" in statement_result:
                statement_result = self.client.get_statement_result(
                    Id=response_id,
                    NextToken=statement_result["NextToken"],
                )
                results = results + response.get("Records", [])

            return self._post_process(statement_result["ColumnMetadata"], results)  #␣
↪pyright: ignore[reportArgumentType]

        if statement["Status"] in ["FAILED", "ABORTED"]:
            return None
        return None

    def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]:
        function_kwargs = {"Database": self.database, "Table": table_name}
        if schema:
            function_kwargs["Schema"] = schema
        if self.workgroup_name:
            function_kwargs["WorkgroupName"] = self.workgroup_name
        if self.cluster_identifier:
            function_kwargs["ClusterIdentifier"] = self.cluster_identifier
        if self.db_user:
            function_kwargs["DbUser"] = self.db_user
        if self.database_credentials_secret_arn:
            function_kwargs["SecretArn"] = self.database_credentials_secret_arn
        response = self.client.describe_table(**function_kwargs)  # pyright:␣
↪ignore[reportArgumentType]
        return str([col["name"] for col in response["ColumnList"] if "name" in col])
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `cluster_identifier = field(default=None, kw_only=True)` class-attribute instance-attribute
- `database = field(kw_only=True)` class-attribute instance-attribute
- `database_credentials_secret_arn = field(default=None, kw_only=True)` class-attribute instance-attribute
- `db_user = field(default=None, kw_only=True)` class-attribute instance-attribute
- `session = field(kw_only=True)` class-attribute instance-attribute
- `wait_for_query_completion_sec = field(default=0.3, kw_only=True)` class-attribute instance-attribute
- `workgroup_name = field(default=None, kw_only=True)` class-attribute instance-attribute

_post_process(meta, records)classmethod

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@classmethod
def _post_process(cls, meta: dict, records: list) -> list[dict[str, Any]]:
```

(continues on next page)

(continued from previous page)

```
    columns = cls._process_columns_from_column_metadata(meta)
    rows = cls._process_rows_from_records(records)
    return cls._process_cells_from_rows_and_columns(columns, rows)
```

_process_cells_from_rows_and_columns(columns, rows)classmethod

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@classmethod
def _process_cells_from_rows_and_columns(cls, columns: list, rows: list[list]) ->␣
↪list[dict[str, Any]]:
    return [{column: r[idx] for idx, column in enumerate(columns)} for r in rows]
```

_process_columns_from_column_metadata(meta)classmethod

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@classmethod
def _process_columns_from_column_metadata(cls, meta: dict) -> list:
    return [k["name"] for k in meta]
```

_process_rows_from_records(records)classmethod

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@classmethod
def _process_rows_from_records(cls, records: list) -> list[list]:
    return [[c[list(c.keys())[0]] for c in r] for r in records]
```

client()

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@lazy_property()
def client(self) -> RedshiftDataAPIServiceClient:
    return self.session.client("redshift-data")
```

execute_query(query)

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
def execute_query(self, query: str) -> Optional[list[BaseSqlDriver.RowResult]]:
    rows = self.execute_query_raw(query)
    if rows:
        return [BaseSqlDriver.RowResult(row) for row in rows]
    return None
```

execute_query_raw(query)

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
def execute_query_raw(self, query: str) -> Optional[list[dict[str, Optional[Any]]]]:
    function_kwargs = {"Sql": query, "Database": self.database}
    if self.workgroup_name:
        function_kwargs["WorkgroupName"] = self.workgroup_name
    if self.cluster_identifier:
        function_kwargs["ClusterIdentifier"] = self.cluster_identifier
    if self.db_user:
        function_kwargs["DbUser"] = self.db_user
    if self.database_credentials_secret_arn:
        function_kwargs["SecretArn"] = self.database_credentials_secret_arn

    response = self.client.execute_statement(**function_kwargs)  # pyright:␣
↪ignore[reportArgumentType]
    response_id = response["Id"]

    statement = self.client.describe_statement(Id=response_id)

    while statement["Status"] in ["SUBMITTED", "PICKED", "STARTED"]:
        time.sleep(self.wait_for_query_completion_sec)
        statement = self.client.describe_statement(Id=response_id)

    if statement["Status"] == "FINISHED":
        statement_result = self.client.get_statement_result(Id=response_id)
        results = statement_result.get("Records", [])

        while "NextToken" in statement_result:
            statement_result = self.client.get_statement_result(
                Id=response_id,
                NextToken=statement_result["NextToken"],
            )
            results = results + response.get("Records", [])

        return self._post_process(statement_result["ColumnMetadata"], results)  #␣
↪pyright: ignore[reportArgumentType]

    if statement["Status"] in ["FAILED", "ABORTED"]:
        return None
    return None
```

get_table_schema(table_name, schema=None)

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]:
    function_kwargs = {"Database": self.database, "Table": table_name}
    if schema:
        function_kwargs["Schema"] = schema
    if self.workgroup_name:
        function_kwargs["WorkgroupName"] = self.workgroup_name
    if self.cluster_identifier:
        function_kwargs["ClusterIdentifier"] = self.cluster_identifier
    if self.db_user:
```

(continues on next page)

(continued from previous page)

```
        function_kwargs["DbUser"] = self.db_user
    if self.database_credentials_secret_arn:
        function_kwargs["SecretArn"] = self.database_credentials_secret_arn
    response = self.client.describe_table(**function_kwargs)  # pyright:␣
↪ignore[reportArgumentType]
    return str([col["name"] for col in response["ColumnList"] if "name" in col])
```

validate*params(*, workgroup_name)

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@workgroup_name.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_params(self, _: Attribute, workgroup_name: Optional[str]) -> None:
    if not self.cluster_identifier and not self.workgroup_name:
        raise ValueError("Provide a value for one of `cluster_identifier` or␣
↪`workgroup_name`")
    if self.cluster_identifier and self.workgroup_name:
        raise ValueError("Provide a value for either `cluster_identifier` or␣
↪`workgroup_name`, but not both")
```

AmazonS3FileManagerDriver

Bases: *BaseFileManagerDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.AmazonS3FileManagerDriver.session" title="session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True) class-attribute instance-attribute (griptape.drivers.AmazonS3FileManagerDriver.session)"><code class="language-python">	*Session*	The boto3 session to use for S3 operations.

continues on next page

Table 41 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.AmazonS3FileManagerDriver.bucket" title="bucket = field(kw_only=True) class-attribute instance-attribute (griptape.drivers.AmazonS3FileManagerDriver.bucket)"><code class="language-python">	*str*	The name of the S3 bucket.
<a href="./#griptape.drivers.AmazonS3FileManagerDriver.workdir" title="workdir property writable (griptape.drivers.AmazonS3FileManagerDriver.workdir)"><code class="language-python">	*str*	The absolute working directory (must start with "/"). List, load, and save operations will be performed relative to this directory.
<a href="./#griptape.drivers.AmazonS3FileManagerDriver.client" title="client() (griptape.drivers.AmazonS3FileManagerDriver.client)"><code class="language-python">	*S3Client*	The S3 client to use for S3 operations.

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
@define
class AmazonS3FileManagerDriver(BaseFileManagerDriver):
    """AmazonS3FileManagerDriver can be used to list, load, and save files in an␣
↪Amazon S3 bucket.

    Attributes:
        session: The boto3 session to use for S3 operations.
        bucket: The name of the S3 bucket.
        workdir: The absolute working directory (must start with "/"). List, load,␣
↪and save
            operations will be performed relative to this directory.
        client: The S3 client to use for S3 operations.
    """

    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
```

(continues on next page)

(continued from previous page)

```
    bucket: str = field(kw_only=True)
    _workdir: str = field(default="/", kw_only=True, alias="workdir")
    _client: Optional[S3Client] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @property
    def workdir(self) -> str:
        if self._workdir.startswith("/"):
            return self._workdir
        return f"/{self._workdir}"

    @workdir.setter
    def workdir(self, value: str) -> None:
        self._workdir = value

    @lazy_property()
    def client(self) -> S3Client:
        return self.session.client("s3")

    def try_list_files(self, path: str) -> list[str]:
        full_key = self._to_dir_full_key(path)
        files_and_dirs = self._list_files_and_dirs(full_key)
        if len(files_and_dirs) == 0:
            if len(self._list_files_and_dirs(full_key.rstrip("/"), max_items=1)) > 0:
                raise NotADirectoryError
            raise FileNotFoundError
        return files_and_dirs

    def try_load_file(self, path: str) -> bytes:
        botocore = import_optional_dependency("botocore")
        full_key = self._to_full_key(path)

        if self._is_a_directory(full_key):
            raise IsADirectoryError

        try:
            response = self.client.get_object(Bucket=self.bucket, Key=full_key)
            return response["Body"].read()
        except botocore.exceptions.ClientError as e:
            if e.response["Error"]["Code"] in {"NoSuchKey", "404"}:
                raise FileNotFoundError from e
            raise e

    def try_save_file(self, path: str, value: bytes) -> str:
        full_key = self._to_full_key(path)
        if self._is_a_directory(full_key):
            raise IsADirectoryError
        self.client.put_object(Bucket=self.bucket, Key=full_key, Body=value)
        return f"s3://{self.bucket}/{full_key}"

    def _to_full_key(self, path: str) -> str:
        path = path.lstrip("/")
```

(continues on next page)

(continued from previous page)

```
        full_key = f"{self.workdir}/{path}"
        # Need to keep the trailing slash if it was there,
        # because it means the path is a directory.
        ended_with_slash = path.endswith("/")

        full_key = self._normpath(full_key)

        if ended_with_slash:
            full_key += "/"
        return full_key.lstrip("/")

    def _to_dir_full_key(self, path: str) -> str:
        full_key = self._to_full_key(path)
        # S3 "directories" always end with a slash, except for the root.
        if full_key != "" and not full_key.endswith("/"):
            full_key += "/"
        return full_key

    def _list_files_and_dirs(self, full_key: str, **kwargs) -> list[str]:
        max_items = kwargs.get("max_items")
        pagination_config: PaginatorConfigTypeDef = {}
        if max_items is not None:
            pagination_config["MaxItems"] = max_items

        paginator = self.client.get_paginator("list_objects_v2")
        pages = paginator.paginate(
            Bucket=self.bucket,
            Prefix=full_key,
            Delimiter="/",
            PaginationConfig=pagination_config,
        )
        files_and_dirs = []
        for page in pages:
            for obj in page.get("CommonPrefixes", []):
                prefix = obj.get("Prefix", "")
                directory = prefix[len(full_key) :].rstrip("/")
                files_and_dirs.append(directory)

            for obj in page.get("Contents", []):
                key = obj.get("Key", "")
                file = key[len(full_key) :]
                files_and_dirs.append(file)
        return files_and_dirs

    def _is_a_directory(self, full_key: str) -> bool:
        botocore = import_optional_dependency("botocore")
        if full_key == "" or full_key.endswith("/"):
            return True

        try:
            self.client.head_object(Bucket=self.bucket, Key=full_key)
        except botocore.exceptions.ClientError as e:
```

(continues on next page)

(continued from previous page)

```
            if e.response["Error"]["Code"] in {"NoSuchKey", "404"}:
                return len(self._list_files_and_dirs(full_key, max_items=1)) > 0
            raise e

        return False

    def _normpath(self, path: str) -> str:
        unix_path = path.replace("\", "/")
        parts = unix_path.split("/")
        stack = []

        for part in parts:
            if part == "" or part == ".":
                continue
            if part == "..":
                if stack:
                    stack.pop()
            else:
                stack.append(part)

        return "/".join(stack)
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `_workdir = field(default='/', kw_only=True, alias='workdir')` class-attribute instance-attribute
- `bucket = field(kw_only=True)` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `workdir` property writable

_is_a_directory(full_key)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def _is_a_directory(self, full_key: str) -> bool:
    botocore = import_optional_dependency("botocore")
    if full_key == "" or full_key.endswith("/"):
        return True

    try:
        self.client.head_object(Bucket=self.bucket, Key=full_key)
    except botocore.exceptions.ClientError as e:
        if e.response["Error"]["Code"] in {"NoSuchKey", "404"}:
            return len(self._list_files_and_dirs(full_key, max_items=1)) > 0
        raise e

    return False
```

_list_files_and_dirs(full_key, **kwargs)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def _list_files_and_dirs(self, full_key: str, **kwargs) -> list[str]:
    max_items = kwargs.get("max_items")
    pagination_config: PaginatorConfigTypeDef = {}
    if max_items is not None:
        pagination_config["MaxItems"] = max_items

    paginator = self.client.get_paginator("list_objects_v2")
    pages = paginator.paginate(
        Bucket=self.bucket,
        Prefix=full_key,
        Delimiter="/",
        PaginationConfig=pagination_config,
    )
    files_and_dirs = []
    for page in pages:
        for obj in page.get("CommonPrefixes", []):
            prefix = obj.get("Prefix", "")
            directory = prefix[len(full_key) :].rstrip("/")
            files_and_dirs.append(directory)

        for obj in page.get("Contents", []):
            key = obj.get("Key", "")
            file = key[len(full_key) :]
            files_and_dirs.append(file)
    return files_and_dirs
```

_normpath(path)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def _normpath(self, path: str) -> str:
    unix_path = path.replace("\", "/")
    parts = unix_path.split("/")
    stack = []

    for part in parts:
        if part == "" or part == ".":
            continue
        if part == "..":
            if stack:
                stack.pop()
        else:
            stack.append(part)

    return "/".join(stack)
```

_to_dir_full_key(path)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def _to_dir_full_key(self, path: str) -> str:
    full_key = self._to_full_key(path)
    # S3 "directories" always end with a slash, except for the root.
    if full_key != "" and not full_key.endswith("/"):
        full_key += "/"
    return full_key
```

_to_full_key(path)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def _to_full_key(self, path: str) -> str:
    path = path.lstrip("/")
    full_key = f"{self.workdir}/{path}"
    # Need to keep the trailing slash if it was there,
    # because it means the path is a directory.
    ended_with_slash = path.endswith("/")

    full_key = self._normpath(full_key)

    if ended_with_slash:
        full_key += "/"
    return full_key.lstrip("/")
```

client()

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
@lazy_property()
def client(self) -> S3Client:
    return self.session.client("s3")
```

try_list_files(path)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def try_list_files(self, path: str) -> list[str]:
    full_key = self._to_dir_full_key(path)
    files_and_dirs = self._list_files_and_dirs(full_key)
    if len(files_and_dirs) == 0:
        if len(self._list_files_and_dirs(full_key.rstrip("/"), max_items=1)) > 0:
            raise NotADirectoryError
        raise FileNotFoundError
    return files_and_dirs
```

try_load_file(path)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def try_load_file(self, path: str) -> bytes:
    botocore = import_optional_dependency("botocore")
    full_key = self._to_full_key(path)

    if self._is_a_directory(full_key):
        raise IsADirectoryError

    try:
        response = self.client.get_object(Bucket=self.bucket, Key=full_key)
        return response["Body"].read()
    except botocore.exceptions.ClientError as e:
        if e.response["Error"]["Code"] in {"NoSuchKey", "404"}:
            raise FileNotFoundError from e
        raise e
```

try_save_file(path, value)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def try_save_file(self, path: str, value: bytes) -> str:
    full_key = self._to_full_key(path)
    if self._is_a_directory(full_key):
        raise IsADirectoryError
    self.client.put_object(Bucket=self.bucket, Key=full_key, Body=value)
    return f"s3://{self.bucket}/{full_key}"
```

AmazonSageMakerJumpstartEmbeddingDriver

Bases: *BaseEmbeddingDriver* “)

Source Code in griptape/drivers/embedding/amazon_sagemaker_jumpstart_embedding_driver.py

```
@define
class AmazonSageMakerJumpstartEmbeddingDriver(BaseEmbeddingDriver):
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    endpoint: str = field(kw_only=True, metadata={"serializable": True})
    custom_attributes: str = field(default="accept_eula=true", kw_only=True, metadata=
↪{"serializable": True})
    inference_component_name: Optional[str] = field(default=None, kw_only=True,␣
↪metadata={"serializable": True})
    _client: Optional[SageMakerRuntimeClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> SageMakerRuntimeClient:
        return self.session.client("sagemaker-runtime")

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        payload = {"text_inputs": chunk, "mode": "embedding"}

        endpoint_response = self.client.invoke_endpoint(
```

(continues on next page)

(continued from previous page)

```
            EndpointName=self.endpoint,
            ContentType="application/json",
            Body=json.dumps(payload).encode("utf-8"),
            CustomAttributes=self.custom_attributes,
            **(
                {"InferenceComponentName": self.inference_component_name}
                if self.inference_component_name is not None
                else {}
            ),
        )

        response = json.loads(endpoint_response.get("Body").read().decode("utf-8"))

        if "embedding" in response:
            embedding = response["embedding"]

            if embedding:
                if isinstance(embedding[0], list):
                    return embedding[0]
                return embedding
            raise ValueError("model response is empty")
        raise ValueError("invalid response from model")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `custom_attributes = field(default='accept_eula=true', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `inference_component_name = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/amazon_sagemaker_jumpstart_embedding_driver.py

```
@lazy_property()
def client(self) -> SageMakerRuntimeClient:
    return self.session.client("sagemaker-runtime")
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/amazon_sagemaker_jumpstart_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    payload = {"text_inputs": chunk, "mode": "embedding"}

    endpoint_response = self.client.invoke_endpoint(
```

(continues on next page)

(continued from previous page)

```
        EndpointName=self.endpoint,
        ContentType="application/json",
        Body=json.dumps(payload).encode("utf-8"),
        CustomAttributes=self.custom_attributes,
        **(
            {"InferenceComponentName": self.inference_component_name}
            if self.inference_component_name is not None
            else {}
        ),
    )

    response = json.loads(endpoint_response.get("Body").read().decode("utf-8"))

    if "embedding" in response:
        embedding = response["embedding"]

        if embedding:
            if isinstance(embedding[0], list):
                return embedding[0]
            return embedding
        raise ValueError("model response is empty")
    raise ValueError("invalid response from model")
```

AmazonSageMakerJumpstartPromptDriver

Bases: *BasePromptDriver* “)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
@define
class AmazonSageMakerJumpstartPromptDriver(BasePromptDriver):
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    endpoint: str = field(kw_only=True, metadata={"serializable": True})
    custom_attributes: str = field(default="accept_eula=true", kw_only=True, metadata=
↪{"serializable": True})
    inference_component_name: Optional[str] = field(default=None, kw_only=True,␣
↪metadata={"serializable": True})
    stream: bool = field(default=False, kw_only=True, metadata={"serializable": True})
    max_tokens: int = field(default=250, kw_only=True, metadata={"serializable": True}
↪)
    tokenizer: HuggingFaceTokenizer = field(
        default=Factory(
            lambda self: HuggingFaceTokenizer(model=self.model, max_output_
↪tokens=self.max_tokens),
            takes_self=True,
        ),
        kw_only=True,
    )
    structured_output_strategy: StructuredOutputStrategy = field(
        default="rule", kw_only=True, metadata={"serializable": True}
    )
```

(continues on next page)

(continued from previous page)

```
    _client: Optional[Any] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @structured_output_strategy.validator  # pyright:␣
↪ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
        if value != "rule":
            raise ValueError(f"{__class__.__name__} does not support `{value}`␣
↪structured output strategy.")

        return value

    @lazy_property()
    def client(self) -> Any:
        return self.session.client("sagemaker-runtime")

    @stream.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_stream(self, _: Attribute, stream: bool) -> None:  # noqa: FBT001
        if stream:
            raise ValueError("streaming is not supported")

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        payload = {
            "inputs": self.prompt_stack_to_string(prompt_stack),
            "parameters": {**self._base_params(prompt_stack)},
        }
        logger.debug(payload)

        response = self.client.invoke_endpoint(
            EndpointName=self.endpoint,
            ContentType="application/json",
            Body=json.dumps(payload),
            CustomAttributes=self.custom_attributes,
            **(
                {"InferenceComponentName": self.inference_component_name}
                if self.inference_component_name is not None
                else {}
            ),
        )

        decoded_body = json.loads(response["Body"].read().decode("utf8"))
        logger.debug(decoded_body)

        if isinstance(decoded_body, list):
            if decoded_body:
                generated_text = decoded_body[0]["generated_text"]
            else:
                raise ValueError("model response is empty")
        else:
            generated_text = decoded_body["generated_text"]
```

(continues on next page)

(continued from previous page)

```
        input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))
        output_tokens = len(self.tokenizer.tokenizer.encode(generated_text))  #␣
↪pyright: ignore[reportArgumentType]

        return Message(
            content=[TextMessageContent(TextArtifact(generated_text))],
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(input_tokens=input_tokens, output_tokens=output_
↪tokens),
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        raise NotImplementedError("streaming is not supported")

    def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
        return self.tokenizer.tokenizer.decode(self.__prompt_stack_to_tokens(prompt_
↪stack))  # pyright: ignore[reportArgumentType]

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        return {
            "temperature": self.temperature,
            "max_new_tokens": self.max_tokens,
            "do_sample": True,
            "eos_token_id": self.tokenizer.tokenizer.eos_token_id,
            "stop_strings": self.tokenizer.stop_sequences,
            "return_full_text": False,
            **self.extra_params,
        }

    def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
        messages = []

        for message in prompt_stack.messages:
            messages.append({"role": message.role, "content": message.to_text()})

        return messages

    def __prompt_stack_to_tokens(self, prompt_stack: PromptStack) -> list[int]:
        messages = self._prompt_stack_to_messages(prompt_stack)

        tokens = self.tokenizer.tokenizer.apply_chat_template(messages, add_
↪generation_prompt=True, tokenize=True)

        if isinstance(tokens, list):
            return tokens  # pyright: ignore[reportReturnType] According to the␣
↪[docs](https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils
↪#transformers.PreTrainedTokenizerBase.apply_chat_template), the return type is␣
↪List[int].
        raise ValueError("Invalid output type.")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute

- `custom_attributes = field(default='accept_eula=true', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `inference_component_name = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `max_tokens = field(default=250, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `stream = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='rule', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: HuggingFaceTokenizer(model=self.model, max_output_tokens=self.max_tokens), takes_self=True), kw_only=True)` class-attribute instance-attribute

__prompt_stack_to_tokens(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
def __prompt_stack_to_tokens(self, prompt_stack: PromptStack) -> list[int]:
    messages = self._prompt_stack_to_messages(prompt_stack)

    tokens = self.tokenizer.tokenizer.apply_chat_template(messages, add_generation_
↪prompt=True, tokenize=True)

    if isinstance(tokens, list):
        return tokens  # pyright: ignore[reportReturnType] According to the␣
↪[docs](https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils
↪#transformers.PreTrainedTokenizerBase.apply_chat_template), the return type is␣
↪List[int].
    raise ValueError("Invalid output type.")
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    return {
        "temperature": self.temperature,
        "max_new_tokens": self.max_tokens,
        "do_sample": True,
        "eos_token_id": self.tokenizer.tokenizer.eos_token_id,
        "stop_strings": self.tokenizer.stop_sequences,
        "return_full_text": False,
        **self.extra_params,
    }
```

_prompt_stack_to_messages(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
    messages = []

    for message in prompt_stack.messages:
        messages.append({"role": message.role, "content": message.to_text()})

    return messages
```

client()

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
@lazy_property()
def client(self) -> Any:
    return self.session.client("sagemaker-runtime")
```

prompt_stack_to_string(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
    return self.tokenizer.tokenizer.decode(self.__prompt_stack_to_tokens(prompt_
↪stack))  # pyright: ignore[reportArgumentType]
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    payload = {
        "inputs": self.prompt_stack_to_string(prompt_stack),
        "parameters": {**self._base_params(prompt_stack)},
    }
    logger.debug(payload)

    response = self.client.invoke_endpoint(
        EndpointName=self.endpoint,
        ContentType="application/json",
        Body=json.dumps(payload),
        CustomAttributes=self.custom_attributes,
        **(
            {"InferenceComponentName": self.inference_component_name}
            if self.inference_component_name is not None
            else {}
        ),
    )

    decoded_body = json.loads(response["Body"].read().decode("utf8"))
    logger.debug(decoded_body)
```

(continues on next page)

(continued from previous page)

```
    if isinstance(decoded_body, list):
        if decoded_body:
            generated_text = decoded_body[0]["generated_text"]
        else:
            raise ValueError("model response is empty")
    else:
        generated_text = decoded_body["generated_text"]

    input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))
    output_tokens = len(self.tokenizer.tokenizer.encode(generated_text))  # pyright:␣
↪ignore[reportArgumentType]

    return Message(
        content=[TextMessageContent(TextArtifact(generated_text))],
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(input_tokens=input_tokens, output_tokens=output_tokens),
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    raise NotImplementedError("streaming is not supported")
```

validate*stream(*, stream)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
@stream.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_stream(self, _: Attribute, stream: bool) -> None:  # noqa: FBT001
    if stream:
        raise ValueError("streaming is not supported")
```

validate*structured_output_strategy(*, value)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
@structured_output_strategy.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
    if value != "rule":
        raise ValueError(f"{__class__.__name__} does not support `{value}` structured␣
↪output strategy.")

    return value
```

AmazonSqsEventListenerDriver

Bases: *BaseEventListenerDriver* “)

Source Code in griptape/drivers/event_listener/amazon_sqs_event_listener_driver.py

```
@define
class AmazonSqsEventListenerDriver(BaseEventListenerDriver):
    queue_url: str = field(kw_only=True)
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    _client: Optional[SQSClient] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> SQSClient:
        return self.session.client("sqs")

    def try_publish_event_payload(self, event_payload: dict) -> None:
        self.client.send_message(QueueUrl=self.queue_url, MessageBody=json.
↪dumps(event_payload))

    def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None:
        entries: Sequence[SendMessageBatchRequestEntryTypeDef] = [
            {"Id": str(event_payload["id"]), "MessageBody": json.dumps(event_payload)}
            for event_payload in event_payload_batch
        ]

        self.client.send_message_batch(QueueUrl=self.queue_url, Entries=entries)
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `queue_url = field(kw_only=True)` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/event_listener/amazon_sqs_event_listener_driver.py

```
@lazy_property()
def client(self) -> SQSClient:
    return self.session.client("sqs")
```

try_publish_event_payload(event_payload)

Source Code in griptape/drivers/event_listener/amazon_sqs_event_listener_driver.py

```
def try_publish_event_payload(self, event_payload: dict) -> None:
    self.client.send_message(QueueUrl=self.queue_url, MessageBody=json.dumps(event_
↪payload))
```

try_publish_event_payload_batch(event_payload_batch)

Source Code in griptape/drivers/event_listener/amazon_sqs_event_listener_driver.py

```
def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None:
    entries: Sequence[SendMessageBatchRequestEntryTypeDef] = [
        {"Id": str(event_payload["id"]), "MessageBody": json.dumps(event_payload)}
        for event_payload in event_payload_batch
    ]

    self.client.send_message_batch(QueueUrl=self.queue_url, Entries=entries)
```

AnthropicPromptDriver

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.AnthropicPromptDriver.api_key" title="api_key = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.AnthropicPromptDriver.api_key)"><code class="language-python">	*Optional[str]*	Anthropic API key.
<a href="./#griptape.drivers.AnthropicPromptDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.AnthropicPromptDriver.model)"><code class="language-python">	*str*	Anthropic model name.

continues on next page

Table 42 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.AnthropicPromptDriver.client" title="client() (griptape.drivers.AnthropicPromptDriver.client)"><code class="language-python">	*Client*	Custom *Anthropic* client.

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
@define
class AnthropicPromptDriver(BasePromptDriver):
    """Anthropic Prompt Driver.

    Attributes:
        api_key: Anthropic API key.
        model: Anthropic model name.
        client: Custom `Anthropic` client.
    """

    api_key: Optional[str] = field(kw_only=True, default=None, metadata={"serializable
↪": False})
    model: str = field(kw_only=True, metadata={"serializable": True})
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: AnthropicTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
    top_p: float = field(default=0.999, kw_only=True, metadata={"serializable": True})
    top_k: int = field(default=250, kw_only=True, metadata={"serializable": True})
    tool_choice: dict = field(default=Factory(lambda: {"type": "auto"}), kw_only=True,
↪ metadata={"serializable": False})
    use_native_tools: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})
    structured_output_strategy: StructuredOutputStrategy = field(
        default="tool", kw_only=True, metadata={"serializable": True}
    )
    max_tokens: int = field(default=1000, kw_only=True, metadata={"serializable":␣
↪True})
    _client: Optional[Client] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> Client:
        return import_optional_dependency("anthropic").Anthropic(api_key=self.api_key)

    @structured_output_strategy.validator  # pyright:␣
↪ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
```

(continues on next page)

(continued from previous page)

```
        if value == "native":
            raise ValueError(f"{__class__.__name__} does not support `{value}` structured output strategy.")

        return value

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        params = self._base_params(prompt_stack)
        logger.debug(params)
        response = self.client.messages.create(**params)

        logger.debug(response.model_dump())

        return Message(
            content=[self.__to_prompt_stack_message_content(content) for content in response.content],
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(input_tokens=response.usage.input_tokens, output_tokens=response.usage.output_tokens),
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        params = {**self._base_params(prompt_stack), "stream": True}
        logger.debug(params)
        events = self.client.messages.create(**params)

        for event in events:
            logger.debug(event)
            if event.type == "content_block_delta" or event.type == "content_block_start":
                yield DeltaMessage(content=self.__to_prompt_stack_delta_message_content(event))
            elif event.type == "message_start":
                yield DeltaMessage(usage=DeltaMessage.Usage(input_tokens=event.message.usage.input_tokens))
            elif event.type == "message_delta":
                yield DeltaMessage(usage=DeltaMessage.Usage(output_tokens=event.usage.output_tokens))

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        messages = self.__to_anthropic_messages([i for i in prompt_stack.messages if not i.is_system()])

        system_messages = prompt_stack.system_messages
        system_message = system_messages[0].to_text() if system_messages else None

        params = {
            "model": self.model,
            "temperature": self.temperature,
            "stop_sequences": self.tokenizer.stop_sequences,
```

(continues on next page)

(continued from previous page)

```
            "top_p": self.top_p,
            "top_k": self.top_k,
            "max_tokens": self.max_tokens,
            "messages": messages,
            **({"system": system_message} if system_message else {}),
            **self.extra_params,
        }

        if prompt_stack.tools and self.use_native_tools:
            params["tool_choice"] = self.tool_choice

            if prompt_stack.output_schema is not None and self.structured_output_
↪strategy == "tool":
                params["tool_choice"] = {"type": "any"}

            params["tools"] = self.__to_anthropic_tools(prompt_stack.tools)

        return params

    def __to_anthropic_messages(self, messages: list[Message]) -> list[dict]:
        return [
            {"role": self.__to_anthropic_role(message), "content": self.__to_
↪anthropic_content(message)}
            for message in messages
        ]

    def __to_anthropic_role(self, message: Message) -> str:
        if message.is_assistant():
            return "assistant"
        return "user"

    def __to_anthropic_tools(self, tools: list[BaseTool]) -> list[dict]:
        tool_schemas = [
            {
                "name": tool.to_native_tool_name(activity),
                "description": tool.activity_description(activity),
                "input_schema": tool.to_activity_json_schema(activity, "Input Schema
↪"),
            }
            for tool in tools
            for activity in tool.activities()
        ]

        # Anthropic doesnt support $schema and $id
        for tool_schema in tool_schemas:
            del tool_schema["input_schema"]["$schema"]
            del tool_schema["input_schema"]["$id"]

        return tool_schemas

    def __to_anthropic_content(self, message: Message) -> str | list[dict]:
        if message.has_all_content_type(TextMessageContent):
```

(continues on next page)

(continued from previous page)

```
            return message.to_text()
        return [self.__to_anthropic_message_content(content) for content in message.
↪content]

    def __to_anthropic_message_content(self, content: BaseMessageContent) -> dict:
        if isinstance(content, TextMessageContent):
            return {"type": "text", "text": content.artifact.value}
        if isinstance(content, ImageMessageContent):
            artifact = content.artifact
            if isinstance(artifact, ImageArtifact):
                return {
                    "type": "image",
                    "source": {"type": "base64", "media_type": artifact.mime_type,
↪"data": artifact.base64},
                }
            if isinstance(artifact, ImageUrlArtifact):
                return {
                    "type": "image",
                    "source": {"type": "url", "url": artifact.value},
                }
            raise ValueError(f"Unsupported image artifact type: {type(artifact)}")
        if isinstance(content, ActionCallMessageContent):
            action = content.artifact.value

            return {"type": "tool_use", "id": action.tag, "name": action.to_native_
↪tool_name(), "input": action.input}
        if isinstance(content, ActionResultMessageContent):
            artifact = content.artifact

            if isinstance(artifact, ListArtifact):
                message_content = [self.__to_anthropic_tool_result_content(artifact)␣
↪for artifact in artifact.value]
            else:
                message_content = [self.__to_anthropic_tool_result_content(artifact)]

            return {
                "type": "tool_result",
                "tool_use_id": content.action.tag,
                "content": message_content,
                "is_error": isinstance(artifact, ErrorArtifact),
            }
        return content.artifact.value

    def __to_anthropic_tool_result_content(self, artifact: BaseArtifact) -> dict:
        if isinstance(artifact, ImageArtifact):
            return {
                "type": "image",
                "source": {"type": "base64", "media_type": artifact.mime_type, "data
↪": artifact.base64},
            }
        if isinstance(artifact, (TextArtifact, ErrorArtifact, InfoArtifact)):
            return {"type": "text", "text": artifact.to_text()}
```

(continues on next page)

(continued from previous page)

```
        raise ValueError(f"Unsupported tool result artifact type: {type(artifact)}")

    def __to_prompt_stack_message_content(self, content: ContentBlock) ->␣
↪BaseMessageContent:
        if content.type == "text":
            return TextMessageContent(TextArtifact(content.text))
        if content.type == "tool_use":
            name, path = ToolAction.from_native_tool_name(content.name)

            return ActionCallMessageContent(
                artifact=ActionArtifact(
                    value=ToolAction(tag=content.id, name=name, path=path,␣
↪input=content.input),  # pyright: ignore[reportArgumentType]
                ),
            )
        raise ValueError(f"Unsupported message content type: {content.type}")

    def __to_prompt_stack_delta_message_content(
        self,
        event: ContentBlockDeltaEvent | ContentBlockStartEvent,
    ) -> BaseDeltaMessageContent:
        if event.type == "content_block_start":
            content_block = event.content_block

            if content_block.type == "tool_use":
                name, path = ToolAction.from_native_tool_name(content_block.name)

                return ActionCallDeltaMessageContent(index=event.index, tag=content_
↪block.id, name=name, path=path)
            if content_block.type == "text":
                return TextDeltaMessageContent(content_block.text, index=event.index)
            raise ValueError(f"Unsupported content block type: {content_block.type}")
        if event.type == "content_block_delta":
            content_block_delta = event.delta

            if content_block_delta.type == "text_delta":
                return TextDeltaMessageContent(content_block_delta.text, index=event.
↪index)
            if content_block_delta.type == "input_json_delta":
                return ActionCallDeltaMessageContent(index=event.index, partial_
↪input=content_block_delta.partial_json)
            raise ValueError(f"Unsupported message content type: {event}")
        raise ValueError(f"Unsupported message content type: {event}")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `max_tokens = field(default=1000, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute

instance-attribute

- `structured_output_strategy = field(default='tool', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: AnthropicTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `tool_choice = field(default=Factory(lambda: {'type': 'auto'}), kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `top_k = field(default=250, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `top_p = field(default=0.999, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `use_native_tools = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__to_anthropic_content(message)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_anthropic_content(self, message: Message) -> str | list[dict]:
    if message.has_all_content_type(TextMessageContent):
        return message.to_text()
    return [self.__to_anthropic_message_content(content) for content in message.
↪content]
```

__to_anthropic_message_content(content)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_anthropic_message_content(self, content: BaseMessageContent) -> dict:
    if isinstance(content, TextMessageContent):
        return {"type": "text", "text": content.artifact.value}
    if isinstance(content, ImageMessageContent):
        artifact = content.artifact
        if isinstance(artifact, ImageArtifact):
            return {
                "type": "image",
                "source": {"type": "base64", "media_type": artifact.mime_type, "data
↪": artifact.base64},
            }
        if isinstance(artifact, ImageUrlArtifact):
            return {
                "type": "image",
                "source": {"type": "url", "url": artifact.value},
            }
        raise ValueError(f"Unsupported image artifact type: {type(artifact)}")
    if isinstance(content, ActionCallMessageContent):
        action = content.artifact.value

        return {"type": "tool_use", "id": action.tag, "name": action.to_native_tool_
↪name(), "input": action.input}
    if isinstance(content, ActionResultMessageContent):
```

(continues on next page)

(continued from previous page)

```
        artifact = content.artifact

        if isinstance(artifact, ListArtifact):
            message_content = [self.__to_anthropic_tool_result_content(artifact) for␣
↪artifact in artifact.value]
        else:
            message_content = [self.__to_anthropic_tool_result_content(artifact)]

        return {
            "type": "tool_result",
            "tool_use_id": content.action.tag,
            "content": message_content,
            "is_error": isinstance(artifact, ErrorArtifact),
        }
    return content.artifact.value
```

__to_anthropic_messages(messages)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_anthropic_messages(self, messages: list[Message]) -> list[dict]:
    return [
        {"role": self.__to_anthropic_role(message), "content": self.__to_anthropic_
↪content(message)}
        for message in messages
    ]
```

__to_anthropic_role(message)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_anthropic_role(self, message: Message) -> str:
    if message.is_assistant():
        return "assistant"
    return "user"
```

__to_anthropic_tool_result_content(artifact)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_anthropic_tool_result_content(self, artifact: BaseArtifact) -> dict:
    if isinstance(artifact, ImageArtifact):
        return {
            "type": "image",
            "source": {"type": "base64", "media_type": artifact.mime_type, "data":␣
↪artifact.base64},
        }
    if isinstance(artifact, (TextArtifact, ErrorArtifact, InfoArtifact)):
        return {"type": "text", "text": artifact.to_text()}
    raise ValueError(f"Unsupported tool result artifact type: {type(artifact)}")
```

__to_anthropic_tools(tools)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_anthropic_tools(self, tools: list[BaseTool]) -> list[dict]:
    tool_schemas = [
        {
            "name": tool.to_native_tool_name(activity),
            "description": tool.activity_description(activity),
            "input_schema": tool.to_activity_json_schema(activity, "Input Schema"),
        }
        for tool in tools
        for activity in tool.activities()
    ]

    # Anthropic doesnt support $schema and $id
    for tool_schema in tool_schemas:
        del tool_schema["input_schema"]["$schema"]
        del tool_schema["input_schema"]["$id"]

    return tool_schemas
```

__to_prompt_stack_delta_message_content(event)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_prompt_stack_delta_message_content(
    self,
    event: ContentBlockDeltaEvent | ContentBlockStartEvent,
) -> BaseDeltaMessageContent:
    if event.type == "content_block_start":
        content_block = event.content_block

        if content_block.type == "tool_use":
            name, path = ToolAction.from_native_tool_name(content_block.name)

            return ActionCallDeltaMessageContent(index=event.index, tag=content_block.
↪id, name=name, path=path)
        if content_block.type == "text":
            return TextDeltaMessageContent(content_block.text, index=event.index)
        raise ValueError(f"Unsupported content block type: {content_block.type}")
    if event.type == "content_block_delta":
        content_block_delta = event.delta

        if content_block_delta.type == "text_delta":
            return TextDeltaMessageContent(content_block_delta.text, index=event.
↪index)
        if content_block_delta.type == "input_json_delta":
            return ActionCallDeltaMessageContent(index=event.index, partial_
↪input=content_block_delta.partial_json)
        raise ValueError(f"Unsupported message content type: {event}")
    raise ValueError(f"Unsupported message content type: {event}")
```

__to_prompt_stack_message_content(content)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_prompt_stack_message_content(self, content: ContentBlock) ->␣
↪BaseMessageContent:
    if content.type == "text":
        return TextMessageContent(TextArtifact(content.text))
    if content.type == "tool_use":
        name, path = ToolAction.from_native_tool_name(content.name)

        return ActionCallMessageContent(
            artifact=ActionArtifact(
                value=ToolAction(tag=content.id, name=name, path=path, input=content.
↪input),  # pyright: ignore[reportArgumentType]
            ),
        )
    raise ValueError(f"Unsupported message content type: {content.type}")
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    messages = self.__to_anthropic_messages([i for i in prompt_stack.messages if not␣
↪i.is_system()])

    system_messages = prompt_stack.system_messages
    system_message = system_messages[0].to_text() if system_messages else None

    params = {
        "model": self.model,
        "temperature": self.temperature,
        "stop_sequences": self.tokenizer.stop_sequences,
        "top_p": self.top_p,
        "top_k": self.top_k,
        "max_tokens": self.max_tokens,
        "messages": messages,
        **({"system": system_message} if system_message else {}),
        **self.extra_params,
    }

    if prompt_stack.tools and self.use_native_tools:
        params["tool_choice"] = self.tool_choice

        if prompt_stack.output_schema is not None and self.structured_output_strategy␣
↪== "tool":
            params["tool_choice"] = {"type": "any"}

        params["tools"] = self.__to_anthropic_tools(prompt_stack.tools)

    return params
```

client()

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
@lazy_property()
def client(self) -> Client:
    return import_optional_dependency("anthropic").Anthropic(api_key=self.api_key)
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    params = self._base_params(prompt_stack)
    logger.debug(params)
    response = self.client.messages.create(**params)

    logger.debug(response.model_dump())

    return Message(
        content=[self.__to_prompt_stack_message_content(content) for content in␣
↪response.content],
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(input_tokens=response.usage.input_tokens, output_
↪tokens=response.usage.output_tokens),
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    params = {**self._base_params(prompt_stack), "stream": True}
    logger.debug(params)
    events = self.client.messages.create(**params)

    for event in events:
        logger.debug(event)
        if event.type == "content_block_delta" or event.type == "content_block_start":
            yield DeltaMessage(content=self.__to_prompt_stack_delta_message_
↪content(event))
        elif event.type == "message_start":
            yield DeltaMessage(usage=DeltaMessage.Usage(input_tokens=event.message.
↪usage.input_tokens))
        elif event.type == "message_delta":
            yield DeltaMessage(usage=DeltaMessage.Usage(output_tokens=event.usage.
↪output_tokens))
```

validate*structured_output_strategy*(, value)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
@structured_output_strategy.validator  # pyright: ignore[reportAttributeAccessIssue,
↪reportOptionalMemberAccess]
def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
    if value == "native":
        raise ValueError(f"{__class__.__name__} does not support `{value}` structured
↪output strategy.")

    return value
```

AstraDbVectorStoreDriver

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.BaseVectorStoreDriver.embedding_driver" title="embedding_driver = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.AstraDbVectorStoreDriver.embedding_driver)"><code class="language-python">	<a href="embedding#griptape.drivers.embedding.BaseEmbeddingDriver" title="BaseEmbeddingDriver (griptape.drivers.embedding.BaseEmbeddingDriver)"><code class="language-python">BaseEmbeddingDriver</cod	a *griptape.drivers.BaseEmbeddingDriver* for embedding computations within the store
<a href="./#griptape.drivers.AstraDbVectorStoreDriver.api_endpoint" title="api_endpoint = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.AstraDbVectorStoreDriver.api_endpoint)"><code class="language-python">	*str*	the "API Endpoint" for the Astra DB instance.

continues on next page

Table 43 – continued from previous page

Name	Type	Description
<a href="./#grip-tape.drivers.AstraDb-VectorStoreDriver.token" title="token = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (grip-tape.drivers.AstraDb-VectorStoreDriver.token)"><code class="language-python">	*Optional[str I TokenProvider]*	a Database Token ("AstraCS:...") secret to access Astra DB. An instance of *astrapy.authentication.TokenProvider* is also accepted.
<a href="./#grip-tape.drivers.AstraDb-VectorStoreDriver.collection_name" title="collection_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (grip-tape.drivers.AstraDb-VectorStoreDriver.collection_name)"><code class="language-python">	*str*	the name of the collection on Astra DB. The collection must have been created beforehand, and support vectors with a vector dimension matching the embeddings being used by this driver.
<a href="./#grip-tape.drivers.AstraDb-VectorStoreDriver.environment" title="environment = field(kw_only=True, default=None, metadata={'serializable': True}) class-attribute instance-attribute (grip-tape.drivers.AstraDb-VectorStoreDriver.environment)"><code class="language-python">	*Optional[str]*	the environment ("prod", "hcd", ...) hosting the target Data API. It can be omitted for production Astra DB targets. See *astrapy.constants.Environment* for allowed values.

continues on next page

Table 43 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.AstraDbVectorStoreDriver.astra_db_namespace" title="astra_db_namespace = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.AstraDbVectorStoreDriver.astra_db_namespace)"><code class="language-python">	*Optional[str]*	optional specification of the namespace (in the Astra database) for the data. *Note* : not to be confused with the "namespace" mentioned elsewhere, which is a grouping within this vector store.
<a href="./#griptape.drivers.AstraDbVectorStoreDriver.caller_name" title="caller_name = field(default='griptape', kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.AstraDbVectorStoreDriver.caller_name)" class="language-python">	*str*	the name of the caller for the Astra DB client. Defaults to "griptape".
<a href="./#griptape.drivers.AstraDbVectorStoreDriver.client" title="client() (griptape.drivers.AstraDbVectorStoreDriver.client)"><coc class="language-python">	*DataAPIClient*	an instance of *astrapy.DataAPIClient* for the Astra DB.
<a href="./#griptape.drivers.AstraDbVectorStoreDriver.collection" title="collection() (griptape.drivers.AstraDbVectorStoreDriver.collection)"><code class="language-python">	*Collection*	an instance of *astrapy.Collection* for the Astra DB.

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
@define
class AstraDbVectorStoreDriver(BaseVectorStoreDriver):
    """A Vector Store Driver for Astra DB.

    Attributes:
        embedding_driver: a `griptape.drivers.BaseEmbeddingDriver` for embedding␣
↪computations within the store
        api_endpoint: the "API Endpoint" for the Astra DB instance.
        token: a Database Token ("AstraCS:...") secret to access Astra DB. An␣
↪instance of `astrapy.authentication.TokenProvider` is also accepted.
        collection_name: the name of the collection on Astra DB. The collection must␣
↪have been created beforehand,
            and support vectors with a vector dimension matching the embeddings being␣
↪used by this driver.
        environment: the environment ("prod", "hcd", ...) hosting the target Data API.
            It can be omitted for production Astra DB targets. See `astrapy.constants.
↪Environment` for allowed values.
        astra_db_namespace: optional specification of the namespace (in the Astra␣
↪database) for the data.
            *Note*: not to be confused with the "namespace" mentioned elsewhere,␣
↪which is a grouping within this vector store.
        caller_name: the name of the caller for the Astra DB client. Defaults to
↪"griptape".
        client: an instance of `astrapy.DataAPIClient` for the Astra DB.
        collection: an instance of `astrapy.Collection` for the Astra DB.
    """

    api_endpoint: str = field(kw_only=True, metadata={"serializable": True})
    token: Optional[str | astrapy.authentication.TokenProvider] = field(
        kw_only=True, default=None, metadata={"serializable": False}
    )
    collection_name: str = field(kw_only=True, metadata={"serializable": True})
    environment: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": True})
    astra_db_namespace: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    caller_name: str = field(default="griptape", kw_only=True, metadata={"serializable
↪": False})
    _client: Optional[astrapy.DataAPIClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )
    _collection: Optional[astrapy.Collection] = field(
        default=None, kw_only=True, alias="collection", metadata={"serializable":␣
↪False}
    )

    @lazy_property()
    def client(self) -> astrapy.DataAPIClient:
        astrapy = import_optional_dependency("astrapy")

        return astrapy.DataAPIClient(
            callers=[(self.caller_name, None)],
            environment=self.environment or astrapy.utils.unset.UnsetType(),
```

(continues on next page)

(continued from previous page)

```
        )

    @lazy_property()
    def collection(self) -> astrapy.Collection:
        if self.token is None:
            raise ValueError("Astra DB token must be provided.")
        return self.client.get_database(
            self.api_endpoint, token=self.token, keyspace=self.astra_db_namespace
        ).get_collection(self.collection_name)

    def delete_vector(self, vector_id: str) -> None:
        """Delete a vector from Astra DB store.

        The method succeeds regardless of whether a vector with the provided ID
        was actually stored or not in the first place.

        Args:
            vector_id: ID of the vector to delete.
        """
        self.collection.delete_one({"_id": vector_id})

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs: Any,
    ) -> str:
        """Write a vector to the Astra DB store.

        In case the provided ID exists already, an overwrite will take place.

        Args:
            vector: the vector to be upserted.
            vector_id: the ID for the vector to store. If omitted, a server-provided␣
↪new ID will be employed.
            namespace: a namespace (a grouping within the vector store) to assign the␣
↪vector to.
            meta: a metadata dictionary associated to the vector.
            kwargs: additional keyword arguments. Currently none is used: if they are␣
↪passed, they will be ignored with a warning.

        Returns:
            the ID of the written vector (str).
        """
        document = {
            k: v
            for k, v in {"$vector": vector, "_id": vector_id, "keyspace": namespace,
↪"meta": meta}.items()
            if v is not None
```

(continues on next page)

(continued from previous page)

```
        }
        if vector_id is not None:
            self.collection.find_one_and_replace({"_id": vector_id}, document,␣
↪upsert=True)
            return vector_id
        insert_result = self.collection.insert_one(document)
        return insert_result.inserted_id

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        """Load a single vector entry from the Astra DB store given its ID.

        Args:
            vector_id: the ID of the required vector.
            namespace: a namespace, within the vector store, to constrain the search.

        Returns:
            The vector entry (a `BaseVectorStoreDriver.Entry`) if found, otherwise␣
↪None.
        """
        find_filter = {k: v for k, v in {"_id": vector_id, "keyspace": namespace}.
↪items() if v is not None}
        match = self.collection.find_one(filter=find_filter, projection={"*": 1})
        if match is not None:
            return BaseVectorStoreDriver.Entry(
                id=match["_id"], vector=match.get("$vector"), meta=match.get("meta"),␣
↪namespace=match.get("keyspace")
            )
        return None

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Load entries from the Astra DB store.

        Args:
            namespace: a namespace, within the vector store, to constrain the search.

        Returns:
            A list of vector (`BaseVectorStoreDriver.Entry`) entries.
        """
        find_filter: dict[str, str] = {} if namespace is None else {"keyspace":␣
↪namespace}
        return [
            BaseVectorStoreDriver.Entry(
                id=match["_id"], vector=match.get("$vector"), meta=match.get("meta"),␣
↪namespace=match.get("keyspace")
            )
            for match in self.collection.find(filter=find_filter, projection={"*": 1})
        ]

    def query_vector(
        self,
```

(continues on next page)

(continued from previous page)

```
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs: Any,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Run a similarity search on the Astra DB store, based on a vector list.

        Args:
            vector: the vector to be queried.
            count: the maximum number of results to return. If omitted, defaults will␣
↪apply.
            namespace: the namespace to filter results by.
            include_vectors: whether to include vector data in the results.
            kwargs: additional keyword arguments. Currently only the free-form dict␣
↪`filter`
                is recognized (and goes straight to the Data API query);
                others will generate a warning and be ignored.

        Returns:
            A list of vector (`BaseVectorStoreDriver.Entry`) entries,
            with their `score` attribute set to the vector similarity to the query.
        """
        query_filter: Optional[dict[str, Any]] = kwargs.get("filter")
        find_filter_ns: dict[str, Any] = {} if namespace is None else {"keyspace":␣
↪namespace}
        find_filter = {**(query_filter or {}), **find_filter_ns}
        find_projection: Optional[dict[str, int]] = {"*": 1} if include_vectors else␣
↪None
        ann_limit = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
        matches = self.collection.find(
            filter=find_filter,
            sort={"$vector": vector},
            limit=ann_limit,
            projection=find_projection,
            include_similarity=True,
        )
        return [
            BaseVectorStoreDriver.Entry(
                id=match["_id"],
                vector=match.get("$vector"),
                score=match["$similarity"],
                meta=match.get("meta"),
                namespace=match.get("keyspace"),
            )
            for match in matches
        ]
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `_collection = field(default=None, kw_only=True, alias='collection',`

`metadata={'serializable': False})` class-attribute instance-attribute

- `api_endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `astra_db_namespace = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `caller_name = field(default='griptape', kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `collection_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `environment = field(kw_only=True, default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `token = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
@lazy_property()
def client(self) -> astrapy.DataAPIClient:
    astrapy = import_optional_dependency("astrapy")

    return astrapy.DataAPIClient(
        callers=[(self.caller_name, None)],
        environment=self.environment or astrapy.utils.unset.UnsetType(),
    )
```

collection()

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
@lazy_property()
def collection(self) -> astrapy.Collection:
    if self.token is None:
        raise ValueError("Astra DB token must be provided.")
    return self.client.get_database(
        self.api_endpoint, token=self.token, keyspace=self.astra_db_namespace
    ).get_collection(self.collection_name)
```

delete_vector(vector_id)

Delete a vector from Astra DB store.

The method succeeds regardless of whether a vector with the provided ID was actually stored or not in the first place.

Parameters

Name	Type	Description	Default
vector_id	*str*	ID of the vector to delete. 	*required*

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> None:
    """Delete a vector from Astra DB store.

    The method succeeds regardless of whether a vector with the provided ID
    was actually stored or not in the first place.

    Args:
        vector_id: ID of the vector to delete.
    """
    self.collection.delete_one({"_id": vector_id})
```

load_entries(*, namespace=None)

Load entries from the Astra DB store.

Parameters

Name	Type	Description	Default
namespace	*Optional[str]*	a namespace, within the vector store, to constrain the search. 	*None*

Returns

Type	Description
<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	A list of vector (*BaseVectorStoreDriver.Entry*) entries.

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Load entries from the Astra DB store.

    Args:
        namespace: a namespace, within the vector store, to constrain the search.

    Returns:
        A list of vector (`BaseVectorStoreDriver.Entry`) entries.
    """
```

(continues on next page)

(continued from previous page)

```
    find_filter: dict[str, str] = {} if namespace is None else {"keyspace": namespace}
    return [
        BaseVectorStoreDriver.Entry(
            id=match["_id"], vector=match.get("$vector"), meta=match.get("meta"),␣
↪namespace=match.get("keyspace")
        )
        for match in self.collection.find(filter=find_filter, projection={"*": 1})
    ]
```

load_entry(vector_id, *, namespace=None)

Load a single vector entry from the Astra DB store given its ID.

Parameters

Name	Type	Description	Default
vector_id	*str*	the ID of the required vector. 	*required*
namespace	*Optional[str]*	a namespace, within the vector store, to constrain the search. 	*None*

Returns

Type	Description
<a href="vector#griptape.drivers.vector.BaseVectorStoreDriver.Entry" title="Entry (griptape.drivers.vector.BaseVectorStoreDriver.Entry)"><code class="language-python">C	The vector entry (a *BaseVectorStoreDriver.Entry*) if found, otherwise None.

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    """Load a single vector entry from the Astra DB store given its ID.

    Args:
        vector_id: the ID of the required vector.
        namespace: a namespace, within the vector store, to constrain the search.

    Returns:
        The vector entry (a `BaseVectorStoreDriver.Entry`) if found, otherwise None.
    """
```

(continues on next page)

(continued from previous page)

```
    find_filter = {k: v for k, v in {"_id": vector_id, "keyspace": namespace}.items()␣
↪if v is not None}
    match = self.collection.find_one(filter=find_filter, projection={"*": 1})
    if match is not None:
        return BaseVectorStoreDriver.Entry(
            id=match["_id"], vector=match.get("$vector"), meta=match.get("meta"),␣
↪namespace=match.get("keyspace")
        )
    return None
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs)

Run a similarity search on the Astra DB store, based on a vector list.

Parameters

Name	Type	Description	Default
vector	<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	the vector to be queried. 	*required*
count	*Optional[int]*	the maximum number of results to return. If omitted, defaults will apply. 	*None*
namespace	*Optional[str]*	the namespace to filter results by. 	*None*
include_vectors	*bool*	whether to include vector data in the results. 	*False*
kwargs	*Any*	additional keyword arguments. Currently only the free-form dict *filter* is recognized (and goes straight to the Data API query); others will generate a warning and be ignored. 	*{}*

Returns

Type	Description
<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	A list of vector (*BaseVectorStoreDriver.Entry*) entries,
<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	with their *score* attribute set to the vector similarity to the query.

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs: Any,
) -> list[BaseVectorStoreDriver.Entry]:
    """Run a similarity search on the Astra DB store, based on a vector list.

    Args:
        vector: the vector to be queried.
        count: the maximum number of results to return. If omitted, defaults will␣
↪apply.
        namespace: the namespace to filter results by.
        include_vectors: whether to include vector data in the results.
        kwargs: additional keyword arguments. Currently only the free-form dict␣
↪`filter`
            is recognized (and goes straight to the Data API query);
            others will generate a warning and be ignored.

    Returns:
        A list of vector (`BaseVectorStoreDriver.Entry`) entries,
        with their `score` attribute set to the vector similarity to the query.
    """
    query_filter: Optional[dict[str, Any]] = kwargs.get("filter")
    find_filter_ns: dict[str, Any] = {} if namespace is None else {"keyspace":␣
↪namespace}
    find_filter = {**(query_filter or {}), **find_filter_ns}
    find_projection: Optional[dict[str, int]] = {"*": 1} if include_vectors else None
    ann_limit = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
    matches = self.collection.find(
        filter=find_filter,
        sort={"$vector": vector},
        limit=ann_limit,
        projection=find_projection,
        include_similarity=True,
    )
    return [
```

(continues on next page)

(continued from previous page)

```
        BaseVectorStoreDriver.Entry(
            id=match["_id"],
            vector=match.get("$vector"),
            score=match["$similarity"],
            meta=match.get("meta"),
            namespace=match.get("keyspace"),
        )
        for match in matches
    ]
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Write a vector to the Astra DB store.

In case the provided ID exists already, an overwrite will take place.

Parameters

Name	Type	Description	Default
vector	<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	the vector to be upserted. 	*required*
vector_id	*Optional[str]*	the ID for the vector to store. If omitted, a server-provided new ID will be employed. 	*None*
namespace	*Optional[str]*	a namespace (a grouping within the vector store) to assign the vector to. 	*None*
meta	*Optional[dict]*	a metadata dictionary associated to the vector. 	*None*
kwargs	*Any*	additional keyword arguments. Currently none is used: if they are passed, they will be ignored with a warning. 	*{}*

Returns

Type	Description
str	the ID of the written vector (str).

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs: Any,
) -> str:
    """Write a vector to the Astra DB store.

    In case the provided ID exists already, an overwrite will take place.

    Args:
        vector: the vector to be upserted.
        vector_id: the ID for the vector to store. If omitted, a server-provided new␣
↪ID will be employed.
        namespace: a namespace (a grouping within the vector store) to assign the␣
↪vector to.
        meta: a metadata dictionary associated to the vector.
        kwargs: additional keyword arguments. Currently none is used: if they are␣
↪passed, they will be ignored with a warning.

    Returns:
        the ID of the written vector (str).
    """
    document = {
        k: v
        for k, v in {"$vector": vector, "_id": vector_id, "keyspace": namespace, "meta
↪": meta}.items()
        if v is not None
    }
    if vector_id is not None:
        self.collection.find_one_and_replace({"_id": vector_id}, document,␣
↪upsert=True)
        return vector_id
    insert_result = self.collection.insert_one(document)
    return insert_result.inserted_id
```

AwsIotCoreEventListenerDriver

Bases: *BaseEventListenerDriver* “)

Source Code in griptape/drivers/event_listener/aws_iot_core_event_listener_driver.py

```
@define
class AwsIotCoreEventListenerDriver(BaseEventListenerDriver):
    iot_endpoint: str = field(kw_only=True)
```

(continues on next page)

(continued from previous page)

```
    topic: str = field(kw_only=True)
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    _client: Optional[IoTDataPlaneClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> IoTDataPlaneClient:
        return self.session.client("iot-data")

    def try_publish_event_payload(self, event_payload: dict) -> None:
        self.client.publish(topic=self.topic, payload=json.dumps(event_payload))

    def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None:
        self.client.publish(topic=self.topic, payload=json.dumps(event_payload_batch))
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `iot_endpoint = field(kw_only=True)` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `topic = field(kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/event_listener/aws_iot_core_event_listener_driver.py

```
@lazy_property()
def client(self) -> IoTDataPlaneClient:
    return self.session.client("iot-data")
```

try_publish_event_payload(event_payload)

Source Code in griptape/drivers/event_listener/aws_iot_core_event_listener_driver.py

```
def try_publish_event_payload(self, event_payload: dict) -> None:
    self.client.publish(topic=self.topic, payload=json.dumps(event_payload))
```

try_publish_event_payload_batch(event_payload_batch)

Source Code in griptape/drivers/event_listener/aws_iot_core_event_listener_driver.py

```
def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None:
    self.client.publish(topic=self.topic, payload=json.dumps(event_payload_batch))
```

AzureMongoDbVectorStoreDriver

Bases: *MongoDbAtlasVectorStoreDriver* “)

Source Code in griptape/drivers/vector/azure_mongodb_vector_store_driver.py

```
@define
class AzureMongoDbVectorStoreDriver(MongoDbAtlasVectorStoreDriver):
    """A Vector Store Driver for CosmosDB with MongoDB vCore API."""

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        offset: Optional[int] = None,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Queries the MongoDB collection for documents that match the provided␣
↪vector list.

        Results can be customized based on parameters like count, namespace,␣
↪inclusion of vectors, offset, and index.
        """
        collection = self.get_collection()

        count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
        offset = offset or 0

        pipeline = []

        pipeline.append(
            {
                "$search": {
                    "cosmosSearch": {
                        "vector": vector,
                        "path": self.vector_path,
                        "k": min(count * self.num_candidates_multiplier, self.MAX_NUM_
↪CANDIDATES),
                    },
                    "returnStoredSource": True,
                },
            },
        )

        if namespace:
            pipeline.append({"$match": {"namespace": namespace}})

        pipeline.append({"$project": {"similarityScore": {"$meta": "searchScore"},
↪"document": "$$ROOT"}})

        return [
```

(continues on next page)

(continued from previous page)

```
            BaseVectorStoreDriver.Entry(
                id=str(doc["_id"]),
                vector=doc[self.vector_path] if include_vectors else [],
                score=doc["similarityScore"],
                meta=doc["document"]["meta"],
                namespace=namespace,
            )
            for doc in collection.aggregate(pipeline)
        ]
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, offset=None, **kwargs)

Source Code in griptape/drivers/vector/azure_mongodb_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    offset: Optional[int] = None,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    """Queries the MongoDB collection for documents that match the provided vector␣
↪list.

    Results can be customized based on parameters like count, namespace, inclusion of␣
↪vectors, offset, and index.
    """
    collection = self.get_collection()

    count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
    offset = offset or 0

    pipeline = []

    pipeline.append(
        {
            "$search": {
                "cosmosSearch": {
                    "vector": vector,
                    "path": self.vector_path,
                    "k": min(count * self.num_candidates_multiplier, self.MAX_NUM_
↪CANDIDATES),
                },
                "returnStoredSource": True,
            },
        }
    )
```

(continues on next page)

(continued from previous page)

```
    if namespace:
        pipeline.append({"$match": {"namespace": namespace}})

    pipeline.append({"$project": {"similarityScore": {"$meta": "searchScore"},
↪"document": "$$ROOT"}})

    return [
        BaseVectorStoreDriver.Entry(
            id=str(doc["_id"]),
            vector=doc[self.vector_path] if include_vectors else [],
            score=doc["similarityScore"],
            meta=doc["document"]["meta"],
            namespace=namespace,
        )
        for doc in collection.aggregate(pipeline)
    ]
```

AzureOpenAiChatPromptDriver

Bases: *AzureOpenAiChatPromptDriver* “)

Attributes

Name	Type	Description
<a href="./#grip-tape.drivers.Azure-OpenAiChatPrompt-Driver.azure_de-ployment" ti-tle="azure_deployment = field(kw_only=True, default=Fac-tory(lambda self: self.model, takes_self=True), meta-data={'serializable': True}) class-attribute instance-attribute (grip-tape.drivers.Azure-OpenAiChatPrompt-Driver.azure_de-ployment)"><code class="language-python">	*str*	An optional Azure OpenAi de-ployment id. Defaults to the model name.

continues on next page

Table 53 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.AzureOpenAiChatPromptDriver.azure_endpoint" title="azure_endpoint = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.AzureOpenAiChatPromptDriver.azure_endpoint)"><code class="language-python">	*str*	An Azure OpenAi endpoint.
<a href="./#griptape.drivers.AzureOpenAiChatPromptDriver.azure_ad_token" title="azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.AzureOpenAiChatPromptDriver.azure_ad_token)"><code class="language-python">	*Optional[str]*	An optional Azure Active Directory token.
<a href="./#griptape.drivers.AzureOpenAiChatPromptDriver.azure_ad_token_provider" title="azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.AzureOpenAiChatPromptDriver.azure_ad_token_provider)"><code class="language-python">	*Optional[Callable[[], str]]*	An optional Azure Active Directory token provider.

continues on next page

Table 53 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.AzureOpenAiChatPromptDriver.api_version" title="api_version = field(default='2024-10-21', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.AzureOpenAiChatPromptDriver.api_version)"><code class="language-python">	*str*	An Azure OpenAi API version.
<a href="./#griptape.drivers.AzureOpenAiChatPromptDriver.client" title="client() (griptape.drivers.AzureOpenAiChatPromptDriver.client)"><code class="language-python">	*AzureOpenAI*	An *openai.AzureOpenAI* client.

Source Code in griptape/drivers/prompt/azure_openai_chat_prompt_driver.py

```
@define
class AzureOpenAiChatPromptDriver(OpenAiChatPromptDriver):
    """Azure OpenAi Chat Prompt Driver.

    Attributes:
        azure_deployment: An optional Azure OpenAi deployment id. Defaults to the␣
↪model name.
        azure_endpoint: An Azure OpenAi endpoint.
        azure_ad_token: An optional Azure Active Directory token.
        azure_ad_token_provider: An optional Azure Active Directory token provider.
        api_version: An Azure OpenAi API version.
        client: An `openai.AzureOpenAI` client.
    """

    azure_deployment: str = field(
        kw_only=True,
        default=Factory(lambda self: self.model, takes_self=True),
        metadata={"serializable": True},
    )
    azure_endpoint: str = field(kw_only=True, metadata={"serializable": True})
    azure_ad_token: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": False})
```

(continues on next page)

(continued from previous page)

```
    azure_ad_token_provider: Optional[Callable[[], str]] = field(
        kw_only=True,
        default=None,
        metadata={"serializable": False},
    )
    api_version: str = field(default="2024-10-21", kw_only=True, metadata={
↪"serializable": True})
    _client: Optional[openai.AzureOpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.AzureOpenAI:
        return openai.AzureOpenAI(
            organization=self.organization,
            api_key=self.api_key,
            api_version=self.api_version,
            azure_endpoint=self.azure_endpoint,
            azure_deployment=self.azure_deployment,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        params = super()._base_params(prompt_stack)
        if self.api_version < "2024-02-01" and "seed" in params:
            del params["seed"]
        if self.api_version < "2024-10-21":
            if "stream_options" in params:
                del params["stream_options"]
            if "parallel_tool_calls" in params:
                del params["parallel_tool_calls"]

        # TODO: Add once Azure supports modalities
        if "modalities" in params:
            del params["modalities"]
        return params
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_version = field(default='2024-10-21', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True})` class-attribute instance-attribute
- `azure_endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/azure_openai_chat_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    params = super()._base_params(prompt_stack)
    if self.api_version < "2024-02-01" and "seed" in params:
        del params["seed"]
    if self.api_version < "2024-10-21":
        if "stream_options" in params:
            del params["stream_options"]
        if "parallel_tool_calls" in params:
            del params["parallel_tool_calls"]

    # TODO: Add once Azure supports modalities
    if "modalities" in params:
        del params["modalities"]
    return params
```

client()

Source Code in griptape/drivers/prompt/azure_openai_chat_prompt_driver.py

```
@lazy_property()
def client(self) -> openai.AzureOpenAI:
    return openai.AzureOpenAI(
        organization=self.organization,
        api_key=self.api_key,
        api_version=self.api_version,
        azure_endpoint=self.azure_endpoint,
        azure_deployment=self.azure_deployment,
        azure_ad_token=self.azure_ad_token,
        azure_ad_token_provider=self.azure_ad_token_provider,
    )
```

AzureOpenAiEmbeddingDriver

Bases: *OpenAiEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.AzureOpenAiEmbeddingDriver.azure_deployment" title="azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.AzureOpenAiEmbeddingDriver.azure_deployment)"><code class="language-python">	*str*	An optional Azure OpenAi deployment id. Defaults to the model name.
<a href="./#griptape.drivers.AzureOpenAiEmbeddingDriver.azure_endpoint" title="azure_endpoint = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.AzureOpenAiEmbeddingDriver.azure_endpoint)"><code class="language-python">	*str*	An Azure OpenAi endpoint.
<a href="./#griptape.drivers.AzureOpenAiEmbeddingDriver.azure_ad_token" title="azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.AzureOpenAiEmbeddingDriver.azure_ad_token)"><code class="language-python">	*Optional[str]*	An optional Azure Active Directory token.

continues on next page

Table 54 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.AzureOpenAiEmbeddingDriver.azure_ad_token_provider" title="azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.AzureOpenAiEmbeddingDriver.azure_ad_token_provider)"><code class="language-python">	*Optional[Callable[[], str]]*	An optional Azure Active Directory token provider.
<a href="./#griptape.drivers.AzureOpenAiEmbeddingDriver.api_version" title="api_version = field(default='2024-10-21', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.AzureOpenAiEmbeddingDriver.api_version)"><code class="language-python">	*str*	An Azure OpenAi API version.

continues on next page

Table 54 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.AzureOpenAiEmbeddingDriver.tokenizer" title="tokenizer = field(default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.drivers.AzureOpenAiEmbeddingDriver.tokenizer)"><code class="language-python">	<a href="../tokenizers#griptape.tokenizers.OpenAiTokenizer" title="OpenAiTokenizer (griptape.tokenizers.OpenAiTokenizer)"><code class="language-python">OpenAiTokenizer</code></a	An *OpenAiTokenizer*.
<a href="./#griptape.drivers.AzureOpenAiEmbeddingDriver.client" title="client() (griptape.drivers.AzureOpenAiEmbeddingDriver.client)"><code class="language-python">	*AzureOpenAI*	An *openai.AzureOpenAI* client.

Source Code in griptape/drivers/embedding/azure_openai_embedding_driver.py

```
@define
class AzureOpenAiEmbeddingDriver(OpenAiEmbeddingDriver):
    """Azure OpenAi Embedding Driver.

    Attributes:
        azure_deployment: An optional Azure OpenAi deployment id. Defaults to the␣
↪model name.
        azure_endpoint: An Azure OpenAi endpoint.
        azure_ad_token: An optional Azure Active Directory token.
        azure_ad_token_provider: An optional Azure Active Directory token provider.
        api_version: An Azure OpenAi API version.
        tokenizer: An `OpenAiTokenizer`.
        client: An `openai.AzureOpenAI` client.
    """

    azure_deployment: str = field(
        kw_only=True,
        default=Factory(lambda self: self.model, takes_self=True),
        metadata={"serializable": True},
    )
```

(continues on next page)

(continued from previous page)

```
    azure_endpoint: str = field(kw_only=True, metadata={"serializable": True})
    azure_ad_token: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": False})
    azure_ad_token_provider: Optional[Callable[[], str]] = field(
        kw_only=True,
        default=None,
        metadata={"serializable": False},
    )
    api_version: str = field(default="2024-10-21", kw_only=True, metadata={
↪"serializable": True})
    tokenizer: OpenAiTokenizer = field(
        default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
    _client: Optional[openai.AzureOpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.AzureOpenAI:
        return openai.AzureOpenAI(
            organization=self.organization,
            api_key=self.api_key,
            api_version=self.api_version,
            azure_endpoint=self.azure_endpoint,
            azure_deployment=self.azure_deployment,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_version = field(default='2024-10-21', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True})` class-attribute instance-attribute
- `azure_endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/azure_openai_embedding_driver.py

```
@lazy_property()
def client(self) -> openai.AzureOpenAI:
    return openai.AzureOpenAI(
        organization=self.organization,
        api_key=self.api_key,
        api_version=self.api_version,
        azure_endpoint=self.azure_endpoint,
        azure_deployment=self.azure_deployment,
        azure_ad_token=self.azure_ad_token,
        azure_ad_token_provider=self.azure_ad_token_provider,
    )
```

AzureOpenAiImageGenerationDriver

Bases: *OpenAiImageGenerationDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.AzureOpenAiImageGenerationDriver.azure_deployment" title="azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.AzureOpenAiImageGenerationDriver.azure_deployment)"><code class="language-python">	*str*	An optional Azure OpenAi deployment id. Defaults to the model name.

continues on next page

Table 55 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.AzureOpenAiImageGenerationDriver.azure_endpoint" title="azure_endpoint = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.AzureOpenAiImageGenerationDriver.azure_endpoint)"><code class="language-python">	*str*	An Azure OpenAi endpoint.
<a href="./#griptape.drivers.AzureOpenAiImageGenerationDriver.azure_ad_token" title="azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.AzureOpenAiImageGenerationDriver.azure_ad_token)"><code class="language-python">	*Optional[str]*	An optional Azure Active Directory token.
<a href="./#griptape.drivers.AzureOpenAiImageGenerationDriver.azure_ad_token_provider" title="azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.AzureOpenAiImageGenerationDriver.azure_ad_token_provider)"><code class="language-python">	*Optional[Callable[[], str]]*	An optional Azure Active Directory token provider.

continues on next page

Table 55 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.AzureOpenAiImageGenerationDriver.api_version" title="api_version = field(default='2024-02-01', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.AzureOpenAiImageGenerationDriver.api_version)"><code class="language-python">	*str*	An Azure OpenAi API version.
<a href="./#griptape.drivers.AzureOpenAiImageGenerationDriver.client" title="client() (griptape.drivers.AzureOpenAiImageGenerationDriver.client)"><code class="language-python">	*AzureOpenAI*	An *openai.AzureOpenAI* client.

Source Code in griptape/drivers/image_generation/azure_openai_image_generation_driver.py

```
@define
class AzureOpenAiImageGenerationDriver(OpenAiImageGenerationDriver):
    """Driver for Azure-hosted OpenAI image generation API.

    Attributes:
        azure_deployment: An optional Azure OpenAi deployment id. Defaults to the␣
↪model name.
        azure_endpoint: An Azure OpenAi endpoint.
        azure_ad_token: An optional Azure Active Directory token.
        azure_ad_token_provider: An optional Azure Active Directory token provider.
        api_version: An Azure OpenAi API version.
        client: An `openai.AzureOpenAI` client.
    """

    azure_deployment: str = field(
        kw_only=True,
        default=Factory(lambda self: self.model, takes_self=True),
        metadata={"serializable": True},
    )
    azure_endpoint: str = field(kw_only=True, metadata={"serializable": True})
    azure_ad_token: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": False})
```

(continues on next page)

(continued from previous page)

```
    azure_ad_token_provider: Optional[Callable[[], str]] = field(
        kw_only=True,
        default=None,
        metadata={"serializable": False},
    )
    api_version: str = field(default="2024-02-01", kw_only=True, metadata={
↪"serializable": True})
    _client: Optional[openai.AzureOpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.AzureOpenAI:
        return openai.AzureOpenAI(
            organization=self.organization,
            api_key=self.api_key,
            api_version=self.api_version,
            azure_endpoint=self.azure_endpoint,
            azure_deployment=self.azure_deployment,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_version = field(default='2024-02-01', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True})` class-attribute instance-attribute
- `azure_endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/image_generation/azure_openai_image_generation_driver.py

```
@lazy_property()
def client(self) -> openai.AzureOpenAI:
    return openai.AzureOpenAI(
        organization=self.organization,
        api_key=self.api_key,
        api_version=self.api_version,
        azure_endpoint=self.azure_endpoint,
        azure_deployment=self.azure_deployment,
        azure_ad_token=self.azure_ad_token,
```

(continues on next page)

(continued from previous page)

```
        azure_ad_token_provider=self.azure_ad_token_provider,
    )
```

AzureOpenAiTextToSpeechDriver

Bases: *AzureOpenAiTextToSpeechDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.AzureOpenAiTextToSpeechDriver.azure_deployment" title="azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.AzureOpenAiTextToSpeechDriver.azure_deployment)"><code class="language-python">	*str*	An optional Azure OpenAi deployment id. Defaults to the model name.
<a href="./#griptape.drivers.AzureOpenAiTextToSpeechDriver.azure_endpoint" title="azure_endpoint = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.AzureOpenAiTextToSpeechDriver.azure_endpoint)"><code class="language-python">	*str*	An Azure OpenAi endpoint.

continues on next page

Table 56 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.AzureOpenAiTextToSpeechDriver.azure_ad_token" title="azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.AzureOpenAiTextToSpeechDriver.azure_ad_token)"><code class="language-python">	*Optional[str]*	An optional Azure Active Directory token.
<a href="./#griptape.drivers.AzureOpenAiTextToSpeechDriver.azure_ad_token_provider" title="azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.AzureOpenAiTextToSpeechDriver.azure_ad_token_provider)"><code class="language-python">	*Optional[Callable[[], str]]*	An optional Azure Active Directory token provider.
<a href="./#griptape.drivers.AzureOpenAiTextToSpeechDriver.api_version" title="api_version = field(default='2024-07-01-previe kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.AzureOpenAiTextToSpeechDriver.api_version)"><code class="language-python">	*str*	An Azure OpenAi API version.

continues on next page

Table 56 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.AzureOpenAiTextToSpeechDriver.client" title="client() (griptape.drivers.AzureOpenAiTextToSpeechDriver.client)"><code class="language-python">	*AzureOpenAI*	An *openai.AzureOpenAI* client.

Source Code in griptape/drivers/text_to_speech/azure_openai_text_to_speech_driver.py

```
@define
class AzureOpenAiTextToSpeechDriver(OpenAiTextToSpeechDriver):
    """Azure OpenAi Text to Speech Driver.

    Attributes:
        azure_deployment: An optional Azure OpenAi deployment id. Defaults to the␣
↪model name.
        azure_endpoint: An Azure OpenAi endpoint.
        azure_ad_token: An optional Azure Active Directory token.
        azure_ad_token_provider: An optional Azure Active Directory token provider.
        api_version: An Azure OpenAi API version.
        client: An `openai.AzureOpenAI` client.
    """

    model: str = field(default="tts", kw_only=True, metadata={"serializable": True})
    azure_deployment: str = field(
        kw_only=True,
        default=Factory(lambda self: self.model, takes_self=True),
        metadata={"serializable": True},
    )
    azure_endpoint: str = field(kw_only=True, metadata={"serializable": True})
    azure_ad_token: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": False})
    azure_ad_token_provider: Optional[Callable[[], str]] = field(
        kw_only=True,
        default=None,
        metadata={"serializable": False},
    )
    api_version: str = field(default="2024-07-01-preview", kw_only=True, metadata={
↪"serializable": True})
    _client: Optional[openai.AzureOpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.AzureOpenAI:
        return openai.AzureOpenAI(
            organization=self.organization,
```

(continues on next page)

(continued from previous page)

```
            api_key=self.api_key,
            api_version=self.api_version,
            azure_endpoint=self.azure_endpoint,
            azure_deployment=self.azure_deployment,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_version = field(default='2024-07-01-preview', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True})` class-attribute instance-attribute
- `azure_endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(default='tts', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/text_to_speech/azure_openai_text_to_speech_driver.py

```
@lazy_property()
def client(self) -> openai.AzureOpenAI:
    return openai.AzureOpenAI(
        organization=self.organization,
        api_key=self.api_key,
        api_version=self.api_version,
        azure_endpoint=self.azure_endpoint,
        azure_deployment=self.azure_deployment,
        azure_ad_token=self.azure_ad_token,
        azure_ad_token_provider=self.azure_ad_token_provider,
    )
```

BaseAssistantDriver

Bases:

ABC

Source Code in griptape/drivers/assistant/base_assistant_driver.py

```
@define
class BaseAssistantDriver(ABC):
    """Base class for AssistantDrivers."""
```

(continues on next page)

(continued from previous page)

```
    def run(self, *args: BaseArtifact) -> TextArtifact:
        return self.try_run(*args)

    @abstractmethod
    def try_run(self, *args: BaseArtifact) -> TextArtifact: ...
```

run(*args)

Source Code in griptape/drivers/assistant/base_assistant_driver.py

```
def run(self, *args: BaseArtifact) -> TextArtifact:
    return self.try_run(*args)
```

try_run(*args)abstractmethod

Source Code in griptape/drivers/assistant/base_assistant_driver.py

```
@abstractmethod
def try_run(self, *args: BaseArtifact) -> TextArtifact: ...
```

BaseAudioTranscriptionDriver

Bases: *SerializableMixin* “) , *ExponentialBackoffMixin* ”) , ABC

Source Code in griptape/drivers/audio_transcription/base_audio_transcription_driver.py

```
@define
class BaseAudioTranscriptionDriver(SerializableMixin, ExponentialBackoffMixin, ABC):
    model: str = field(kw_only=True, metadata={"serializable": True})

    def before_run(self) -> None:
        EventBus.publish_event(StartAudioTranscriptionEvent())

    def after_run(self) -> None:
        EventBus.publish_event(FinishAudioTranscriptionEvent())

    def run(self, audio: AudioArtifact, prompts: Optional[list[str]] = None) ->␣
↪TextArtifact:
        for attempt in self.retrying():
            with attempt:
                self.before_run()
                result = self.try_run(audio, prompts)
                self.after_run()

                return result

        raise Exception("Failed to run audio transcription")

    @abstractmethod
    def try_run(self, audio: AudioArtifact, prompts: Optional[list[str]] = None) ->␣
↪TextArtifact: ...
```

- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

after_run()

Source Code in griptape/drivers/audio_transcription/base_audio_transcription_driver.py

```
def after_run(self) -> None:
    EventBus.publish_event(FinishAudioTranscriptionEvent())
```

before_run()

Source Code in griptape/drivers/audio_transcription/base_audio_transcription_driver.py

```
def before_run(self) -> None:
    EventBus.publish_event(StartAudioTranscriptionEvent())
```

run(audio, prompts=None)

Source Code in griptape/drivers/audio_transcription/base_audio_transcription_driver.py

```
def run(self, audio: AudioArtifact, prompts: Optional[list[str]] = None) ->␣
↪TextArtifact:
    for attempt in self.retrying():
        with attempt:
            self.before_run()
            result = self.try_run(audio, prompts)
            self.after_run()

            return result

    raise Exception("Failed to run audio transcription")
```

try_run(audio, prompts=None)abstractmethod

Source Code in griptape/drivers/audio_transcription/base_audio_transcription_driver.py

```
@abstractmethod
def try_run(self, audio: AudioArtifact, prompts: Optional[list[str]] = None) ->␣
↪TextArtifact: ...
```

BaseConversationMemoryDriver

Bases: *SerializableMixin* “) , ABC

Source Code in griptape/drivers/memory/conversation/base_conversation_memory_driver.py

```
class BaseConversationMemoryDriver(SerializableMixin, ABC):
    @abstractmethod
    def store(self, runs: list[Run], metadata: dict[str, Any]) -> None: ...

    @abstractmethod
    def load(self) -> tuple[list[Run], dict[str, Any]]: ...
```

(continues on next page)

(continued from previous page)

```
    def _to_params_dict(self, runs: list[Run], metadata: dict[str, Any]) -> dict:
        return {"runs": [run.to_dict() for run in runs], "metadata": metadata}

    def _from_params_dict(self, params_dict: dict[str, Any]) -> tuple[list[Run],␣
↪dict[str, Any]]:
        from griptape.memory.structure import Run

        return [Run.from_dict(run) for run in params_dict.get("runs", [])], params_
↪dict.get("metadata", {})
```

_from_params_dict(params_dict)

Source Code in griptape/drivers/memory/conversation/base_conversation_memory_driver.py

```
def _from_params_dict(self, params_dict: dict[str, Any]) -> tuple[list[Run], dict[str,
↪ Any]]:
    from griptape.memory.structure import Run

    return [Run.from_dict(run) for run in params_dict.get("runs", [])], params_dict.
↪get("metadata", {})
```

_to_params_dict(runs, metadata)

Source Code in griptape/drivers/memory/conversation/base_conversation_memory_driver.py

```
def _to_params_dict(self, runs: list[Run], metadata: dict[str, Any]) -> dict:
    return {"runs": [run.to_dict() for run in runs], "metadata": metadata}
```

load()abstractmethod

Source Code in griptape/drivers/memory/conversation/base_conversation_memory_driver.py

```
@abstractmethod
def load(self) -> tuple[list[Run], dict[str, Any]]: ...
```

store(runs, metadata)abstractmethod

Source Code in griptape/drivers/memory/conversation/base_conversation_memory_driver.py

```
@abstractmethod
def store(self, runs: list[Run], metadata: dict[str, Any]) -> None: ...
```

BaseDiffusionImageGenerationPipelineDriver

Bases:

ABC

Source Code in griptape/drivers/image_generation_pipeline/base_image_generation_pipeline_driver.py

```
@define
class BaseDiffusionImageGenerationPipelineDriver(ABC):
```

(continues on next page)

(continued from previous page)

```
    @abstractmethod
    def prepare_pipeline(self, model: str, device: Optional[str]) -> Any: ...

    @abstractmethod
    def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:␣
↪...

    @abstractmethod
    def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict: ...

    @property
    @abstractmethod
    def output_image_dimensions(self) -> tuple[int, int]: ...
```

- `output_image_dimensions` abstractmethod property

make_additional_params(negative_prompts, device)abstractmethod

Source Code in griptape/drivers/image_generation_pipeline/base_image_generation_pipeline_driver.py

```
@abstractmethod
def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict: ...
```

make_image_param(image)abstractmethod

Source Code in griptape/drivers/image_generation_pipeline/base_image_generation_pipeline_driver.py

```
@abstractmethod
def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]: ...
```

prepare_pipeline(model, device)abstractmethod

Source Code in griptape/drivers/image_generation_pipeline/base_image_generation_pipeline_driver.py

```
@abstractmethod
def prepare_pipeline(self, model: str, device: Optional[str]) -> Any: ...
```

BaseEmbeddingDriver

Bases: *SerializableMixin* “) , *ExponentialBackoffMixin* ”) , ABC

Attributes

Name	Type	Description
<a href="./#griptape.drivers.BaseEmbeddingDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.BaseEmbedding-Driver.model)"><code class="language-python">	*str*	The name of the model to use.
<a href="./#griptape.drivers.BaseEmbeddingDriver.tokenizer" title="tokenizer = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.drivers.BaseEmbeddingDriver.tokenizer)"><code class="language-python">	<a href="../tokenizers#griptape.tokenizers.BaseTokenizer" title="BaseTokenizer (griptape.tokenizers.BaseTokenizer)"><code class="language-python">Optional[BaseTokenizer]</c	An instance of *BaseTokenizer* to use when calculating tokens.

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
@define
class BaseEmbeddingDriver(SerializableMixin, ExponentialBackoffMixin, ABC):
    """Base Embedding Driver.

    Attributes:
        model: The name of the model to use.
        tokenizer: An instance of `BaseTokenizer` to use when calculating tokens.
    """

    model: str = field(kw_only=True, metadata={"serializable": True})
    tokenizer: Optional[BaseTokenizer] = field(default=None, kw_only=True)
    chunker: Optional[BaseChunker] = field(init=False)

    def __attrs_post_init__(self) -> None:
        self.chunker = TextChunker(tokenizer=self.tokenizer) if self.tokenizer else␣
↪None

    def embed_text_artifact(
        self, artifact: TextArtifact, *, vector_operation: VectorOperation | None =␣
↪None
    ) -> list[float]:
        warnings.warn(
            "`BaseEmbeddingDriver.embed_text_artifact` is deprecated and will be␣
```

(continues on next page)

(continued from previous page)

```
↪removed in a future release. `BaseEmbeddingDriver.embed` is a drop-in replacement.",
            DeprecationWarning,
            stacklevel=2,
        )
        return self.embed(artifact, vector_operation=vector_operation)

    def embed_string(self, string: str, *, vector_operation: VectorOperation | None =␣
↪None) -> list[float]:
        warnings.warn(
            "`BaseEmbeddingDriver.embed_string` is deprecated and will be removed in␣
↪a future release. `BaseEmbeddingDriver.embed` is a drop-in replacement.",
            DeprecationWarning,
            stacklevel=2,
        )
        return self.embed(string, vector_operation=vector_operation)

    def embed(
        self, value: str | TextArtifact | ImageArtifact, *, vector_operation:␣
↪VectorOperation | None = None
    ) -> list[float]:
        for attempt in self.retrying():
            with attempt:
                if isinstance(value, str):
                    if (
                        self.tokenizer is not None
                        and self.tokenizer.count_tokens(value) > self.tokenizer.max_
↪input_tokens
                    ):
                        return self._embed_long_string(value, vector_operation=vector_
↪operation)
                    return self.try_embed_chunk(value, vector_operation=vector_
↪operation)
                if isinstance(value, TextArtifact):
                    return self.embed(value.to_text(), vector_operation=vector_
↪operation)
                if isinstance(value, ImageArtifact):
                    return self.try_embed_artifact(value, vector_operation=vector_
↪operation)
        raise RuntimeError("Failed to embed string.")

    def try_embed_artifact(
        self, artifact: TextArtifact | ImageArtifact, *, vector_operation:␣
↪VectorOperation | None = None
    ) -> list[float]:
        # TODO: Mark as abstract method for griptape 2.0
        if isinstance(artifact, TextArtifact):
            return self.try_embed_chunk(artifact.value, vector_operation=vector_
↪operation)
        raise ValueError(f"{self.__class__.__name__} does not support embedding␣
↪images.")

    @abstractmethod
```

(continues on next page)

(continued from previous page)

```
    def try_embed_chunk(self, chunk: str, *, vector_operation: VectorOperation | None␣
↪= None) -> list[float]:
        # TODO: Remove for griptape 2.0, subclasses should implement `try_embed_
↪artifact` instead
        ...

    def _embed_long_string(self, string: str, *, vector_operation: VectorOperation |␣
↪None = None) -> list[float]:
        """Embeds a string that is too long to embed in one go.

        Adapted from: https://github.com/openai/openai-cookbook/blob/
↪683e5f5a71bc7a1b0e5b7a35e087f53cc55fceea/examples/Embedding_long_inputs.ipynb
        """
        chunks = self.chunker.chunk(string)  # pyright:␣
↪ignore[reportOptionalMemberAccess] In practice this is never None

        embedding_chunks = []
        length_chunks = []
        for chunk in chunks:
            embedding_chunks.append(self.embed(chunk.value, vector_operation=vector_
↪operation))
            length_chunks.append(len(chunk))

        # generate weighted averages
        embedding_chunks = np.average(embedding_chunks, axis=0, weights=length_chunks)

        # normalize length to 1
        embedding_chunks = embedding_chunks / np.linalg.norm(embedding_chunks)

        return embedding_chunks.tolist()
```

- `chunker = field(init=False)` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=None, kw_only=True)` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
def __attrs_post_init__(self) -> None:
    self.chunker = TextChunker(tokenizer=self.tokenizer) if self.tokenizer else None
```

_embed_long_string(string, *, vector_operation=None)

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
def _embed_long_string(self, string: str, *, vector_operation: VectorOperation | None␣
↪= None) -> list[float]:
    """Embeds a string that is too long to embed in one go.
```

(continues on next page)

(continued from previous page)

```
    Adapted from: https://github.com/openai/openai-cookbook/blob/
↪683e5f5a71bc7a1b0e5b7a35e087f53cc55fceea/examples/Embedding_long_inputs.ipynb
    """
    chunks = self.chunker.chunk(string)  # pyright:␣
↪ignore[reportOptionalMemberAccess] In practice this is never None

    embedding_chunks = []
    length_chunks = []
    for chunk in chunks:
        embedding_chunks.append(self.embed(chunk.value, vector_operation=vector_
↪operation))
        length_chunks.append(len(chunk))

    # generate weighted averages
    embedding_chunks = np.average(embedding_chunks, axis=0, weights=length_chunks)

    # normalize length to 1
    embedding_chunks = embedding_chunks / np.linalg.norm(embedding_chunks)

    return embedding_chunks.tolist()
```

embed(value, *, vector_operation=None)

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
def embed(
    self, value: str | TextArtifact | ImageArtifact, *, vector_operation:␣
↪VectorOperation | None = None
) -> list[float]:
    for attempt in self.retrying():
        with attempt:
            if isinstance(value, str):
                if (
                    self.tokenizer is not None
                    and self.tokenizer.count_tokens(value) > self.tokenizer.max_input_
↪tokens
                ):
                    return self._embed_long_string(value, vector_operation=vector_
↪operation)
                return self.try_embed_chunk(value, vector_operation=vector_operation)
            if isinstance(value, TextArtifact):
                return self.embed(value.to_text(), vector_operation=vector_operation)
            if isinstance(value, ImageArtifact):
                return self.try_embed_artifact(value, vector_operation=vector_
↪operation)
    raise RuntimeError("Failed to embed string.")
```

embed_string(string, *, vector_operation=None)

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
def embed_string(self, string: str, *, vector_operation: VectorOperation | None =␣
↪None) -> list[float]:
    warnings.warn(
        "`BaseEmbeddingDriver.embed_string` is deprecated and will be removed in a␣
↪future release. `BaseEmbeddingDriver.embed` is a drop-in replacement.",
        DeprecationWarning,
        stacklevel=2,
    )
    return self.embed(string, vector_operation=vector_operation)
```

embed_text_artifact(artifact, *, vector_operation=None)

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
def embed_text_artifact(
    self, artifact: TextArtifact, *, vector_operation: VectorOperation | None = None
) -> list[float]:
    warnings.warn(
        "`BaseEmbeddingDriver.embed_text_artifact` is deprecated and will be removed␣
↪in a future release. `BaseEmbeddingDriver.embed` is a drop-in replacement.",
        DeprecationWarning,
        stacklevel=2,
    )
    return self.embed(artifact, vector_operation=vector_operation)
```

try_embed_artifact(artifact, *, vector_operation=None)

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
def try_embed_artifact(
    self, artifact: TextArtifact | ImageArtifact, *, vector_operation:␣
↪VectorOperation | None = None
) -> list[float]:
    # TODO: Mark as abstract method for griptape 2.0
    if isinstance(artifact, TextArtifact):
        return self.try_embed_chunk(artifact.value, vector_operation=vector_operation)
    raise ValueError(f"{self.__class__.__name__} does not support embedding images.")
```

try_embed_chunk(chunk, *, vector_operation=None)abstractmethod

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
@abstractmethod
def try_embed_chunk(self, chunk: str, *, vector_operation: VectorOperation | None =␣
↪None) -> list[float]:
    # TODO: Remove for griptape 2.0, subclasses should implement `try_embed_artifact`␣
↪instead
    ...
```

BaseEventListenerDriver

Bases: *FuturesExecutorMixin* “) , *ExponentialBackoffMixin* ”) , ABC

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
@define
class BaseEventListenerDriver(FuturesExecutorMixin, ExponentialBackoffMixin, ABC):
    batched: bool = field(default=True, kw_only=True)
    batch_size: int = field(default=10, kw_only=True)

    _batch: list[dict] = field(default=Factory(list), kw_only=True)

    @property
    def batch(self) -> list[dict]:
        return self._batch

    def publish_event(self, event: BaseEvent | dict) -> None:
        event_payload = event if isinstance(event, dict) else event.to_dict()

        with self.create_futures_executor() as futures_executor:
            if self.batched:
                self._batch.append(event_payload)
                if len(self.batch) >= self.batch_size:
                    futures_executor.submit(with_contextvars(self._safe_publish_event_
↪payload_batch), self.batch)
                    self._batch = []
            else:
                futures_executor.submit(with_contextvars(self._safe_publish_event_
↪payload), event_payload)

    def flush_events(self) -> None:
        if self.batch:
            with self.create_futures_executor() as futures_executor:
                futures_executor.submit(with_contextvars(self._safe_publish_event_
↪payload_batch), self.batch)
            self._batch = []

    @abstractmethod
    def try_publish_event_payload(self, event_payload: dict) -> None: ...

    @abstractmethod
    def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None: ...

    def _safe_publish_event_payload(self, event_payload: dict) -> None:
        try:
            for attempt in self.retrying():
                with attempt:
                    self.try_publish_event_payload(event_payload)
        except Exception:
            logger.warning("Failed to publish event after %s attempts", self.max_
↪attempts, exc_info=True)
```

(continues on next page)

(continued from previous page)

```
    def _safe_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None:
        try:
            for attempt in self.retrying():
                with attempt:
                    self.try_publish_event_payload_batch(event_payload_batch)
        except Exception:
            logger.warning("Failed to publish event batch after %s attempts", self.
↪max_attempts, exc_info=True)
```

- `_batch = field(default=Factory(list), kw_only=True)` class-attribute instance-attribute
- `batch` property
- `batch_size = field(default=10, kw_only=True)` class-attribute instance-attribute
- `batched = field(default=True, kw_only=True)` class-attribute instance-attribute

_safe_publish_event_payload(event_payload)

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
def _safe_publish_event_payload(self, event_payload: dict) -> None:
    try:
        for attempt in self.retrying():
            with attempt:
                self.try_publish_event_payload(event_payload)
    except Exception:
        logger.warning("Failed to publish event after %s attempts", self.max_attempts,
↪ exc_info=True)
```

_safe_publish_event_payload_batch(event_payload_batch)

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
def _safe_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None:
    try:
        for attempt in self.retrying():
            with attempt:
                self.try_publish_event_payload_batch(event_payload_batch)
    except Exception:
        logger.warning("Failed to publish event batch after %s attempts", self.max_
↪attempts, exc_info=True)
```

flush_events()

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
def flush_events(self) -> None:
    if self.batch:
        with self.create_futures_executor() as futures_executor:
            futures_executor.submit(with_contextvars(self._safe_publish_event_payload_
↪batch), self.batch)
        self._batch = []
```

publish_event(event)

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
def publish_event(self, event: BaseEvent | dict) -> None:
    event_payload = event if isinstance(event, dict) else event.to_dict()

    with self.create_futures_executor() as futures_executor:
        if self.batched:
            self._batch.append(event_payload)
            if len(self.batch) >= self.batch_size:
                futures_executor.submit(with_contextvars(self._safe_publish_event_
↪payload_batch), self.batch)
                self._batch = []
        else:
            futures_executor.submit(with_contextvars(self._safe_publish_event_
↪payload), event_payload)
```

try_publish_event_payload(event_payload)abstractmethod

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
@abstractmethod
def try_publish_event_payload(self, event_payload: dict) -> None: ...
```

try_publish_event_payload_batch(event_payload_batch)abstractmethod

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
@abstractmethod
def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None: ..
↪.
```

BaseFileManagerDriver

Bases:

ABC

Attributes

Name	Type	Description
default_loader	``	The default loader to use for loading file contents into artifacts.
loaders	``	Dictionary of file extension specific loaders to use for loading file contents into artifacts.

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
@define
class BaseFileManagerDriver(ABC):
    """BaseFileManagerDriver can be used to list, load, and save files.

    Attributes:
        default_loader: The default loader to use for loading file contents into␣
↪artifacts.
        loaders: Dictionary of file extension specific loaders to use for loading␣
↪file contents into artifacts.
    """

    _workdir: str = field(kw_only=True, alias="workdir")
    encoding: Optional[str] = field(default=None, kw_only=True)

    @property
    @abstractmethod
    def workdir(self) -> str: ...

    @workdir.setter
    @abstractmethod
    def workdir(self, value: str) -> None: ...

    def list_files(self, path: str) -> TextArtifact:
        entries = self.try_list_files(path)
        return TextArtifact("\n".join(list(entries)))

    @abstractmethod
    def try_list_files(self, path: str) -> list[str]: ...

    def load_file(self, path: str) -> BlobArtifact | TextArtifact:
        if self.encoding is None:
            return BlobArtifact(self.try_load_file(path))
        return TextArtifact(self.try_load_file(path).decode(encoding=self.encoding),␣
↪encoding=self.encoding)

    @abstractmethod
    def try_load_file(self, path: str) -> bytes: ...

    def save_file(self, path: str, value: bytes | str) -> InfoArtifact:
        if isinstance(value, str):
            value = value.encode() if self.encoding is None else value.
↪encode(encoding=self.encoding)
        elif isinstance(value, (bytearray, memoryview)):
            raise ValueError(f"Unsupported type: {type(value)}")

        location = self.try_save_file(path, value)

        return InfoArtifact(f"Successfully saved file at: {location}")

    @abstractmethod
    def try_save_file(self, path: str, value: bytes) -> str: ...

    def load_artifact(self, path: str) -> BaseArtifact:
```

(continues on next page)

(continued from previous page)

```
        response = self.try_load_file(path)
        return BaseArtifact.from_json(
            response.decode() if self.encoding is None else response.
↪decode(encoding=self.encoding)
        )

    def save_artifact(self, path: str, artifact: BaseArtifact) -> InfoArtifact:
        artifact_json = artifact.to_json()
        value = artifact_json.encode() if self.encoding is None else artifact_json.
↪encode(encoding=self.encoding)

        location = self.try_save_file(path, value)

        return InfoArtifact(f"Successfully saved artifact at: {location}")
```

- `_workdir = field(kw_only=True, alias='workdir')` class-attribute instance-attribute
- `encoding = field(default=None, kw_only=True)` class-attribute instance-attribute
- `workdir` abstractmethod property writable

list_files(path)

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
def list_files(self, path: str) -> TextArtifact:
    entries = self.try_list_files(path)
    return TextArtifact("\n".join(list(entries)))
```

load_artifact(path)

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
def load_artifact(self, path: str) -> BaseArtifact:
    response = self.try_load_file(path)
    return BaseArtifact.from_json(
        response.decode() if self.encoding is None else response.decode(encoding=self.
↪encoding)
    )
```

load_file(path)

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
def load_file(self, path: str) -> BlobArtifact | TextArtifact:
    if self.encoding is None:
        return BlobArtifact(self.try_load_file(path))
    return TextArtifact(self.try_load_file(path).decode(encoding=self.encoding),␣
↪encoding=self.encoding)
```

save_artifact(path, artifact)

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
def save_artifact(self, path: str, artifact: BaseArtifact) -> InfoArtifact:
    artifact_json = artifact.to_json()
    value = artifact_json.encode() if self.encoding is None else artifact_json.
↪encode(encoding=self.encoding)

    location = self.try_save_file(path, value)

    return InfoArtifact(f"Successfully saved artifact at: {location}")
```

save_file(path, value)

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
def save_file(self, path: str, value: bytes | str) -> InfoArtifact:
    if isinstance(value, str):
        value = value.encode() if self.encoding is None else value.
↪encode(encoding=self.encoding)
    elif isinstance(value, (bytearray, memoryview)):
        raise ValueError(f"Unsupported type: {type(value)}")

    location = self.try_save_file(path, value)

    return InfoArtifact(f"Successfully saved file at: {location}")
```

try_list_files(path)abstractmethod

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
@abstractmethod
def try_list_files(self, path: str) -> list[str]: ...
```

try_load_file(path)abstractmethod

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
@abstractmethod
def try_load_file(self, path: str) -> bytes: ...
```

try_save_file(path, value)abstractmethod

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
@abstractmethod
def try_save_file(self, path: str, value: bytes) -> str: ...
```

BaseImageGenerationDriver

Bases: *SerializableMixin* “) , *ExponentialBackoffMixin* ”) , ABC

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
@define
class BaseImageGenerationDriver(SerializableMixin, ExponentialBackoffMixin, ABC):
    model: str = field(kw_only=True, metadata={"serializable": True})

    def before_run(self, prompts: list[str], negative_prompts: Optional[list[str]] =␣
↪None) -> None:
        EventBus.publish_event(StartImageGenerationEvent(prompts=prompts, negative_
↪prompts=negative_prompts))

    def after_run(self) -> None:
        EventBus.publish_event(FinishImageGenerationEvent())

    def run_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact:
        for attempt in self.retrying():
            with attempt:
                self.before_run(prompts, negative_prompts)
                result = self.try_text_to_image(prompts, negative_prompts)
                self.after_run()

                return result

        raise Exception("Failed to run text to image generation")

    def run_image_variation(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        for attempt in self.retrying():
            with attempt:
                self.before_run(prompts, negative_prompts)
                result = self.try_image_variation(prompts, image, negative_prompts)
                self.after_run()

                return result

        raise Exception("Failed to generate image variations")

    def run_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        for attempt in self.retrying():
            with attempt:
                self.before_run(prompts, negative_prompts)
                result = self.try_image_inpainting(prompts, image, mask, negative_
↪prompts)
                self.after_run()
```

(continues on next page)

(continued from previous page)

```
                return result

        raise Exception("Failed to run image inpainting")

    def run_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        for attempt in self.retrying():
            with attempt:
                self.before_run(prompts, negative_prompts)
                result = self.try_image_outpainting(prompts, image, mask, negative_
↪prompts)
                self.after_run()

                return result

        raise Exception("Failed to run image outpainting")

    @abstractmethod
    def try_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact: ...

    @abstractmethod
    def try_image_variation(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact: ...

    @abstractmethod
    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact: ...

    @abstractmethod
    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact: ...
```

- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

after_run()

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
def after_run(self) -> None:
    EventBus.publish_event(FinishImageGenerationEvent())
```

before_run(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
def before_run(self, prompts: list[str], negative_prompts: Optional[list[str]] =␣
↪None) -> None:
    EventBus.publish_event(StartImageGenerationEvent(prompts=prompts, negative_
↪prompts=negative_prompts))
```

run_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
def run_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    for attempt in self.retrying():
        with attempt:
            self.before_run(prompts, negative_prompts)
            result = self.try_image_inpainting(prompts, image, mask, negative_prompts)
            self.after_run()

            return result

    raise Exception("Failed to run image inpainting")
```

run_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
def run_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    for attempt in self.retrying():
        with attempt:
```

(continues on next page)

(continued from previous page)

```
            self.before_run(prompts, negative_prompts)
            result = self.try_image_outpainting(prompts, image, mask, negative_
↪prompts)
            self.after_run()

            return result

    raise Exception("Failed to run image outpainting")
```

run_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
def run_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    for attempt in self.retrying():
        with attempt:
            self.before_run(prompts, negative_prompts)
            result = self.try_image_variation(prompts, image, negative_prompts)
            self.after_run()

            return result

    raise Exception("Failed to generate image variations")
```

run_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
def run_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    for attempt in self.retrying():
        with attempt:
            self.before_run(prompts, negative_prompts)
            result = self.try_text_to_image(prompts, negative_prompts)
            self.after_run()

            return result

    raise Exception("Failed to run text to image generation")
```

try_image_inpainting(prompts, image, mask, negative_prompts=None)abstractmethod

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
@abstractmethod
def try_image_inpainting(
    self,
```

(continues on next page)

(continued from previous page)

```
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact: ...
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)abstractmethod

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
@abstractmethod
def try_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact: ...
```

try_image_variation(prompts, image, negative_prompts=None)abstractmethod

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
@abstractmethod
def try_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact: ...
```

try_text_to_image(prompts, negative_prompts=None)abstractmethod

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
@abstractmethod
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact: ...
```

BaseImageGenerationModelDriver

Bases: *SerializableMixin* “) , ABC

Source Code in griptape/drivers/image_generation_model/base_image_generation_model_driver.py

```
@define
class BaseImageGenerationModelDriver(SerializableMixin, ABC):
    @abstractmethod
    def get_generated_image(self, response: dict) -> bytes: ...

    @abstractmethod
    def text_to_image_request_parameters(
```

(continues on next page)

(continued from previous page)

```
        self,
        prompts: list[str],
        image_width: int,
        image_height: int,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict[str, Any]: ...

    @abstractmethod
    def image_variation_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict[str, Any]: ...

    @abstractmethod
    def image_inpainting_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict[str, Any]: ...

    @abstractmethod
    def image_outpainting_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict[str, Any]: ...
```

get_generated_image(response)abstractmethod

Source Code in griptape/drivers/image_generation_model/base_image_generation_model_driver.py

```
@abstractmethod
def get_generated_image(self, response: dict) -> bytes: ...
```

image_inpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)abstractmethod

Source Code in griptape/drivers/image_generation_model/base_image_generation_model_driver.py

```
@abstractmethod
def image_inpainting_request_parameters(
    self,
```

(continues on next page)

(continued from previous page)

```
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict[str, Any]: ...
```

image_outpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)abstractmethod

Source Code in griptape/drivers/image_generation_model/base_image_generation_model_driver.py

```
@abstractmethod
def image_outpainting_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict[str, Any]: ...
```

image_variation_request_parameters(prompts, image, negative_prompts=None, seed=None)abstractmethod

Source Code in griptape/drivers/image_generation_model/base_image_generation_model_driver.py

```
@abstractmethod
def image_variation_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict[str, Any]: ...
```

text_to_image_request_parameters(prompts, image_width, image_height, negative_prompts=None, seed=None)abstractmethod

Source Code in griptape/drivers/image_generation_model/base_image_generation_model_driver.py

```
@abstractmethod
def text_to_image_request_parameters(
    self,
    prompts: list[str],
    image_width: int,
    image_height: int,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict[str, Any]: ...
```

BaseMultiModelImageGenerationDriver

Bases: *BaseImageGenerationDriver* “) , ABC

Attributes

Name	Type	Description
<a href="./#griptape.drivers.BaseMultiModelImageGenerationDriver.image_generation_model_driver" title="image_generation_model_driver = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.BaseMultiModelImageGenerationDriver.image_generation_model_driver)"><co class="language-python">	<a href="image_generation_model#griptape.drivers.image_generation_model.BaseImageGenerationModelDriver" title="BaseImageGenerationModelDriver (griptape.drivers.image_generation_model.BaseImageGenerationModelDriver)"><code class="language-python">BaseImageGenerationModel	Image Model Driver to use.

Source Code in griptape/drivers/image_generation/base_multi_model_image_generation_driver.py

```
@define
class BaseMultiModelImageGenerationDriver(BaseImageGenerationDriver, ABC):
    """Image Generation Driver for platforms like Amazon Bedrock that host many LLM␣
↪models.

    Instances of this Image Generation Driver require a Image Generation Model Driver␣
↪which is used to structure the
    image generation request in the format required by the model and to process the␣
↪output.

    Attributes:
        image_generation_model_driver: Image Model Driver to use.
    """

    image_generation_model_driver: BaseImageGenerationModelDriver = field(kw_
↪only=True, metadata={"serializable": True})
```

- `image_generation_model_driver = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

BaseObservabilityDriver

Bases:

ABC

Source Code in griptape/drivers/observability/base_observability_driver.py

```
@define
class BaseObservabilityDriver(ABC):
    def __enter__(self) -> None:  # noqa: B027
        pass

    def __exit__(
        self,
        exc_type: Optional[type[BaseException]],
        exc_value: Optional[BaseException],
        exc_traceback: Optional[TracebackType],
    ) -> bool:
        return False

    @abstractmethod
    def observe(self, call: Observable.Call) -> Any: ...

    @abstractmethod
    def get_span_id(self) -> Optional[str]: ...
```

enter()

Source Code in griptape/drivers/observability/base_observability_driver.py

```
def __enter__(self) -> None:  # noqa: B027
    pass
```

exit(exc_type, exc_value, exc_traceback)

Source Code in griptape/drivers/observability/base_observability_driver.py

```
def __exit__(
    self,
    exc_type: Optional[type[BaseException]],
    exc_value: Optional[BaseException],
    exc_traceback: Optional[TracebackType],
) -> bool:
    return False
```

get_span_id()abstractmethod

Source Code in griptape/drivers/observability/base_observability_driver.py

```
@abstractmethod
def get_span_id(self) -> Optional[str]: ...
```

observe(call)abstractmethod

Source Code in griptape/drivers/observability/base_observability_driver.py

```
@abstractmethod
def observe(self, call: Observable.Call) -> Any: ...
```

BasePromptDriver

Bases: *SerializableMixin* “) , *ExponentialBackoffMixin* ”) , ABC

Attributes

Name	Type	Description
<code class="language-python">	*float*	The temperature to use for the completion.
<code class="language-python">	*Optional[int]*	The maximum number of tokens to generate. If not specified, the value will be automatically generated based by the tokenizer.
	str	A function that converts a *PromptStack* to a string.

continues on next page

Table 60 – continued from previous page

Name	Type	Description
<code class="language-python">	*tuple[type[Exception], ...]*	A tuple of exception types to ignore.
<code class="language-python">	*str*	The model name.
<code class="language-python">	<code class="language-python">BaseTokenizer</code>	An instance of *BaseTokenizer* to when calculating tokens.
<a href="./#griptape.drivers.BasePromptDriver.stream" title="stream = field(default=False, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.BasePromptDriver.stream)"><code class="language-python">	*bool*	Whether to stream the completion or not. *CompletionChunkEvent`s will be published to the `Structure* if one is provided.

continues on next page

Table 60 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.BasePromptDriver.use_native_tools" title="use_native_tools = field(default=False, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.BasePromptDriver.use_native_tools)"><code class="language-python">	*bool*	Whether to use LLM's native function calling capabilities. Must be supported by the model.
<a href="./#griptape.drivers.BasePromptDriver.extra_params" title="extra_params = field(factory=dict, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.BasePromptDriver.extra_params)"><code class="language-python">	*dict*	Extra parameters to pass to the model.

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
@define(kw_only=True)
class BasePromptDriver(SerializableMixin, ExponentialBackoffMixin, ABC):
    """Base class for the Prompt Drivers.

    Attributes:
        temperature: The temperature to use for the completion.
        max_tokens: The maximum number of tokens to generate. If not specified, the␣
↪value will be automatically generated based by the tokenizer.
        prompt_stack_to_string: A function that converts a `PromptStack` to a string.
        ignored_exception_types: A tuple of exception types to ignore.
        model: The model name.
        tokenizer: An instance of `BaseTokenizer` to when calculating tokens.
        stream: Whether to stream the completion or not. `CompletionChunkEvent`s will␣
↪be published to the `Structure` if one is provided.
        use_native_tools: Whether to use LLMs native function calling capabilities.␣
↪Must be supported by the model.
        extra_params: Extra parameters to pass to the model.
    """
```

(continues on next page)

(continued from previous page)

```
    temperature: float = field(default=0.1, metadata={"serializable": True})
    max_tokens: Optional[int] = field(default=None, metadata={"serializable": True})
    ignored_exception_types: tuple[type[Exception], ...] =␣
↪field(default=Factory(lambda: (ImportError, ValueError)))
    model: str = field(metadata={"serializable": True})
    tokenizer: BaseTokenizer
    stream: bool = field(default=False, kw_only=True, metadata={"serializable": True})
    use_native_tools: bool = field(default=False, kw_only=True, metadata={
↪"serializable": True})
    structured_output_strategy: StructuredOutputStrategy = field(
        default="rule", kw_only=True, metadata={"serializable": True}
    )
    extra_params: dict = field(factory=dict, kw_only=True, metadata={"serializable":␣
↪True})

    def before_run(self, prompt_stack: PromptStack) -> None:
        self._init_structured_output(prompt_stack)
        EventBus.publish_event(StartPromptEvent(model=self.model, prompt_stack=prompt_
↪stack))

    def after_run(self, result: Message) -> None:
        EventBus.publish_event(
            FinishPromptEvent(
                model=self.model,
                result=result.value,
                input_token_count=result.usage.input_tokens,
                output_token_count=result.usage.output_tokens,
            ),
        )

    @observable(tags=["PromptDriver.run()"])
    def run(self, prompt_input: PromptStack | BaseArtifact) -> Message:
        if isinstance(prompt_input, BaseArtifact):
            prompt_stack = PromptStack.from_artifact(prompt_input)
        else:
            prompt_stack = prompt_input

        for attempt in self.retrying():
            with attempt:
                self.before_run(prompt_stack)

                result = self.__process_stream(prompt_stack) if self.stream else self.
↪__process_run(prompt_stack)

                self.after_run(result)

                return result
        raise Exception("prompt driver failed after all retry attempts")

    def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
        """Converts a Prompt Stack to a string for token counting or model prompt_
↪input.
```

(continues on next page)

(continued from previous page)

```
        This base implementation is only a rough approximation, and should be␣
↪overridden by subclasses with model-specific tokens.

        Args:
            prompt_stack: The Prompt Stack to convert to a string.

        Returns:
            A single string representation of the Prompt Stack.
        """
        prompt_lines = []

        for i in prompt_stack.messages:
            content = i.to_text()
            if i.is_user():
                prompt_lines.append(f"User: {content}")
            elif i.is_assistant():
                prompt_lines.append(f"Assistant: {content}")
            else:
                prompt_lines.append(content)

        prompt_lines.append("Assistant:")

        return "\n\n".join(prompt_lines)

    @abstractmethod
    def try_run(self, prompt_stack: PromptStack) -> Message: ...

    @abstractmethod
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]: ...

    def _init_structured_output(self, prompt_stack: PromptStack) -> None:
        from griptape.tools import StructuredOutputTool

        if (output_schema := prompt_stack.output_schema) is not None:
            if self.structured_output_strategy == "tool":
                structured_output_tool = StructuredOutputTool(output_schema=output_
↪schema)
                if structured_output_tool not in prompt_stack.tools:
                    prompt_stack.tools.append(structured_output_tool)
            elif self.structured_output_strategy == "rule":
                output_artifact = TextArtifact(JsonSchemaRule(prompt_stack.to_output_
↪json_schema()).to_text())
                system_messages = prompt_stack.system_messages
                if system_messages:
                    last_system_message = prompt_stack.system_messages[-1]
                    last_system_message.content.extend(
                        [
                            TextMessageContent(TextArtifact("\n\n")),
                            TextMessageContent(output_artifact),
                        ]
                    )
```

(continues on next page)

(continued from previous page)

```
                else:
                    prompt_stack.messages.insert(
                        0,
                        Message(
                            content=[TextMessageContent(output_artifact)],
                            role=Message.SYSTEM_ROLE,
                        ),
                    )

    def __process_run(self, prompt_stack: PromptStack) -> Message:
        return self.try_run(prompt_stack)

    def __process_stream(self, prompt_stack: PromptStack) -> Message:
        delta_contents: dict[int, list[BaseDeltaMessageContent]] = {}
        usage = DeltaMessage.Usage()

        # Aggregate all content deltas from the stream
        message_deltas = self.try_stream(prompt_stack)
        for message_delta in message_deltas:
            usage += message_delta.usage
            content = message_delta.content

            if content is not None:
                if content.index in delta_contents:
                    delta_contents[content.index].append(content)
                else:
                    delta_contents[content.index] = [content]
                if isinstance(content, TextDeltaMessageContent):
                    EventBus.publish_event(TextChunkEvent(token=content.text,␣
↪index=content.index))
                elif isinstance(content, AudioDeltaMessageContent) and content.data␣
↪is not None:
                    EventBus.publish_event(AudioChunkEvent(data=content.data))
                elif isinstance(content, ActionCallDeltaMessageContent):
                    EventBus.publish_event(
                        ActionChunkEvent(
                            partial_input=content.partial_input,
                            tag=content.tag,
                            name=content.name,
                            path=content.path,
                            index=content.index,
                        ),
                    )

        # Build a complete content from the content deltas
        return self.__build_message(list(delta_contents.values()), usage)

    def __build_message(
        self, delta_contents: list[list[BaseDeltaMessageContent]], usage:␣
↪DeltaMessage.Usage
    ) -> Message:
        content = []
```

(continues on next page)

(continued from previous page)

```
        for delta_content in delta_contents:
            text_deltas = [delta for delta in delta_content if isinstance(delta,␣
↪TextDeltaMessageContent)]
            audio_deltas = [delta for delta in delta_content if isinstance(delta,␣
↪AudioDeltaMessageContent)]
            action_deltas = [delta for delta in delta_content if isinstance(delta,␣
↪ActionCallDeltaMessageContent)]

            if text_deltas:
                content.append(TextMessageContent.from_deltas(text_deltas))
            if audio_deltas:
                content.append(AudioMessageContent.from_deltas(audio_deltas))
            if action_deltas:
                content.append(ActionCallMessageContent.from_deltas(action_deltas))

        return Message(
            content=content,
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(input_tokens=usage.input_tokens, output_tokens=usage.
↪output_tokens),
        )
```

- `extra_params = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `ignored_exception_types = field(default=Factory(lambda: (ImportError, ValueError)))` class-attribute instance-attribute
- `max_tokens = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(metadata={'serializable': True})` class-attribute instance-attribute
- `stream = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='rule', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `temperature = field(default=0.1, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer` instance-attribute
- `use_native_tools = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__build_message(delta_contents, usage)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def __build_message(
    self, delta_contents: list[list[BaseDeltaMessageContent]], usage: DeltaMessage.
↪Usage
) -> Message:
    content = []
```

(continues on next page)

(continued from previous page)

```
    for delta_content in delta_contents:
        text_deltas = [delta for delta in delta_content if isinstance(delta,␣
↪TextDeltaMessageContent)]
        audio_deltas = [delta for delta in delta_content if isinstance(delta,␣
↪AudioDeltaMessageContent)]
        action_deltas = [delta for delta in delta_content if isinstance(delta,␣
↪ActionCallDeltaMessageContent)]

        if text_deltas:
            content.append(TextMessageContent.from_deltas(text_deltas))
        if audio_deltas:
            content.append(AudioMessageContent.from_deltas(audio_deltas))
        if action_deltas:
            content.append(ActionCallMessageContent.from_deltas(action_deltas))

    return Message(
        content=content,
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(input_tokens=usage.input_tokens, output_tokens=usage.
↪output_tokens),
    )
```

__process_run(prompt_stack)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def __process_run(self, prompt_stack: PromptStack) -> Message:
    return self.try_run(prompt_stack)
```

__process_stream(prompt_stack)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def __process_stream(self, prompt_stack: PromptStack) -> Message:
    delta_contents: dict[int, list[BaseDeltaMessageContent]] = {}
    usage = DeltaMessage.Usage()

    # Aggregate all content deltas from the stream
    message_deltas = self.try_stream(prompt_stack)
    for message_delta in message_deltas:
        usage += message_delta.usage
        content = message_delta.content

        if content is not None:
            if content.index in delta_contents:
                delta_contents[content.index].append(content)
            else:
                delta_contents[content.index] = [content]
            if isinstance(content, TextDeltaMessageContent):
                EventBus.publish_event(TextChunkEvent(token=content.text,␣
↪index=content.index))
            elif isinstance(content, AudioDeltaMessageContent) and content.data is␣
```

(continues on next page)

(continued from previous page)

```
↪not None:
                EventBus.publish_event(AudioChunkEvent(data=content.data))
            elif isinstance(content, ActionCallDeltaMessageContent):
                EventBus.publish_event(
                    ActionChunkEvent(
                        partial_input=content.partial_input,
                        tag=content.tag,
                        name=content.name,
                        path=content.path,
                        index=content.index,
                    ),
                )

    # Build a complete content from the content deltas
    return self.__build_message(list(delta_contents.values()), usage)
```

_init_structured_output(prompt_stack)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def _init_structured_output(self, prompt_stack: PromptStack) -> None:
    from griptape.tools import StructuredOutputTool

    if (output_schema := prompt_stack.output_schema) is not None:
        if self.structured_output_strategy == "tool":
            structured_output_tool = StructuredOutputTool(output_schema=output_schema)
            if structured_output_tool not in prompt_stack.tools:
                prompt_stack.tools.append(structured_output_tool)
        elif self.structured_output_strategy == "rule":
            output_artifact = TextArtifact(JsonSchemaRule(prompt_stack.to_output_json_
↪schema()).to_text())
            system_messages = prompt_stack.system_messages
            if system_messages:
                last_system_message = prompt_stack.system_messages[-1]
                last_system_message.content.extend(
                    [
                        TextMessageContent(TextArtifact("\n\n")),
                        TextMessageContent(output_artifact),
                    ]
                )
            else:
                prompt_stack.messages.insert(
                    0,
                    Message(
                        content=[TextMessageContent(output_artifact)],
                        role=Message.SYSTEM_ROLE,
                    ),
                )
```

after_run(result)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def after_run(self, result: Message) -> None:
    EventBus.publish_event(
        FinishPromptEvent(
            model=self.model,
            result=result.value,
            input_token_count=result.usage.input_tokens,
            output_token_count=result.usage.output_tokens,
        ),
    )
```

before_run(prompt_stack)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def before_run(self, prompt_stack: PromptStack) -> None:
    self._init_structured_output(prompt_stack)
    EventBus.publish_event(StartPromptEvent(model=self.model, prompt_stack=prompt_
↪stack))
```

prompt_stack_to_string(prompt_stack)

Converts a Prompt Stack to a string for token counting or model prompt_input.

This base implementation is only a rough approximation, and should be overridden by subclasses with model-specific tokens.

Parameters

Name	Type	Description	Default
prompt_stack	<a href="../common#griptape.common.PromptStack" title="PromptStack (griptape.common.PromptStack)"><code class="language-pythc	The Prompt Stack to convert to a string. 	*required*

Returns

Type	Description
str	A single string representation of the Prompt Stack.

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
    """Converts a Prompt Stack to a string for token counting or model prompt_input.

    This base implementation is only a rough approximation, and should be overridden␣
↪by subclasses with model-specific tokens.

    Args:
        prompt_stack: The Prompt Stack to convert to a string.

    Returns:
        A single string representation of the Prompt Stack.
    """
    prompt_lines = []

    for i in prompt_stack.messages:
        content = i.to_text()
        if i.is_user():
            prompt_lines.append(f"User: {content}")
        elif i.is_assistant():
            prompt_lines.append(f"Assistant: {content}")
        else:
            prompt_lines.append(content)

    prompt_lines.append("Assistant:")

    return "\n\n".join(prompt_lines)
```

run(prompt_input)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
@observable(tags=["PromptDriver.run()"])
def run(self, prompt_input: PromptStack | BaseArtifact) -> Message:
    if isinstance(prompt_input, BaseArtifact):
        prompt_stack = PromptStack.from_artifact(prompt_input)
    else:
        prompt_stack = prompt_input

    for attempt in self.retrying():
        with attempt:
            self.before_run(prompt_stack)

            result = self.__process_stream(prompt_stack) if self.stream else self.__
↪process_run(prompt_stack)

            self.after_run(result)

            return result
    raise Exception("prompt driver failed after all retry attempts")
```

try_run(prompt_stack)abstractmethod

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
@abstractmethod
def try_run(self, prompt_stack: PromptStack) -> Message: ...
```

try_stream(prompt_stack)abstractmethod

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
@abstractmethod
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]: ...
```

BaseRerankDriver

Bases:

ABC

Source Code in griptape/drivers/rerank/base_rerank_driver.py

```
@define(kw_only=True)
class BaseRerankDriver(ABC):
    @abstractmethod
    def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]: ..
↪.
```

run(query, artifacts)abstractmethod

Source Code in griptape/drivers/rerank/base_rerank_driver.py

```
@abstractmethod
def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]: ...
```

BaseRulesetDriver

Bases: *SerializableMixin* “) , ABC

Attributes

Name	Type	Description
<a href="./#griptape.drivers.BaseRulesetDriver.raise_not_found" title="raise_not_found = field(default=True, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.BaseRulesetDriver.raise_not_found)" class="language-python">	*bool*	Whether to raise an error if the ruleset is not found. Defaults to True.

Source Code in griptape/drivers/ruleset/base_ruleset_driver.py

```
@define
class BaseRulesetDriver(SerializableMixin, ABC):
    """Base class for ruleset drivers.

    Attributes:
        raise_not_found: Whether to raise an error if the ruleset is not found.␣
↪Defaults to True.
    """

    raise_not_found: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})

    @abstractmethod
    def load(self, ruleset_name: str) -> tuple[list[BaseRule], dict[str, Any]]: ...

    def _from_ruleset_dict(self, params_dict: dict[str, Any]) -> tuple[list[BaseRule],
↪ dict[str, Any]]:
        return [
            self._get_rule(rule["value"], rule.get("meta", {})) for rule in params_
↪dict.get("rules", [])
        ], params_dict.get("meta", {})

    def _get_rule(self, value: Any, meta: dict[str, Any]) -> BaseRule:
        from griptape.rules import JsonSchemaRule, Rule

        return JsonSchemaRule(value=value, meta=meta) if isinstance(value, dict) else␣
↪Rule(value=str(value), meta=meta)
```

- `raise_not_found = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_from_ruleset_dict(params_dict)

Source Code in griptape/drivers/ruleset/base_ruleset_driver.py

```
def _from_ruleset_dict(self, params_dict: dict[str, Any]) -> tuple[list[BaseRule],␣
↪dict[str, Any]]:
    return [
        self._get_rule(rule["value"], rule.get("meta", {})) for rule in params_dict.
↪get("rules", [])
    ], params_dict.get("meta", {})
```

_get_rule(value, meta)

Source Code in griptape/drivers/ruleset/base_ruleset_driver.py

```
def _get_rule(self, value: Any, meta: dict[str, Any]) -> BaseRule:
    from griptape.rules import JsonSchemaRule, Rule

    return JsonSchemaRule(value=value, meta=meta) if isinstance(value, dict) else␣
↪Rule(value=str(value), meta=meta)
```

load(ruleset_name)abstractmethod

Source Code in griptape/drivers/ruleset/base_ruleset_driver.py

```
@abstractmethod
def load(self, ruleset_name: str) -> tuple[list[BaseRule], dict[str, Any]]: ...
```

BaseSqlDriver

Bases:

ABC

Source Code in griptape/drivers/sql/base_sql_driver.py

```
@define
class BaseSqlDriver(ABC):
    @dataclass
    class RowResult:
        cells: dict[str, Any]

    @abstractmethod
    def execute_query(self, query: str) -> Optional[list[RowResult]]: ...

    @abstractmethod
    def execute_query_raw(self, query: str) -> Optional[list[dict[str, Any]]]: ...

    @abstractmethod
    def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]: ...
```

RowResultdataclass

Source Code in griptape/drivers/sql/base_sql_driver.py

```
@dataclass
class RowResult:
    cells: dict[str, Any]
```

- `cells` instance-attribute

execute_query(query)abstractmethod

Source Code in griptape/drivers/sql/base_sql_driver.py

```
@abstractmethod
def execute_query(self, query: str) -> Optional[list[RowResult]]: ...
```

execute_query_raw(query)abstractmethod

Source Code in griptape/drivers/sql/base_sql_driver.py

```
@abstractmethod
def execute_query_raw(self, query: str) -> Optional[list[dict[str, Any]]]: ...
```

get_table_schema(table_name, schema=None)abstractmethod

Source Code in griptape/drivers/sql/base_sql_driver.py

```
@abstractmethod
def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]: ...
```

BaseStructureRunDriver

Bases:

ABC

Attributes

Name	Type	Description
<a href="./#griptape.drivers.BaseStructureRunDriver.env" title="env = field(default=Factory(dict), kw_only=True) class-attribute instance-attribute (griptape.drivers.BaseStructureRunDriver.env)"><code class="language-python">	*dict[str, str]*	Environment variables to set before running the Structure.

Source Code in griptape/drivers/structure_run/base_structure_run_driver.py

```
@define
class BaseStructureRunDriver(ABC):
    """Base class for Structure Run Drivers.

    Attributes:
        env: Environment variables to set before running the Structure.
    """

    env: dict[str, str] = field(default=Factory(dict), kw_only=True)

    def run(self, *args: BaseArtifact) -> BaseArtifact:
        return self.try_run(*args)

    @abstractmethod
    def try_run(self, *args: BaseArtifact) -> BaseArtifact: ...
```

- `env = field(default=Factory(dict), kw_only=True)` class-attribute instance-attribute

run(*args)

Source Code in griptape/drivers/structure_run/base_structure_run_driver.py

```
def run(self, *args: BaseArtifact) -> BaseArtifact:
    return self.try_run(*args)
```

try_run(*args)abstractmethod

Source Code in griptape/drivers/structure_run/base_structure_run_driver.py

```
@abstractmethod
def try_run(self, *args: BaseArtifact) -> BaseArtifact: ...
```

BaseTextToSpeechDriver

Bases: *SerializableMixin* “) , *ExponentialBackoffMixin* ”) , ABC

Source Code in griptape/drivers/text_to_speech/base_text_to_speech_driver.py

```python
@define
class BaseTextToSpeechDriver(SerializableMixin, ExponentialBackoffMixin, ABC):
    model: str = field(kw_only=True, metadata={"serializable": True})

    def before_run(self, prompts: list[str]) -> None:
        EventBus.publish_event(StartTextToSpeechEvent(prompts=prompts))

    def after_run(self) -> None:
        EventBus.publish_event(FinishTextToSpeechEvent())

    def run_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
        for attempt in self.retrying():
            with attempt:
                self.before_run(prompts)
                result = self.try_text_to_audio(prompts)
                self.after_run()

                return result

        raise Exception("Failed to run text to audio generation")

    @abstractmethod
    def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact: ...
```

- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

after_run()

Source Code in griptape/drivers/text_to_speech/base_text_to_speech_driver.py

```python
def after_run(self) -> None:
    EventBus.publish_event(FinishTextToSpeechEvent())
```

before_run(prompts)

Source Code in griptape/drivers/text_to_speech/base_text_to_speech_driver.py

```python
def before_run(self, prompts: list[str]) -> None:
    EventBus.publish_event(StartTextToSpeechEvent(prompts=prompts))
```

run_text_to_audio(prompts)

Source Code in griptape/drivers/text_to_speech/base_text_to_speech_driver.py

```python
def run_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
    for attempt in self.retrying():
        with attempt:
            self.before_run(prompts)
            result = self.try_text_to_audio(prompts)
            self.after_run()

            return result
```

(continues on next page)

(continued from previous page)

```
    raise Exception("Failed to run text to audio generation")
```

try_text_to_audio(prompts)abstractmethod

Source Code in griptape/drivers/text_to_speech/base_text_to_speech_driver.py

```
@abstractmethod
def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact: ...
```

BaseVectorStoreDriver

Bases: *SerializableMixin* “) , *FuturesExecutorMixin* ”) , ABC

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
@define
class BaseVectorStoreDriver(SerializableMixin, FuturesExecutorMixin, ABC):
    DEFAULT_QUERY_COUNT = 5

    @define
    class Entry(SerializableMixin):
        id: str = field(metadata={"serializable": True})
        vector: Optional[list[float]] = field(default=None, metadata={"serializable":␣
↪True})
        score: Optional[float] = field(default=None, metadata={"serializable": True})
        meta: Optional[dict] = field(default=None, metadata={"serializable": True})
        namespace: Optional[str] = field(default=None, metadata={"serializable": True}
↪)

        def to_artifact(self) -> BaseArtifact:
            return BaseArtifact.from_json(self.meta["artifact"])  # pyright:␣
↪ignore[reportOptionalSubscript]

    embedding_driver: BaseEmbeddingDriver = field(kw_only=True, metadata={
↪"serializable": True})

    def upsert_text_artifacts(
        self,
        artifacts: list[TextArtifact] | dict[str, list[TextArtifact]],
        *,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> list[str] | dict[str, list[str]]:
        warnings.warn(
            "`BaseVectorStoreDriver.upsert_text_artifacts` is deprecated and will be␣
↪removed in a future release. `BaseVectorStoreDriver.upsert_collection` is a drop-in␣
↪replacement.",
            DeprecationWarning,
            stacklevel=2,
        )
        return self.upsert_collection(artifacts, meta=meta, **kwargs)
```

(continues on next page)

(continued from previous page)

```
    def upsert_text_artifact(
        self,
        artifact: TextArtifact,
        *,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        vector_id: Optional[str] = None,
        **kwargs,
    ) -> str:
        warnings.warn(
            "`BaseVectorStoreDriver.upsert_text_artifacts` is deprecated and will be␣
↪removed in a future release. `BaseVectorStoreDriver.upsert` is a drop-in␣
↪replacement.",
            DeprecationWarning,
            stacklevel=2,
        )
        return self.upsert(artifact, namespace=namespace, meta=meta, vector_id=vector_
↪id, **kwargs)

    def upsert_text(
        self,
        string: str,
        *,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        vector_id: Optional[str] = None,
        **kwargs,
    ) -> str:
        warnings.warn(
            "`BaseVectorStoreDriver.upsert_text` is deprecated and will be removed in␣
↪a future release. `BaseVectorStoreDriver.upsert` is a drop-in replacement.",
            DeprecationWarning,
            stacklevel=2,
        )
        return self.upsert(string, namespace=namespace, meta=meta, vector_id=vector_
↪id, **kwargs)

    @overload
    def upsert_collection(
        self,
        artifacts: list[TextArtifact] | list[ImageArtifact],
        *,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> list[str]: ...

    @overload
    def upsert_collection(
        self,
        artifacts: dict[str, list[TextArtifact]] | dict[str, list[ImageArtifact]],
        *,
```

(continues on next page)

(continued from previous page)

```
        meta: Optional[dict] = None,
        **kwargs,
    ) -> dict[str, list[str]]: ...

    def upsert_collection(
        self,
        artifacts: list[TextArtifact]
        | list[ImageArtifact]
        | dict[str, list[TextArtifact]]
        | dict[str, list[ImageArtifact]],
        *,
        meta: Optional[dict] = None,
        **kwargs,
    ):
        with self.create_futures_executor() as futures_executor:
            if isinstance(artifacts, list):
                return utils.execute_futures_list(
                    [
                        futures_executor.submit(with_contextvars(self.upsert), a,␣
↪namespace=None, meta=meta, **kwargs)
                        for a in artifacts
                    ],
                )
            futures_dict = {}

            for namespace, artifact_list in artifacts.items():
                for a in artifact_list:
                    if not futures_dict.get(namespace):
                        futures_dict[namespace] = []

                    futures_dict[namespace].append(
                        futures_executor.submit(
                            with_contextvars(self.upsert), a, namespace=namespace,␣
↪meta=meta, **kwargs
                        )
                    )

            return utils.execute_futures_list_dict(futures_dict)

    def upsert(
        self,
        value: str | TextArtifact | ImageArtifact,
        *,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        vector_id: Optional[str] = None,
        **kwargs,
    ) -> str:
        artifact = TextArtifact(value) if isinstance(value, str) else value

        meta = {} if meta is None else meta
```

(continues on next page)

(continued from previous page)

```
        if vector_id is None:
            value = artifact.to_text() if artifact.reference is None else artifact.to_
↪text() + str(artifact.reference)
            vector_id = self._get_default_vector_id(value)

        if self.does_entry_exist(vector_id, namespace=namespace):
            return vector_id
        meta = {**meta, "artifact": artifact.to_json()}

        vector = self.embedding_driver.embed(artifact, vector_operation="upsert")

        return self.upsert_vector(vector, vector_id=vector_id, namespace=namespace,␣
↪meta=meta, **kwargs)

    def does_entry_exist(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪bool:
        try:
            return self.load_entry(vector_id, namespace=namespace) is not None
        except Exception:
            return False

    def load_artifacts(self, *, namespace: Optional[str] = None) -> ListArtifact:
        result = self.load_entries(namespace=namespace)
        artifacts = [r.to_artifact() for r in result]

        return ListArtifact([a for a in artifacts if isinstance(a, TextArtifact)])

    @abstractmethod
    def delete_vector(self, vector_id: str) -> None: ...

    @abstractmethod
    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str: ...

    @abstractmethod
    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[Entry]: ...

    @abstractmethod
    def load_entries(self, *, namespace: Optional[str] = None) -> list[Entry]: ...

    def query_vector(
        self,
        vector: list[float],
        *,
```

(continues on next page)

(continued from previous page)

```
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[Entry]:
        # TODO: Mark as abstract method for griptape 2.0
        raise NotImplementedError(f"{self.__class__.__name__} does not support vector query.")

    def query(
        self,
        query: str | TextArtifact | ImageArtifact,
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[Entry]:
        try:
            vector = self.embedding_driver.embed(query, vector_operation="query")
        except ValueError as e:
            raise ValueError(
                "The Embedding Driver, %s, used by the Vector Store does not support embedding the %s type."
                "To resolve, provide an Embedding Driver that supports this type.",
                self.embedding_driver.__class__.__name__,
                type(query),
            ) from e
        return self.query_vector(vector, count=count, namespace=namespace, include_vectors=include_vectors, **kwargs)

    def _get_default_vector_id(self, value: str) -> str:
        return str(uuid.uuid5(uuid.NAMESPACE_OID, value))
```

- `DEFAULT_QUERY_COUNT = 5` class-attribute instance-attribute
- `embedding_driver = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

Entry

Bases: *SerializableMixin* “)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
@define
class Entry(SerializableMixin):
    id: str = field(metadata={"serializable": True})
    vector: Optional[list[float]] = field(default=None, metadata={"serializable": True})
    score: Optional[float] = field(default=None, metadata={"serializable": True})
    meta: Optional[dict] = field(default=None, metadata={"serializable": True})
    namespace: Optional[str] = field(default=None, metadata={"serializable": True})
```

(continues on next page)

(continued from previous page)

```
    def to_artifact(self) -> BaseArtifact:
        return BaseArtifact.from_json(self.meta["artifact"])  # pyright:␣
↪ignore[reportOptionalSubscript]
```

- `id = field(metadata={'serializable': True})` class-attribute instance-attribute
- `meta = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `namespace = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `score = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `vector = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

_get_default_vector_id(value)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def _get_default_vector_id(self, value: str) -> str:
    return str(uuid.uuid5(uuid.NAMESPACE_OID, value))
```

delete_vector(vector_id)abstractmethod

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
@abstractmethod
def delete_vector(self, vector_id: str) -> None: ...
```

does_entry_exist(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def does_entry_exist(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪bool:
    try:
        return self.load_entry(vector_id, namespace=namespace) is not None
    except Exception:
        return False
```

load_artifacts(*, namespace=None)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def load_artifacts(self, *, namespace: Optional[str] = None) -> ListArtifact:
    result = self.load_entries(namespace=namespace)
    artifacts = [r.to_artifact() for r in result]

    return ListArtifact([a for a in artifacts if isinstance(a, TextArtifact)])
```

load_entries(*, namespace=None)abstractmethod

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
@abstractmethod
def load_entries(self, *, namespace: Optional[str] = None) -> list[Entry]: ...
```

load_entry(vector_id, *, namespace=None)abstractmethod

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
@abstractmethod
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[Entry]: ...
```

query(query, *, count=None, namespace=None, include_vectors=False, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def query(
    self,
    query: str | TextArtifact | ImageArtifact,
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[Entry]:
    try:
        vector = self.embedding_driver.embed(query, vector_operation="query")
    except ValueError as e:
        raise ValueError(
            "The Embedding Driver, %s, used by the Vector Store does not support␣
↪embedding the %s type."
            "To resolve, provide an Embedding Driver that supports this type.",
            self.embedding_driver.__class__.__name__,
            type(query),
        ) from e
    return self.query_vector(vector, count=count, namespace=namespace, include_
↪vectors=include_vectors, **kwargs)
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[Entry]:
```

(continues on next page)

(continued from previous page)

```
    # TODO: Mark as abstract method for griptape 2.0
    raise NotImplementedError(f"{self.__class__.__name__} does not support vector␣
↪query.")
```

upsert(value, *, namespace=None, meta=None, vector_id=None, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def upsert(
    self,
    value: str | TextArtifact | ImageArtifact,
    *,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    vector_id: Optional[str] = None,
    **kwargs,
) -> str:
    artifact = TextArtifact(value) if isinstance(value, str) else value

    meta = {} if meta is None else meta

    if vector_id is None:
        value = artifact.to_text() if artifact.reference is None else artifact.to_
↪text() + str(artifact.reference)
        vector_id = self._get_default_vector_id(value)

    if self.does_entry_exist(vector_id, namespace=namespace):
        return vector_id
    meta = {**meta, "artifact": artifact.to_json()}

    vector = self.embedding_driver.embed(artifact, vector_operation="upsert")

    return self.upsert_vector(vector, vector_id=vector_id, namespace=namespace,␣
↪meta=meta, **kwargs)
```

upsert_collection(artifacts, *, meta=None, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def upsert_collection(
    self,
    artifacts: list[TextArtifact]
    | list[ImageArtifact]
    | dict[str, list[TextArtifact]]
    | dict[str, list[ImageArtifact]],
    *,
    meta: Optional[dict] = None,
    **kwargs,
):
    with self.create_futures_executor() as futures_executor:
        if isinstance(artifacts, list):
            return utils.execute_futures_list(
```

(continues on next page)

(continued from previous page)

```
                [
                    futures_executor.submit(with_contextvars(self.upsert), a,␣
↪namespace=None, meta=meta, **kwargs)
                    for a in artifacts
                ],
            )
        futures_dict = {}

        for namespace, artifact_list in artifacts.items():
            for a in artifact_list:
                if not futures_dict.get(namespace):
                    futures_dict[namespace] = []

                futures_dict[namespace].append(
                    futures_executor.submit(
                        with_contextvars(self.upsert), a, namespace=namespace,␣
↪meta=meta, **kwargs
                    )
                )

        return utils.execute_futures_list_dict(futures_dict)
```

upsert_text(string, *, namespace=None, meta=None, vector_id=None, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def upsert_text(
    self,
    string: str,
    *,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    vector_id: Optional[str] = None,
    **kwargs,
) -> str:
    warnings.warn(
        "`BaseVectorStoreDriver.upsert_text` is deprecated and will be removed in a␣
↪future release. `BaseVectorStoreDriver.upsert` is a drop-in replacement.",
        DeprecationWarning,
        stacklevel=2,
    )
    return self.upsert(string, namespace=namespace, meta=meta, vector_id=vector_id,␣
↪**kwargs)
```

upsert_text_artifact(artifact, *, namespace=None, meta=None, vector_id=None, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def upsert_text_artifact(
    self,
    artifact: TextArtifact,
    *,
```

(continues on next page)

(continued from previous page)

```
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    vector_id: Optional[str] = None,
    **kwargs,
) -> str:
    warnings.warn(
        "`BaseVectorStoreDriver.upsert_text_artifacts` is deprecated and will be␣
↪removed in a future release. `BaseVectorStoreDriver.upsert` is a drop-in␣
↪replacement.",
        DeprecationWarning,
        stacklevel=2,
    )
    return self.upsert(artifact, namespace=namespace, meta=meta, vector_id=vector_id,␣
↪**kwargs)
```

upsert_text_artifacts(artifacts, *, meta=None, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def upsert_text_artifacts(
    self,
    artifacts: list[TextArtifact] | dict[str, list[TextArtifact]],
    *,
    meta: Optional[dict] = None,
    **kwargs,
) -> list[str] | dict[str, list[str]]:
    warnings.warn(
        "`BaseVectorStoreDriver.upsert_text_artifacts` is deprecated and will be␣
↪removed in a future release. `BaseVectorStoreDriver.upsert_collection` is a drop-in␣
↪replacement.",
        DeprecationWarning,
        stacklevel=2,
    )
    return self.upsert_collection(artifacts, meta=meta, **kwargs)
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)abstractmethod

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
@abstractmethod
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str: ...
```

BaseWebScraperDriver

Bases:

ABC

Source Code in griptape/drivers/web_scraper/base_web_scraper_driver.py

```
class BaseWebScraperDriver(ABC):
    def scrape_url(self, url: str) -> TextArtifact:
        source = self.fetch_url(url)

        return self.extract_page(source)

    @abstractmethod
    def fetch_url(self, url: str) -> str: ...

    @abstractmethod
    def extract_page(self, page: str) -> TextArtifact: ...
```

extract_page(page)abstractmethod

Source Code in griptape/drivers/web_scraper/base_web_scraper_driver.py

```
@abstractmethod
def extract_page(self, page: str) -> TextArtifact: ...
```

fetch_url(url)abstractmethod

Source Code in griptape/drivers/web_scraper/base_web_scraper_driver.py

```
@abstractmethod
def fetch_url(self, url: str) -> str: ...
```

scrape_url(url)

Source Code in griptape/drivers/web_scraper/base_web_scraper_driver.py

```
def scrape_url(self, url: str) -> TextArtifact:
    source = self.fetch_url(url)

    return self.extract_page(source)
```

BaseWebSearchDriver

Bases:

ABC

Source Code in griptape/drivers/web_search/base_web_search_driver.py

```
@define
class BaseWebSearchDriver(ABC):
    results_count: int = field(default=5, kw_only=True)
```

(continues on next page)

(continued from previous page)

```
    @abstractmethod
    def search(self, query: str, **kwargs) -> ListArtifact: ...
```

- `results_count = field(default=5, kw_only=True)` class-attribute instance-attribute

search(query, **kwargs)abstractmethod

Source Code in griptape/drivers/web_search/base_web_search_driver.py

```
@abstractmethod
def search(self, query: str, **kwargs) -> ListArtifact: ...
```

BedrockStableDiffusionImageGenerationModelDriver

Bases: *BaseImageGenerationModelDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.BedrockStableDiffusionImageGenerationModelDriver.cfg_scale" title="cfg_scale = field(default=7, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.BedrockStableDiffusionImageGenerationModelDriver.cfg_scale)"><co class="language-python">	*int*	Specifies how strictly image generation follows the provided prompt. Defaults to 7.
mask_source	*int*	Specifies mask image configuration for image-to-image generations. Defaults to "MASK_IMAGE_BLACK".

continues on next page

Table 65 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.BedrockStableDiffusionImageGenerationModelDriver.style_preset" title="style_preset = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.BedrockStableDiffusionImageGenerationModelDriver.style_preset)"><code class="language-python">	*Optional[str]*	If provided, specifies a specific image generation style preset.
<a href="./#griptape.drivers.BedrockStableDiffusionImageGenerationModelDriver.clip_guidance_preset" title="clip_guidance_preset = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.BedrockStableDiffusionImageGenerationModelDriver.clip_guidance_preset)"><code class="language-python">	*Optional[str]*	If provided, requests a specific clip guidance preset to be used in the diffusion process.

continues on next page

Table 65 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.BedrockStableDiffusionImageGenerationModelDriver.sampler" title="sampler = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.BedrockStableDiffusionImageGenerationModelDriver.sampler)"><code class="language-python">	*Optional[str]*	If provided, requests a specific sampler to be used in the diffusion process.
<a href="./#griptape.drivers.BedrockStableDiffusionImageGenerationModelDriver.steps" title="steps = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.BedrockStableDiffusionImageGenerationModelDriver.steps)"><code class="language-python">	*Optional[int]*	If provided, specifies the number of diffusion steps to use in the image generation.

continues on next page

Table 65 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.BedrockStableDiffusionImageGenerationModelDriver.start_schedule" title="start_schedule = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.BedrockStableDiffusionImageGenerationModelDriver.start_schedule)"><code class="language-python">	*Optional[float]*	If provided, specifies the start_schedule parameter used to determine the influence of the input image in image-to-image generation.

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
@define
class BedrockStableDiffusionImageGenerationModelDriver(BaseImageGenerationModelDriver):
    """Image generation model driver for Stable Diffusion models on Amazon Bedrock.

    For more information on all supported parameters, see the Stable Diffusion documentation:
        https://platform.stability.ai/docs/api-reference#tag/v1generation

    Attributes:
        cfg_scale: Specifies how strictly image generation follows the provided prompt. Defaults to 7.
        mask_source: Specifies mask image configuration for image-to-image generations. Defaults to "MASK_IMAGE_BLACK".
        style_preset: If provided, specifies a specific image generation style preset.
        clip_guidance_preset: If provided, requests a specific clip guidance preset to be used in the diffusion process.
        sampler: If provided, requests a specific sampler to be used in the diffusion process.
        steps: If provided, specifies the number of diffusion steps to use in the image generation.
        start_schedule: If provided, specifies the start_schedule parameter used to determine the influence of the input
            image in image-to-image generation.
    """

    cfg_scale: int = field(default=7, kw_only=True, metadata={"serializable": True})
    style_preset: Optional[str] = field(default=None, kw_only=True, metadata={
```

(continues on next page)

(continued from previous page)

```
↪"serializable": True})
    clip_guidance_preset: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    sampler: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    steps: Optional[int] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    start_schedule: Optional[float] = field(default=None, kw_only=True, metadata={
↪"serializable": True})

    def text_to_image_request_parameters(
        self,
        prompts: list[str],
        image_width: int,
        image_height: int,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        return self._request_parameters(
            prompts,
            width=image_width,
            height=image_height,
            negative_prompts=negative_prompts,
            seed=seed,
        )

    def image_variation_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        return self._request_parameters(prompts, image=image, negative_
↪prompts=negative_prompts, seed=seed)

    def image_inpainting_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        return self._request_parameters(
            prompts,
            image=image,
            mask=mask,
            mask_source="MASK_IMAGE_BLACK",
            negative_prompts=negative_prompts,
            seed=seed,
        )
```

(continues on next page)

(continued from previous page)

```
    def image_outpainting_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        return self._request_parameters(
            prompts,
            image=image,
            mask=mask,
            mask_source="MASK_IMAGE_WHITE",
            negative_prompts=negative_prompts,
            seed=seed,
        )

    def _request_parameters(
        self,
        prompts: list[str],
        width: Optional[int] = None,
        height: Optional[int] = None,
        image: Optional[ImageArtifact] = None,
        mask: Optional[ImageArtifact] = None,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
        mask_source: Optional[str] = None,
    ) -> dict:
        if negative_prompts is None:
            negative_prompts = []

        text_prompts = [{"text": prompt, "weight": 1.0} for prompt in prompts]
        text_prompts += [{"text": negative_prompt, "weight": -1.0} for negative_
↪prompt in negative_prompts]

        request = {
            "text_prompts": text_prompts,
            "cfg_scale": self.cfg_scale,
            "style_preset": self.style_preset,
            "clip_guidance_preset": self.clip_guidance_preset,
            "sampler": self.sampler,
            "steps": self.steps,
            "seed": seed,
            "start_schedule": self.start_schedule,
        }

        if image is not None:
            request["init_image"] = image.base64
            request["width"] = image.width
            request["height"] = image.height
        else:
```

(continues on next page)

(continued from previous page)

```
            request["width"] = width
            request["height"] = height

        if mask is not None:
            if not mask_source:
                raise ValueError("mask_source must be provided when mask is provided")

            request["mask_source"] = mask_source
            request["mask_image"] = mask.base64

        return {k: v for k, v in request.items() if v is not None}

    def get_generated_image(self, response: dict) -> bytes:
        image_response = response["artifacts"][0]

        # finishReason may be SUCCESS, CONTENT_FILTERED, or ERROR.
        if image_response.get("finishReason") == "ERROR":
            raise Exception(f"Image generation failed: {image_response.
↪get(finishReason)}")
        if image_response.get("finishReason") == "CONTENT_FILTERED":
            logging.warning("Image generation triggered content filter and may be␣
↪blurred")

        return base64.decodebytes(bytes(image_response.get("base64"), "utf-8"))
```

- `cfg_scale = field(default=7, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `clip_guidance_preset = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `sampler = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `start_schedule = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `steps = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `style_preset = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_request_parameters(prompts, width=None, height=None, image=None, mask=None, negative_prompts=None, seed=None, mask_source=None)

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
def _request_parameters(
    self,
    prompts: list[str],
    width: Optional[int] = None,
    height: Optional[int] = None,
    image: Optional[ImageArtifact] = None,
```

(continues on next page)

(continued from previous page)

```
    mask: Optional[ImageArtifact] = None,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
    mask_source: Optional[str] = None,
) -> dict:
    if negative_prompts is None:
        negative_prompts = []

    text_prompts = [{"text": prompt, "weight": 1.0} for prompt in prompts]
    text_prompts += [{"text": negative_prompt, "weight": -1.0} for negative_prompt in␣
↪negative_prompts]

    request = {
        "text_prompts": text_prompts,
        "cfg_scale": self.cfg_scale,
        "style_preset": self.style_preset,
        "clip_guidance_preset": self.clip_guidance_preset,
        "sampler": self.sampler,
        "steps": self.steps,
        "seed": seed,
        "start_schedule": self.start_schedule,
    }

    if image is not None:
        request["init_image"] = image.base64
        request["width"] = image.width
        request["height"] = image.height
    else:
        request["width"] = width
        request["height"] = height

    if mask is not None:
        if not mask_source:
            raise ValueError("mask_source must be provided when mask is provided")

        request["mask_source"] = mask_source
        request["mask_image"] = mask.base64

    return {k: v for k, v in request.items() if v is not None}
```

get_generated_image(response)

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
def get_generated_image(self, response: dict) -> bytes:
    image_response = response["artifacts"][0]

    # finishReason may be SUCCESS, CONTENT_FILTERED, or ERROR.
    if image_response.get("finishReason") == "ERROR":
        raise Exception(f"Image generation failed: {image_response.get(finishReason)}
↪")
```

(continues on next page)

(continued from previous page)

```
    if image_response.get("finishReason") == "CONTENT_FILTERED":
        logging.warning("Image generation triggered content filter and may be blurred
↪")

    return base64.decodebytes(bytes(image_response.get("base64"), "utf-8"))
```

image_inpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
def image_inpainting_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    return self._request_parameters(
        prompts,
        image=image,
        mask=mask,
        mask_source="MASK_IMAGE_BLACK",
        negative_prompts=negative_prompts,
        seed=seed,
    )
```

image_outpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
def image_outpainting_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    return self._request_parameters(
        prompts,
        image=image,
        mask=mask,
        mask_source="MASK_IMAGE_WHITE",
        negative_prompts=negative_prompts,
        seed=seed,
    )
```

image_variation_request_parameters(prompts, image, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
def image_variation_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    return self._request_parameters(prompts, image=image, negative_prompts=negative_
↪prompts, seed=seed)
```

text_to_image_request_parameters(prompts, image_width, image_height, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
def text_to_image_request_parameters(
    self,
    prompts: list[str],
    image_width: int,
    image_height: int,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    return self._request_parameters(
        prompts,
        width=image_width,
        height=image_height,
        negative_prompts=negative_prompts,
        seed=seed,
    )
```

BedrockTitanImageGenerationModelDriver

Bases: *BaseImageGenerationModelDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.BedrockTitanImageGenerationModelDriver.quality" title="quality = field(default='standard', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.BedrockTitanImageGenerationModelDriver.quality)"><code class="language-python">	*str*	The quality of the generated image, defaults to standard.
<a href="./#griptape.drivers.BedrockTitanImageGenerationModelDriver.cfg_scale" title="cfg_scale = field(default=7, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.BedrockTitanImageGenerationModelDriver.cfg_scale)"><co class="language-python">	*int*	Specifies how strictly image generation follows the provided prompt. Defaults to 7, (1.0 to 10.0].
<a href="./#griptape.drivers.BedrockTitanImageGenerationModelDriver.outpainting_mode" title="outpainting_mode = field(default='PRECISE', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.BedrockTitanImageGenerationModelDriver.outpainting_mode)"><code class="language-python">	*str*	Specifies the outpainting mode, defaults to PRECISE.

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
@define
```

(continues on next page)

(continued from previous page)

```
class BedrockTitanImageGenerationModelDriver(BaseImageGenerationModelDriver):
    """Image Generation Model Driver for Amazon Bedrock Titan Image Generator.

    Attributes:
        quality: The quality of the generated image, defaults to standard.
        cfg_scale: Specifies how strictly image generation follows the provided␣
↪prompt. Defaults to 7, (1.0 to 10.0].
        outpainting_mode: Specifies the outpainting mode, defaults to PRECISE.
    """

    quality: str = field(default="standard", kw_only=True, metadata={"serializable":␣
↪True})
    cfg_scale: int = field(default=7, kw_only=True, metadata={"serializable": True})
    outpainting_mode: str = field(default="PRECISE", kw_only=True, metadata={
↪"serializable": True})

    def text_to_image_request_parameters(
        self,
        prompts: list[str],
        image_width: int,
        image_height: int,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        prompt = ", ".join(prompts)

        request = {
            "taskType": "TEXT_IMAGE",
            "textToImageParams": {"text": prompt},
            "imageGenerationConfig": {
                "numberOfImages": 1,
                "quality": self.quality,
                "width": image_width,
                "height": image_height,
                "cfgScale": self.cfg_scale,
            },
        }

        if negative_prompts:
            request["textToImageParams"]["negativeText"] = ", ".join(negative_prompts)

        if seed:
            request["imageGenerationConfig"]["seed"] = seed

        return self._add_common_params(request, image_width, image_height, seed=seed)

    def image_variation_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
```

(continues on next page)

(continued from previous page)

```
    ) -> dict:
        prompt = ", ".join(prompts)

        request = {
            "taskType": "IMAGE_VARIATION",
            "imageVariationParams": {"text": prompt, "images": [image.base64]},
            "imageGenerationConfig": {
                "numberOfImages": 1,
                "quality": self.quality,
                "width": image.width,
                "height": image.height,
                "cfgScale": self.cfg_scale,
            },
        }

        if negative_prompts:
            request["imageVariationParams"]["negativeText"] = ", ".join(negative_
↪prompts)

        if seed:
            request["imageGenerationConfig"]["seed"] = seed

        return self._add_common_params(request, image.width, image.height, seed=seed)

    def image_inpainting_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        prompt = ", ".join(prompts)

        request = {
            "taskType": "INPAINTING",
            "inPaintingParams": {"text": prompt, "image": image.base64, "maskImage":␣
↪mask.base64},
        }

        if negative_prompts:
            request["inPaintingParams"]["negativeText"] = ", ".join(negative_prompts)

        return self._add_common_params(request, image.width, image.height, seed=seed)

    def image_outpainting_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
```

(continues on next page)

(continued from previous page)

```
    ) -> dict:
        prompt = ", ".join(prompts)

        request = {
            "taskType": "OUTPAINTING",
            "outPaintingParams": {
                "text": prompt,
                "image": image.base64,
                "maskImage": mask.base64,
                "outPaintingMode": self.outpainting_mode,
            },
        }

        if negative_prompts:
            request["outPaintingParams"]["negativeText"] = ", ".join(negative_prompts)

        return self._add_common_params(request, image.width, image.height, seed=seed)

    def get_generated_image(self, response: dict) -> bytes:
        b64_image_data = response["images"][0]

        return base64.decodebytes(bytes(b64_image_data, "utf-8"))

    def _add_common_params(self, request: dict[str, Any], width: int, height: int,␣
↪seed: Optional[int] = None) -> dict:
        request["imageGenerationConfig"] = {
            "numberOfImages": 1,
            "quality": self.quality,
            "width": width,
            "height": height,
            "cfgScale": self.cfg_scale,
        }

        if seed:
            request["imageGenerationConfig"]["seed"] = seed

        return request
```

- `cfg_scale = field(default=7, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `outpainting_mode = field(default='PRECISE', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `quality = field(default='standard', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_add_common_params(request, width, height, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
def _add_common_params(self, request: dict[str, Any], width: int, height: int, seed:␣
↪Optional[int] = None) -> dict:
```

(continues on next page)

(continued from previous page)

```
    request["imageGenerationConfig"] = {
        "numberOfImages": 1,
        "quality": self.quality,
        "width": width,
        "height": height,
        "cfgScale": self.cfg_scale,
    }

    if seed:
        request["imageGenerationConfig"]["seed"] = seed

    return request
```

get_generated_image(response)

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
def get_generated_image(self, response: dict) -> bytes:
    b64_image_data = response["images"][0]

    return base64.decodebytes(bytes(b64_image_data, "utf-8"))
```

image_inpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
def image_inpainting_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    prompt = ", ".join(prompts)

    request = {
        "taskType": "INPAINTING",
        "inPaintingParams": {"text": prompt, "image": image.base64, "maskImage": mask.
↪base64},
    }

    if negative_prompts:
        request["inPaintingParams"]["negativeText"] = ", ".join(negative_prompts)

    return self._add_common_params(request, image.width, image.height, seed=seed)
```

image_outpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
def image_outpainting_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    prompt = ", ".join(prompts)

    request = {
        "taskType": "OUTPAINTING",
        "outPaintingParams": {
            "text": prompt,
            "image": image.base64,
            "maskImage": mask.base64,
            "outPaintingMode": self.outpainting_mode,
        },
    }

    if negative_prompts:
        request["outPaintingParams"]["negativeText"] = ", ".join(negative_prompts)

    return self._add_common_params(request, image.width, image.height, seed=seed)
```

image_variation_request_parameters(prompts, image, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
def image_variation_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    prompt = ", ".join(prompts)

    request = {
        "taskType": "IMAGE_VARIATION",
        "imageVariationParams": {"text": prompt, "images": [image.base64]},
        "imageGenerationConfig": {
            "numberOfImages": 1,
            "quality": self.quality,
            "width": image.width,
            "height": image.height,
            "cfgScale": self.cfg_scale,
        },
    }
```

(continues on next page)

(continued from previous page)

```
    if negative_prompts:
        request["imageVariationParams"]["negativeText"] = ", ".join(negative_prompts)

    if seed:
        request["imageGenerationConfig"]["seed"] = seed

    return self._add_common_params(request, image.width, image.height, seed=seed)
```

text_to_image_request_parameters(prompts, image_width, image_height, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
def text_to_image_request_parameters(
    self,
    prompts: list[str],
    image_width: int,
    image_height: int,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    prompt = ", ".join(prompts)

    request = {
        "taskType": "TEXT_IMAGE",
        "textToImageParams": {"text": prompt},
        "imageGenerationConfig": {
            "numberOfImages": 1,
            "quality": self.quality,
            "width": image_width,
            "height": image_height,
            "cfgScale": self.cfg_scale,
        },
    }

    if negative_prompts:
        request["textToImageParams"]["negativeText"] = ", ".join(negative_prompts)

    if seed:
        request["imageGenerationConfig"]["seed"] = seed

    return self._add_common_params(request, image_width, image_height, seed=seed)
```

CohereEmbeddingDriver

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.CohereEmbeddingDriver.api_key" title="api_key = field(kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.CohereEmbeddingDriver.api_key)"><code class="language-python">	*str*	Cohere API key.
<a href="./#griptape.drivers.BaseEmbeddingDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.CohereEmbeddingDriver.model)"><code class="language-python">	*str*	Cohere model name.
<a href="./#griptape.drivers.CohereEmbeddingDriver.client" title="client() (griptape.drivers.CohereEmbeddingDriver.client)"><code class="language-python">	*Client*	Custom *cohere.Client*.

continues on next page

Table 67 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.CohereEmbeddingDriver.tokenizer" title="tokenizer = field(default=Factory(lambda self: CohereTokenizer(model=self.model, client=self.client), takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.drivers.CohereEmbeddingDriver.tokenizer)"><code class="language-python">	<a href="../tokenizers#griptape.tokenizers.CohereTokenizer" title="CohereTokenizer (griptape.tokenizers.CohereTokenizer)"><code class="language-python">CohereTokenizer</code></a	Custom *CohereTokenizer*.
<a href="./#griptape.drivers.CohereEmbeddingDriver.input_type" title="input_type = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.CohereEmbeddingDriver.input_type)"><code class="language-python">	*str*	Cohere embedding input type.

Source Code in griptape/drivers/embedding/cohere_embedding_driver.py

```
@define
class CohereEmbeddingDriver(BaseEmbeddingDriver):
    """Cohere Embedding Driver.

    Attributes:
        api_key: Cohere API key.
        model:  Cohere model name.
        client: Custom `cohere.Client`.
        tokenizer: Custom `CohereTokenizer`.
        input_type: Cohere embedding input type.
    """

    DEFAULT_MODEL = "models/embedding-001"

    api_key: str = field(kw_only=True, metadata={"serializable": False})
    input_type: str = field(kw_only=True, metadata={"serializable": True})
```

(continues on next page)

(continued from previous page)

```
    _client: Optional[Client] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})
    tokenizer: CohereTokenizer = field(
        default=Factory(lambda self: CohereTokenizer(model=self.model, client=self.
↪client), takes_self=True),
        kw_only=True,
    )

    @lazy_property()
    def client(self) -> Client:
        return import_optional_dependency("cohere").Client(self.api_key)

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        result = self.client.embed(texts=[chunk], model=self.model, input_type=self.
↪input_type)

        if isinstance(result.embeddings, list):
            return result.embeddings[0]
        raise ValueError("Non-float embeddings are not supported.")
```

- `DEFAULT_MODEL = 'models/embedding-001'` class-attribute instance-attribute
- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `input_type = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: CohereTokenizer(model=self.model, client=self.client), takes_self=True), kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/cohere_embedding_driver.py

```
@lazy_property()
def client(self) -> Client:
    return import_optional_dependency("cohere").Client(self.api_key)
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/cohere_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    result = self.client.embed(texts=[chunk], model=self.model, input_type=self.input_
↪type)

    if isinstance(result.embeddings, list):
        return result.embeddings[0]
    raise ValueError("Non-float embeddings are not supported.")
```

CoherePromptDriver

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.CoherePromptDriver.api_key" title="api_key = field(default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.CoherePromptDriver.api_key)"><code class="language-python">	*str*	Cohere API key.
<a href="./#griptape.drivers.CoherePromptDriver.model" title="model = field(metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.CoherePromptDriver.model)"><code class="language-python">	*str*	Cohere model name.
<a href="./#griptape.drivers.CoherePromptDriver.client" title="client() (griptape.drivers.CoherePromptDriver.client)"><code class="language-python">	*ClientV2*	Custom *cohere.Client*.

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
@define(kw_only=True)
class CoherePromptDriver(BasePromptDriver):
    """Cohere Prompt Driver.

    Attributes:
        api_key: Cohere API key.
        model:  Cohere model name.
        client: Custom `cohere.Client`.
    """
```

(continues on next page)

(continued from previous page)

```
    api_key: str = field(default=None, metadata={"serializable": False})
    model: str = field(metadata={"serializable": True})
    force_single_step: bool = field(default=False, kw_only=True, metadata={
↪"serializable": True})
    use_native_tools: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})
    _client: Optional[ClientV2] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: CohereTokenizer(model=self.model, client=self.
↪client), takes_self=True),
    )

    @lazy_property()
    def client(self) -> ClientV2:
        return import_optional_dependency("cohere").ClientV2(self.api_key, log_
↪warning_experimental_features=False)

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        params = self._base_params(prompt_stack)

        logger.debug(params)

        result: ChatResponse = self.client.chat(**params)
        logger.debug(result.model_dump())

        return Message(
            content=self.__to_prompt_stack_message_content(result.message),
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(
                input_tokens=result.usage.tokens.input_tokens if result.usage and␣
↪result.usage.tokens else 0,
                output_tokens=result.usage.tokens.output_tokens if result.usage and␣
↪result.usage.tokens else 0,
            ),
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        params = self._base_params(prompt_stack)
        logger.debug(params)
        result: Iterator[StreamedChatResponseV2] = self.client.chat_stream(**params)

        for event in result:
            if event.type == "stream-end":
                usage = event.response.meta.tokens

                yield DeltaMessage(
                    usage=DeltaMessage.Usage(input_tokens=usage.input_tokens, output_
↪tokens=usage.output_tokens),
```

(continues on next page)

(continued from previous page)

```
                )
            elif event.type in ("tool-plan-delta", "content-delta", "tool-call-start",
↪ "tool-call-delta"):
                yield DeltaMessage(content=self.__to_prompt_stack_delta_message_
↪content(event))

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        tool_results = []

        messages = self.__to_cohere_messages(prompt_stack.messages)

        params = {
            "model": self.model,
            "messages": messages,
            "temperature": self.temperature,
            "stop_sequences": self.tokenizer.stop_sequences,
            "max_tokens": self.max_tokens,
            **({"tool_results": tool_results} if tool_results else {}),
            **self.extra_params,
        }

        if prompt_stack.output_schema is not None and self.structured_output_strategy␣
↪== "native":
            params["response_format"] = {"type": "json_object", "schema": prompt_
↪stack.to_output_json_schema()}

        if prompt_stack.tools and self.use_native_tools:
            params["tools"] = self.__to_cohere_tools(prompt_stack.tools)

        return params

    def __to_cohere_messages(self, messages: list[Message]) -> list[dict]:
        cohere_messages = []

        for message in messages:
            # If the message only contains textual content we can send it as a single␣
↪content.
            if message.is_text():
                cohere_messages.append({"role": self.__to_cohere_role(message),
↪"content": message.to_text()})
            # Action results must be sent as separate messages.
            elif message.has_any_content_type(ActionResultMessageContent):
                cohere_messages.extend(
                    {
                        "role": self.__to_cohere_role(message, action_result),
                        "content": self.__to_cohere_message_content(action_result),
                        "tool_call_id": action_result.action.tag,
                    }
                    for action_result in message.get_content_
↪type(ActionResultMessageContent)
                )
```

(continues on next page)

(continued from previous page)

```
                if message.has_any_content_type(TextMessageContent):
                    cohere_messages.append({"role": self.__to_cohere_role(message),
↪"content": message.to_text()})
            else:
                cohere_message = {
                    "role": self.__to_cohere_role(message),
                    "content": [
                        self.__to_cohere_message_content(content)
                        for content in [
                            content for content in message.content if not␣
↪isinstance(content, ActionCallMessageContent)
                        ]
                    ],
                }

                # Action calls must be attached to the message, not sent as content.
                action_call_content = [
                    content for content in message.content if isinstance(content,␣
↪ActionCallMessageContent)
                ]
                if action_call_content:
                    cohere_message["tool_calls"] = [
                        self.__to_cohere_message_content(action_call) for action_call␣
↪in action_call_content
                    ]

                cohere_messages.append(cohere_message)

        return cohere_messages

    def __to_cohere_message_content(self, content: BaseMessageContent) -> str | dict␣
↪| list[dict]:
        if isinstance(content, ActionCallMessageContent):
            action = content.artifact.value

            return {
                "type": "function",
                "id": action.tag,
                "function": {
                    "name": action.to_native_tool_name(),
                    "arguments": json.dumps(action.input),
                },
            }
        if isinstance(content, ActionResultMessageContent):
            artifact = content.artifact

            if isinstance(artifact, ListArtifact):
                message_content = [{"type": "text", "text": artifact.to_text()} for␣
↪artifact in artifact.value]
            else:
                message_content = {"type": "text", "text": artifact.to_text()}
```

(continues on next page)

(continued from previous page)

```
            return message_content
        return {"type": "text", "text": content.artifact.to_text()}

    def __to_cohere_role(self, message: Message, message_content:␣
↪Optional[BaseMessageContent] = None) -> str:
        if message.is_system():
            return "system"
        if message.is_assistant():
            return "assistant"
        if isinstance(message_content, ActionResultMessageContent):
            return "tool"
        return "user"

    def __to_cohere_tools(self, tools: list[BaseTool]) -> list[dict]:
        return [
            {
                "function": {
                    "name": tool.to_native_tool_name(activity),
                    "description": tool.activity_description(activity),
                    "parameters": tool.to_activity_json_schema(activity, "Parameters␣
↪Schema"),
                },
                "type": "function",
            }
            for tool in tools
            for activity in tool.activities()
        ]

    def __to_prompt_stack_message_content(self, response: AssistantMessageResponse) ->
↪ list[BaseMessageContent]:
        content = []
        if response.content:
            content.extend([TextMessageContent(TextArtifact(content.text)) for␣
↪content in response.content])
        if response.tool_plan:
            content.append(TextMessageContent(TextArtifact(response.tool_plan)))
        if response.tool_calls is not None:
            content.extend(
                [
                    ActionCallMessageContent(
                        ActionArtifact(
                            ToolAction(
                                tag=tool_call.id,
                                name=ToolAction.from_native_tool_name(tool_call.
↪function.name)[0],
                                path=ToolAction.from_native_tool_name(tool_call.
↪function.name)[1],
                                input=json.loads(tool_call.function.arguments),
                            ),
                        ),
                    )
                    for tool_call in response.tool_calls
```

(continues on next page)

(continued from previous page)

```
                    if tool_call.id is not None
                    and tool_call.function is not None
                    and tool_call.function.name is not None
                    and tool_call.function.arguments is not None
                ],
            )

        return content

    def __to_prompt_stack_delta_message_content(self, event: Any) ->␣
↪BaseDeltaMessageContent:
        if event.type == "content-delta":
            return TextDeltaMessageContent(event.delta.message.content.text, index=0)
        if event.type == "tool-plan-delta":
            return TextDeltaMessageContent(event.delta.message["tool_plan"])
        if event.type == "tool-call-start":
            tool_call_delta = event.delta.message["tool_calls"]
            name, path = ToolAction.from_native_tool_name(tool_call_delta["function"][
↪"name"])

            return ActionCallDeltaMessageContent(tag=tool_call_delta["id"], name=name,
↪ path=path)
        if event.type == "tool-call-delta":
            tool_call_delta = event.delta.message["tool_calls"]["function"]

            return ActionCallDeltaMessageContent(partial_input=tool_call_delta[
↪"arguments"])
        raise ValueError(f"Unsupported event type: {event.type}")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `force_single_step = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: CohereTokenizer(model=self.model, client=self.client), takes_self=True))` class-attribute instance-attribute
- `use_native_tools = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__to_cohere_message_content(content)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def __to_cohere_message_content(self, content: BaseMessageContent) -> str | dict |␣
↪list[dict]:
    if isinstance(content, ActionCallMessageContent):
        action = content.artifact.value
```

(continues on next page)

(continued from previous page)

```
            return {
                "type": "function",
                "id": action.tag,
                "function": {
                    "name": action.to_native_tool_name(),
                    "arguments": json.dumps(action.input),
                },
            }
        if isinstance(content, ActionResultMessageContent):
            artifact = content.artifact

            if isinstance(artifact, ListArtifact):
                message_content = [{"type": "text", "text": artifact.to_text()} for␣
↪artifact in artifact.value]
            else:
                message_content = {"type": "text", "text": artifact.to_text()}

            return message_content
        return {"type": "text", "text": content.artifact.to_text()}
```

__to_cohere_messages(messages)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def __to_cohere_messages(self, messages: list[Message]) -> list[dict]:
    cohere_messages = []

    for message in messages:
        # If the message only contains textual content we can send it as a single␣
↪content.
        if message.is_text():
            cohere_messages.append({"role": self.__to_cohere_role(message), "content
↪": message.to_text()})
        # Action results must be sent as separate messages.
        elif message.has_any_content_type(ActionResultMessageContent):
            cohere_messages.extend(
                {
                    "role": self.__to_cohere_role(message, action_result),
                    "content": self.__to_cohere_message_content(action_result),
                    "tool_call_id": action_result.action.tag,
                }
                for action_result in message.get_content_
↪type(ActionResultMessageContent)
            )

            if message.has_any_content_type(TextMessageContent):
                cohere_messages.append({"role": self.__to_cohere_role(message),
↪"content": message.to_text()})
        else:
            cohere_message = {
                "role": self.__to_cohere_role(message),
                "content": [
```

(continues on next page)

(continued from previous page)

```
                    self.__to_cohere_message_content(content)
                    for content in [
                        content for content in message.content if not␣
↪isinstance(content, ActionCallMessageContent)
                    ]
                ],
            }

            # Action calls must be attached to the message, not sent as content.
            action_call_content = [
                content for content in message.content if isinstance(content,␣
↪ActionCallMessageContent)
            ]
            if action_call_content:
                cohere_message["tool_calls"] = [
                    self.__to_cohere_message_content(action_call) for action_call in␣
↪action_call_content
                ]

            cohere_messages.append(cohere_message)

    return cohere_messages
```

__to_cohere_role(message, message_content=None)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def __to_cohere_role(self, message: Message, message_content:␣
↪Optional[BaseMessageContent] = None) -> str:
    if message.is_system():
        return "system"
    if message.is_assistant():
        return "assistant"
    if isinstance(message_content, ActionResultMessageContent):
        return "tool"
    return "user"
```

__to_cohere_tools(tools)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def __to_cohere_tools(self, tools: list[BaseTool]) -> list[dict]:
    return [
        {
            "function": {
                "name": tool.to_native_tool_name(activity),
                "description": tool.activity_description(activity),
                "parameters": tool.to_activity_json_schema(activity, "Parameters␣
↪Schema"),
            },
            "type": "function",
        }
```

(continues on next page)

(continued from previous page)

```
        for tool in tools
        for activity in tool.activities()
    ]
```

__to_prompt_stack_delta_message_content(event)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def __to_prompt_stack_delta_message_content(self, event: Any) ->␣
↪BaseDeltaMessageContent:
    if event.type == "content-delta":
        return TextDeltaMessageContent(event.delta.message.content.text, index=0)
    if event.type == "tool-plan-delta":
        return TextDeltaMessageContent(event.delta.message["tool_plan"])
    if event.type == "tool-call-start":
        tool_call_delta = event.delta.message["tool_calls"]
        name, path = ToolAction.from_native_tool_name(tool_call_delta["function"][
↪"name"])

        return ActionCallDeltaMessageContent(tag=tool_call_delta["id"], name=name,␣
↪path=path)
    if event.type == "tool-call-delta":
        tool_call_delta = event.delta.message["tool_calls"]["function"]

        return ActionCallDeltaMessageContent(partial_input=tool_call_delta["arguments
↪"])
    raise ValueError(f"Unsupported event type: {event.type}")
```

__to_prompt_stack_message_content(response)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def __to_prompt_stack_message_content(self, response: AssistantMessageResponse) ->␣
↪list[BaseMessageContent]:
    content = []
    if response.content:
        content.extend([TextMessageContent(TextArtifact(content.text)) for content in␣
↪response.content])
    if response.tool_plan:
        content.append(TextMessageContent(TextArtifact(response.tool_plan)))
    if response.tool_calls is not None:
        content.extend(
            [
                ActionCallMessageContent(
                    ActionArtifact(
                        ToolAction(
                            tag=tool_call.id,
                            name=ToolAction.from_native_tool_name(tool_call.function.
↪name)[0],
                            path=ToolAction.from_native_tool_name(tool_call.function.
↪name)[1],
                            input=json.loads(tool_call.function.arguments),
```

(continues on next page)

(continued from previous page)

```
                        ),
                    ),
                )
                for tool_call in response.tool_calls
                if tool_call.id is not None
                and tool_call.function is not None
                and tool_call.function.name is not None
                and tool_call.function.arguments is not None
            ],
        )

    return content
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    tool_results = []

    messages = self.__to_cohere_messages(prompt_stack.messages)

    params = {
        "model": self.model,
        "messages": messages,
        "temperature": self.temperature,
        "stop_sequences": self.tokenizer.stop_sequences,
        "max_tokens": self.max_tokens,
        **({"tool_results": tool_results} if tool_results else {}),
        **self.extra_params,
    }

    if prompt_stack.output_schema is not None and self.structured_output_strategy ==
↪"native":
        params["response_format"] = {"type": "json_object", "schema": prompt_stack.to_
↪output_json_schema()}

    if prompt_stack.tools and self.use_native_tools:
        params["tools"] = self.__to_cohere_tools(prompt_stack.tools)

    return params
```

client()

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
@lazy_property()
def client(self) -> ClientV2:
    return import_optional_dependency("cohere").ClientV2(self.api_key, log_warning_
↪experimental_features=False)
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    params = self._base_params(prompt_stack)

    logger.debug(params)

    result: ChatResponse = self.client.chat(**params)
    logger.debug(result.model_dump())

    return Message(
        content=self.__to_prompt_stack_message_content(result.message),
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(
            input_tokens=result.usage.tokens.input_tokens if result.usage and result.
↪usage.tokens else 0,
            output_tokens=result.usage.tokens.output_tokens if result.usage and␣
↪result.usage.tokens else 0,
        ),
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    params = self._base_params(prompt_stack)
    logger.debug(params)
    result: Iterator[StreamedChatResponseV2] = self.client.chat_stream(**params)

    for event in result:
        if event.type == "stream-end":
            usage = event.response.meta.tokens

            yield DeltaMessage(
                usage=DeltaMessage.Usage(input_tokens=usage.input_tokens, output_
↪tokens=usage.output_tokens),
            )
        elif event.type in ("tool-plan-delta", "content-delta", "tool-call-start",
↪"tool-call-delta"):
            yield DeltaMessage(content=self.__to_prompt_stack_delta_message_
↪content(event))
```

CohereRerankDriver

Bases: *BaseRerankDriver* “)

Source Code in griptape/drivers/rerank/cohere_rerank_driver.py

```
@define(kw_only=True)
class CohereRerankDriver(BaseRerankDriver):
    model: str = field(default="rerank-english-v3.0", metadata={"serializable": True})
    top_n: Optional[int] = field(default=None)

    api_key: str = field(metadata={"serializable": True})
    client: Client = field(
        default=Factory(lambda self: import_optional_dependency("cohere").Client(self.
↪api_key), takes_self=True),
    )

    def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]:
        # Cohere errors out if passed "empty" documents or no documents at all
        artifacts_dict = {str(hash(a.to_text())): a for a in artifacts if a}

        if artifacts_dict:
            response = self.client.rerank(
                model=self.model,
                query=query,
                documents=[a.to_text() for a in artifacts_dict.values()],
                return_documents=True,
                top_n=self.top_n,
            )
            return [artifacts_dict[str(hash(r.document.text))] for r in response.
↪results if r.document is not None]
        return []
```

- `api_key = field(metadata={'serializable': True})` class-attribute instance-attribute
- `client = field(default=Factory(lambda self: import_optional_dependency('cohere').Client(self.api_key), takes_self=True))` class-attribute instance-attribute
- `model = field(default='rerank-english-v3.0', metadata={'serializable': True})` class-attribute instance-attribute
- `top_n = field(default=None)` class-attribute instance-attribute

run(query, artifacts)

Source Code in griptape/drivers/rerank/cohere_rerank_driver.py

```
def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]:
    # Cohere errors out if passed "empty" documents or no documents at all
    artifacts_dict = {str(hash(a.to_text())): a for a in artifacts if a}

    if artifacts_dict:
        response = self.client.rerank(
            model=self.model,
            query=query,
            documents=[a.to_text() for a in artifacts_dict.values()],
            return_documents=True,
            top_n=self.top_n,
        )
        return [artifacts_dict[str(hash(r.document.text))] for r in response.results␣
```

(continues on next page)

(continued from previous page)

```
↪if r.document is not None]
    return []
```

DatadogObservabilityDriver

Bases: *OpenTelemetryObservabilityDriver* “)

Source Code in griptape/drivers/observability/datadog_observability_driver.py

```
@define
class DatadogObservabilityDriver(OpenTelemetryObservabilityDriver):
    datadog_agent_endpoint: str = field(
        default=Factory(lambda: os.getenv("DD_AGENT_ENDPOINT", "http://localhost:4318
↪")), kw_only=True
    )
    span_processor: SpanProcessor = field(
        default=Factory(
            lambda self: import_optional_dependency("opentelemetry.sdk.trace.export").
↪BatchSpanProcessor(
                import_optional_dependency("opentelemetry.exporter.otlp.proto.http.
↪trace_exporter").OTLPSpanExporter(
                    endpoint=f"{self.datadog_agent_endpoint}/v1/traces"
                )
            ),
            takes_self=True,
        ),
        kw_only=True,
    )
```

- `datadog_agent_endpoint = field(default=Factory(lambda: os.getenv('DD_AGENT_ENDPOINT', 'http://localhost:4318')), kw_only=True)` class-attribute instance-attribute
- `span_processor = field(default=Factory(lambda self: import_optional_dependency('opentelemetry.sdk.trace.export').BatchSpanProcessor(import_optional_dependency('opentelemetry.exporter.otlp.proto.http.trace_exporter').OTLPSpanExporter(endpoint=f'{self.datadog_agent_endpoint}/v1/traces')), takes_self=True), kw_only=True)` class-attribute instance-attribute

DuckDuckGoWebSearchDriver

Bases: *BaseWebSearchDriver* “)

Source Code in griptape/drivers/web_search/duck_duck_go_web_search_driver.py

```
@define
class DuckDuckGoWebSearchDriver(BaseWebSearchDriver):
    language: str = field(default="en", kw_only=True)
    country: str = field(default="us", kw_only=True)
    _client: Optional[DDGS] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
```

(continues on next page)

(continued from previous page)

```
    def client(self) -> DDGS:
        return import_optional_dependency("duckduckgo_search").DDGS()

    def search(self, query: str, **kwargs) -> ListArtifact:
        try:
            results = self.client.text(
                query, region=f"{self.language}-{self.country}", max_results=self.
↪results_count, **kwargs
            )
            return ListArtifact(
                [
                    TextArtifact(
                        json.dumps({"title": result["title"], "url": result["href"],
↪"description": result["body"]}),
                    )
                    for result in results
                ],
            )
        except Exception as e:
            raise Exception(f"Error searching {query} with DuckDuckGo: {e}") from e
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `country = field(default='us', kw_only=True)` class-attribute instance-attribute
- `language = field(default='en', kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/web_search/duck_duck_go_web_search_driver.py

```
@lazy_property()
def client(self) -> DDGS:
    return import_optional_dependency("duckduckgo_search").DDGS()
```

search(query, **kwargs)

Source Code in griptape/drivers/web_search/duck_duck_go_web_search_driver.py

```
def search(self, query: str, **kwargs) -> ListArtifact:
    try:
        results = self.client.text(
            query, region=f"{self.language}-{self.country}", max_results=self.results_
↪count, **kwargs
        )
        return ListArtifact(
            [
                TextArtifact(
                    json.dumps({"title": result["title"], "url": result["href"],
↪"description": result["body"]}),
                )
                for result in results
```

(continues on next page)

(continued from previous page)

```
            ],
        )
    except Exception as e:
        raise Exception(f"Error searching {query} with DuckDuckGo: {e}") from e
```

DummyAudioTranscriptionDriver

Bases: *BaseAudioTranscriptionDriver* “)

Source Code in griptape/drivers/audio_transcription/dummy_audio_transcription_driver.py

```
@define
class DummyAudioTranscriptionDriver(BaseAudioTranscriptionDriver):
    model: str = field(init=False)

    def try_run(self, audio: AudioArtifact, prompts: Optional[list] = None) ->␣
↪TextArtifact:
        raise DummyError(__class__.__name__, "try_transcription")
```

- `model = field(init=False)` class-attribute instance-attribute

try_run(audio, prompts=None)

Source Code in griptape/drivers/audio_transcription/dummy_audio_transcription_driver.py

```
def try_run(self, audio: AudioArtifact, prompts: Optional[list] = None) ->␣
↪TextArtifact:
    raise DummyError(__class__.__name__, "try_transcription")
```

DummyEmbeddingDriver

Bases: *BaseEmbeddingDriver* “)

Source Code in griptape/drivers/embedding/dummy_embedding_driver.py

```
@define
class DummyEmbeddingDriver(BaseEmbeddingDriver):
    model: None = field(init=False, default=None, kw_only=True)

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        raise DummyError(__class__.__name__, "try_embed_chunk")
```

- `model = field(init=False, default=None, kw_only=True)` class-attribute instance-attribute

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/dummy_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    raise DummyError(__class__.__name__, "try_embed_chunk")
```

DummyImageGenerationDriver

Bases: *BaseImageGenerationDriver* “)

Source Code in griptape/drivers/image_generation/dummy_image_generation_driver.py

```
@define
class DummyImageGenerationDriver(BaseImageGenerationDriver):
    model: None = field(init=False, default=None, kw_only=True)

    def try_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact:
        raise DummyError(__class__.__name__, "try_text_to_image")

    def try_image_variation(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise DummyError(__class__.__name__, "try_image_variation")

    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise DummyError(__class__.__name__, "try_image_inpainting")

    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise DummyError(__class__.__name__, "try_image_outpainting")
```

- `model = field(init=False, default=None, kw_only=True)` class-attribute instance-attribute

try_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/dummy_image_generation_driver.py

```
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise DummyError(__class__.__name__, "try_image_inpainting")
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/dummy_image_generation_driver.py

```
def try_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise DummyError(__class__.__name__, "try_image_outpainting")
```

try_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/dummy_image_generation_driver.py

```
def try_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise DummyError(__class__.__name__, "try_image_variation")
```

try_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/dummy_image_generation_driver.py

```
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    raise DummyError(__class__.__name__, "try_text_to_image")
```

DummyPromptDriver

Bases: *BasePromptDriver* “)

Source Code in griptape/drivers/prompt/dummy_prompt_driver.py

```
@define
class DummyPromptDriver(BasePromptDriver):
    model: None = field(init=False, default=None, kw_only=True)
    tokenizer: DummyTokenizer = field(default=Factory(lambda: DummyTokenizer()), kw_
↪only=True)

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        raise DummyError(__class__.__name__, "try_run")

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        raise DummyError(__class__.__name__, "try_stream")
```

- `model = field(init=False, default=None, kw_only=True)` class-attribute instance-attribute

- `tokenizer = field(default=Factory(lambda: DummyTokenizer()), kw_only=True)` class-attribute instance-attribute

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/dummy_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    raise DummyError(__class__.__name__, "try_run")
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/dummy_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    raise DummyError(__class__.__name__, "try_stream")
```

DummyTextToSpeechDriver

Bases: *BaseTextToSpeechDriver* “)

Source Code in griptape/drivers/text_to_speech/dummy_text_to_speech_driver.py

```
@define
class DummyTextToSpeechDriver(BaseTextToSpeechDriver):
    model: None = field(init=False, default=None, kw_only=True)

    def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
        raise DummyError(__class__.__name__, "try_text_to_audio")
```

- `model = field(init=False, default=None, kw_only=True)` class-attribute instance-attribute

try_text_to_audio(prompts)

Source Code in griptape/drivers/text_to_speech/dummy_text_to_speech_driver.py

```
def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
    raise DummyError(__class__.__name__, "try_text_to_audio")
```

DummyVectorStoreDriver

Bases: *BaseVectorStoreDriver* “)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
@define()
class DummyVectorStoreDriver(BaseVectorStoreDriver):
    embedding_driver: BaseEmbeddingDriver = field(
        kw_only=True,
        default=Factory(lambda: DummyEmbeddingDriver()),
        metadata={"serializable": True},
    )
```

(continues on next page)

(continued from previous page)

```
    def delete_vector(self, vector_id: str) -> None:
        raise DummyError(__class__.__name__, "delete_vector")

    def upsert_vector(
        self,
        vector: list[float],
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        raise DummyError(__class__.__name__, "upsert_vector")

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        raise DummyError(__class__.__name__, "load_entry")

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        raise DummyError(__class__.__name__, "load_entries")

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        raise DummyError(__class__.__name__, "query_vector")

    def query(
        self,
        query: str | TextArtifact | ImageArtifact,
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        raise DummyError(__class__.__name__, "query")
```

- `embedding_driver = field(kw_only=True, default=Factory(lambda: DummyEmbeddingDriver()), metadata={'serializable': True})` class-attribute instance-attribute

delete_vector(vector_id)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> None:
    raise DummyError(__class__.__name__, "delete_vector")
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    raise DummyError(__class__.__name__, "load_entries")
```

load_entry(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    raise DummyError(__class__.__name__, "load_entry")
```

query(query, *, count=None, namespace=None, include_vectors=False, **kwargs)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
def query(
    self,
    query: str | TextArtifact | ImageArtifact,
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    raise DummyError(__class__.__name__, "query")
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    raise DummyError(__class__.__name__, "query_vector")
```

upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    vector_id: Optional[str] = None,
```

(continues on next page)

(continued from previous page)

```
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    raise DummyError(__class__.__name__, "upsert_vector")
```

ElevenLabsTextToSpeechDriver

Bases: *BaseTextToSpeechDriver* “)

Source Code in griptape/drivers/text_to_speech/elevenlabs_text_to_speech_driver.py

```
@define
class ElevenLabsTextToSpeechDriver(BaseTextToSpeechDriver):
    api_key: str = field(kw_only=True, metadata={"serializable": True})
    voice: str = field(kw_only=True, metadata={"serializable": True})
    output_format: str = field(default="mp3_44100_128", kw_only=True, metadata={
↪"serializable": True})
    _client: Optional[ElevenLabs] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> ElevenLabs:
        return import_optional_dependency("elevenlabs.client").ElevenLabs(api_
↪key=self.api_key)

    def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
        audio = self.client.generate(
            text=". ".join(prompts),
            voice=self.voice,
            model=self.model,
            output_format=self.output_format,
        )

        content = b""
        for chunk in audio:
            content += chunk

        # All ElevenLabs audio format strings have the following structure:
        # {format}_{sample_rate}_{bitrate}
        artifact_format = self.output_format.split("_")[0]

        return AudioArtifact(value=content, format=artifact_format)
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output_format = field(default='mp3_44100_128', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `voice = field(kw_only=True, metadata={'serializable': True})` class-attribute

instance-attribute

client()

Source Code in griptape/drivers/text_to_speech/elevenlabs_text_to_speech_driver.py

```
@lazy_property()
def client(self) -> ElevenLabs:
    return import_optional_dependency("elevenlabs.client").ElevenLabs(api_key=self.
↪api_key)
```

try_text_to_audio(prompts)

Source Code in griptape/drivers/text_to_speech/elevenlabs_text_to_speech_driver.py

```
def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
    audio = self.client.generate(
        text=". ".join(prompts),
        voice=self.voice,
        model=self.model,
        output_format=self.output_format,
    )

    content = b""
    for chunk in audio:
        content += chunk

    # All ElevenLabs audio format strings have the following structure:
    # {format}_{sample_rate}_{bitrate}
    artifact_format = self.output_format.split("_")[0]

    return AudioArtifact(value=content, format=artifact_format)
```

ExaWebSearchDriver

Bases: *BaseWebSearchDriver* “)

Source Code in griptape/drivers/web_search/exa_web_search_driver.py

```
@define
class ExaWebSearchDriver(BaseWebSearchDriver):
    api_key: str = field(kw_only=True, default=None)
    highlights: bool = field(default=False, kw_only=True)
    use_autoprompt: bool = field(default=False, kw_only=True)
    params: dict[str, Any] = field(factory=dict, kw_only=True, metadata={"serializable
↪": True})
    _client: Optional[Exa] = field(default=None, kw_only=True, alias="client")

    @lazy_property()
    def client(self) -> Exa:
        return import_optional_dependency("exa_py").Exa(api_key=self.api_key)

    def search(self, query: str, **kwargs) -> ListArtifact[JsonArtifact]:
        response = self.client.search_and_contents(  # pyright:␣
```

(continues on next page)

(continued from previous page)

```
↪ignore[reportCallIssue]
            highlights=self.highlights,  # pyright: ignore[reportArgumentType]
            use_autoprompt=self.use_autoprompt,
            query=query,
            num_results=self.results_count,
            text=True,
            **self.params,
            **kwargs,
        )
        results = [
            {"title": result.title, "url": result.url, "highlights": result.
↪highlights, "text": result.text}
            for result in response.results
        ]
        return ListArtifact([JsonArtifact(result) for result in results])
```

- `_client = field(default=None, kw_only=True, alias='client')` class-attribute instance-attribute
- `api_key = field(kw_only=True, default=None)` class-attribute instance-attribute
- `highlights = field(default=False, kw_only=True)` class-attribute instance-attribute
- `params = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `use_autoprompt = field(default=False, kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/web_search/exa_web_search_driver.py

```
@lazy_property()
def client(self) -> Exa:
    return import_optional_dependency("exa_py").Exa(api_key=self.api_key)
```

search(query, **kwargs)

Source Code in griptape/drivers/web_search/exa_web_search_driver.py

```
def search(self, query: str, **kwargs) -> ListArtifact[JsonArtifact]:
    response = self.client.search_and_contents(  # pyright: ignore[reportCallIssue]
        highlights=self.highlights,  # pyright: ignore[reportArgumentType]
        use_autoprompt=self.use_autoprompt,
        query=query,
        num_results=self.results_count,
        text=True,
        **self.params,
        **kwargs,
    )
    results = [
        {"title": result.title, "url": result.url, "highlights": result.highlights,
↪"text": result.text}
        for result in response.results
```

(continues on next page)

(continued from previous page)

```
    ]
    return ListArtifact([JsonArtifact(result) for result in results])
```

GoogleEmbeddingDriver

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.GoogleEmbeddingDriver.api_key" title="api_key = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.GoogleEmbedding-Driver.api_key)"><code class="language-python">	*Optional[str]*	Google API key.
<a href="./#griptape.drivers.GoogleEmbeddingDriver.model" title="model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.GoogleEmbedding-Driver.model)"><code class="language-python">	*str*	Google model name.

continues on next page

Table 69 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.GoogleEmbeddingDriver.task_type" title="task_type = field(default='retrieval_document', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.GoogleEmbeddingDriver.task_type)"><code class="language-python">	*str*	Embedding model task type (<https://ai.google.dev/tutorials/python_quickstart#use_embeddings>). Defaults to *retrieval_document*.
<a href="./#griptape.drivers.GoogleEmbeddingDriver.title" title="title = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.GoogleEmbeddingDriver.title)"><code class="language-python">	*Optional[str]*	Optional title for the content. Only works with *retrieval_document* task type.

Source Code in griptape/drivers/embedding/google_embedding_driver.py

11.5.7 Optional title for the content. Only works with `retrieval_document` task type.

- `DEFAULT_MODEL = 'models/embedding-001'` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `task_type = field(default='retrieval_document', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `title = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_params(chunk)

Source Code in griptape/drivers/embedding/google_embedding_driver.py

```
def _params(self, chunk: str) -> dict:
    return {"input": chunk, "model": self.model}
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/google_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    genai = import_optional_dependency("google.generativeai")
    genai.configure(api_key=self.api_key)

    result = genai.embed_content(model=self.model, content=chunk, task_type=self.task_
↪type, title=self.title)

    return result["embedding"]
```

GooglePromptDriver

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.GooglePromptDriver.api_key" title="api_key = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.GooglePromptDriver.api_key)"><code class="language-python">	*Optional[str]*	Google API key.
<a href="./#griptape.drivers.GooglePromptDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.GooglePromptDriver.model)"><code class="language-python">	*str*	Google model name.
<a href="./#griptape.drivers.GooglePromptDriver.client" title="client() (griptape.drivers.GooglePromptDriver.client)"><code class="language-python">	*GenerativeModel*	Custom *GenerativeModel* client.

continues on next page

Table 70 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.GooglePromptDriver.top_p" title="top_p = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.GooglePromptDriver.top_p)"><code class="language-python">	*Optional[float]*	Optional value for top_p.
<a href="./#griptape.drivers.GooglePromptDriver.top_k" title="top_k = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.GooglePromptDriver.top_k)"><code class="language-python">	*Optional[int]*	Optional value for top_k.

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
@define
class GooglePromptDriver(BasePromptDriver):
    """Google Prompt Driver.

    Attributes:
        api_key: Google API key.
        model: Google model name.
        client: Custom `GenerativeModel` client.
        top_p: Optional value for top_p.
        top_k: Optional value for top_k.
    """

    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    model: str = field(kw_only=True, metadata={"serializable": True})
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: GoogleTokenizer(api_key=self.api_key, model=self.
↪model), takes_self=True),
        kw_only=True,
    )
    top_p: Optional[float] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    top_k: Optional[int] = field(default=None, kw_only=True, metadata={"serializable
```

(continues on next page)

(continued from previous page)

```
↪": True})
    use_native_tools: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})
    structured_output_strategy: StructuredOutputStrategy = field(
        default="tool", kw_only=True, metadata={"serializable": True}
    )
    tool_choice: str = field(default="auto", kw_only=True, metadata={"serializable":␣
↪True})
    _client: Optional[GenerativeModel] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @structured_output_strategy.validator  # pyright:␣
↪ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
        if value == "native":
            raise ValueError(f"{__class__.__name__} does not support `{value}`␣
↪structured output strategy.")

        return value

    @lazy_property()
    def client(self) -> GenerativeModel:
        genai = import_optional_dependency("google.generativeai")
        genai.configure(api_key=self.api_key)

        return genai.GenerativeModel(self.model)

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        messages = self.__to_google_messages(prompt_stack)
        params = self._base_params(prompt_stack)
        logger.debug((messages, params["generation_config"].__dict__))
        response: GenerateContentResponse = self.client.generate_content(messages,␣
↪**params)
        logger.debug(response.to_dict())

        usage_metadata = response.usage_metadata

        return Message(
            content=[self.__to_prompt_stack_message_content(part) for part in␣
↪response.parts],
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(
                input_tokens=usage_metadata.prompt_token_count,
                output_tokens=usage_metadata.candidates_token_count,
            ),
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        messages = self.__to_google_messages(prompt_stack)
```

(continues on next page)

(continued from previous page)

```
        params = {**self._base_params(prompt_stack), "stream": True}
        logger.debug((messages, params))
        response: GenerateContentResponse = self.client.generate_content(
            messages,
            **params,
        )

        prompt_token_count = None
        for chunk in response:
            logger.debug(chunk.to_dict())
            usage_metadata = chunk.usage_metadata

            content = self.__to_prompt_stack_delta_message_content(chunk.parts[0]) if␣
↪chunk.parts else None
            # Only want to output the prompt token count once since it is static each␣
↪chunk
            if prompt_token_count is None:
                prompt_token_count = usage_metadata.prompt_token_count
                yield DeltaMessage(
                    content=content,
                    usage=DeltaMessage.Usage(
                        input_tokens=usage_metadata.prompt_token_count,
                        output_tokens=usage_metadata.candidates_token_count,
                    ),
                )
            else:
                yield DeltaMessage(
                    content=content,
                    usage=DeltaMessage.Usage(output_tokens=usage_metadata.candidates_
↪token_count),
                )

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        types = import_optional_dependency("google.generativeai.types")
        protos = import_optional_dependency("google.generativeai.protos")

        system_messages = prompt_stack.system_messages
        if system_messages:
            self.client._system_instruction = types.ContentDict(
                role="system",
                parts=[protos.Part(text=system_message.to_text()) for system_message␣
↪in system_messages],
            )

        params = {
            "generation_config": types.GenerationConfig(
                **{
                    # For some reason, providing stop sequences when streaming breaks␣
↪native functions
                    # https://github.com/google-gemini/generative-ai-python/issues/446
                    "stop_sequences": [] if self.stream and self.use_native_tools␣
↪else self.tokenizer.stop_sequences,
```

(continues on next page)

(continued from previous page)

```
                    "max_output_tokens": self.max_tokens,
                    "temperature": self.temperature,
                    "top_p": self.top_p,
                    "top_k": self.top_k,
                    **self.extra_params,
                },
            ),
        }

        if prompt_stack.tools and self.use_native_tools:
            params["tool_config"] = {"function_calling_config": {"mode": self.tool_
↪choice}}

            if prompt_stack.output_schema is not None and self.structured_output_
↪strategy == "tool":
                params["tool_config"]["function_calling_config"]["mode"] = "auto"

            params["tools"] = self.__to_google_tools(prompt_stack.tools)

        return params

    def __to_google_messages(self, prompt_stack: PromptStack) -> ContentsType:
        types = import_optional_dependency("google.generativeai.types")

        return [
            types.ContentDict(
                {
                    "role": self.__to_google_role(message),
                    "parts": [self.__to_google_message_content(content) for content␣
↪in message.content],
                },
            )
            for message in prompt_stack.messages
            if not message.is_system()
        ]

    def __to_google_role(self, message: Message) -> str:
        if message.is_assistant():
            return "model"
        return "user"

    def __to_google_tools(self, tools: list[BaseTool]) -> list[dict]:
        types = import_optional_dependency("google.generativeai.types")

        tool_declarations = []
        for tool in tools:
            for activity in tool.activities():
                schema = tool.to_activity_json_schema(activity, "Parameters Schema")

                if "values" in schema["properties"]:
                    schema = schema["properties"]["values"]
```

(continues on next page)

(continued from previous page)

```
                schema = remove_key_in_dict_recursively(schema, "additionalProperties
↪")
                tool_declaration = types.FunctionDeclaration(
                    name=tool.to_native_tool_name(activity),
                    description=tool.activity_description(activity),
                    **(
                        {
                            "parameters": {
                                "type": schema["type"],
                                "properties": schema["properties"],
                                "required": schema.get("required", []),
                            }
                        }
                        if schema.get("properties")
                        else {}
                    ),
                )

                tool_declarations.append(tool_declaration)

        return tool_declarations

    def __to_google_message_content(self, content: BaseMessageContent) -> ContentDict␣
↪| Part | str:
        types = import_optional_dependency("google.generativeai.types")
        protos = import_optional_dependency("google.generativeai.protos")

        if isinstance(content, TextMessageContent):
            return content.artifact.to_text()
        if isinstance(content, ImageMessageContent):
            if isinstance(content.artifact, ImageArtifact):
                return types.ContentDict(mime_type=content.artifact.mime_type,␣
↪data=content.artifact.value)
            # TODO: Google requires uploading to the files endpoint: https://ai.
↪google.dev/gemini-api/docs/image-understanding#upload-image
            # Can be worked around by using GenericMessageContent, similar to videos.
            raise ValueError(f"Unsupported image artifact type: {type(content.
↪artifact)}")
        if isinstance(content, ActionCallMessageContent):
            action = content.artifact.value

            return protos.Part(function_call=protos.FunctionCall(name=action.tag,␣
↪args=action.input))
        if isinstance(content, ActionResultMessageContent):
            artifact = content.artifact

            return protos.Part(
                function_response=protos.FunctionResponse(
                    name=content.action.to_native_tool_name(),
                    response=artifact.to_dict(),
                ),
            )
```

(continues on next page)

(continued from previous page)

```
        if isinstance(content, GenericMessageContent):
            return content.artifact.value
        raise ValueError(f"Unsupported prompt stack content type: {type(content)}")

    def __to_prompt_stack_message_content(self, content: Part) -> BaseMessageContent:
        json_format = import_optional_dependency("google.protobuf.json_format")

        if content.text:
            return TextMessageContent(TextArtifact(content.text))
        if content.function_call:
            function_call = content.function_call

            name, path = ToolAction.from_native_tool_name(function_call.name)

            args = json_format.MessageToDict(function_call._pb).get("args", {})
            return ActionCallMessageContent(
                artifact=ActionArtifact(value=ToolAction(tag=function_call.name,␣
↪name=name, path=path, input=args)),
            )
        raise ValueError(f"Unsupported message content type {content}")

    def __to_prompt_stack_delta_message_content(self, content: Part) ->␣
↪BaseDeltaMessageContent:
        json_format = import_optional_dependency("google.protobuf.json_format")

        if content.text:
            return TextDeltaMessageContent(content.text)
        if content.function_call:
            function_call = content.function_call

            name, path = ToolAction.from_native_tool_name(function_call.name)

            args = json_format.MessageToDict(function_call._pb).get("args", {})
            return ActionCallDeltaMessageContent(
                tag=function_call.name,
                name=name,
                path=path,
                partial_input=json.dumps(args),
            )
        raise ValueError(f"Unsupported message content type {content}")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='tool', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: GoogleTokenizer(api_key=self.api_key, model=self.model), takes_self=True), kw_only=True)` class-attribute

instance-attribute

- `tool_choice = field(default='auto', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `top_k = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `top_p = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `use_native_tools = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__to_google_message_content(content)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def __to_google_message_content(self, content: BaseMessageContent) -> ContentDict |␣
↪Part | str:
    types = import_optional_dependency("google.generativeai.types")
    protos = import_optional_dependency("google.generativeai.protos")

    if isinstance(content, TextMessageContent):
        return content.artifact.to_text()
    if isinstance(content, ImageMessageContent):
        if isinstance(content.artifact, ImageArtifact):
            return types.ContentDict(mime_type=content.artifact.mime_type,␣
↪data=content.artifact.value)
        # TODO: Google requires uploading to the files endpoint: https://ai.google.
↪dev/gemini-api/docs/image-understanding#upload-image
        # Can be worked around by using GenericMessageContent, similar to videos.
        raise ValueError(f"Unsupported image artifact type: {type(content.artifact)}")
    if isinstance(content, ActionCallMessageContent):
        action = content.artifact.value

        return protos.Part(function_call=protos.FunctionCall(name=action.tag,␣
↪args=action.input))
    if isinstance(content, ActionResultMessageContent):
        artifact = content.artifact

        return protos.Part(
            function_response=protos.FunctionResponse(
                name=content.action.to_native_tool_name(),
                response=artifact.to_dict(),
            ),
        )
    if isinstance(content, GenericMessageContent):
        return content.artifact.value
    raise ValueError(f"Unsupported prompt stack content type: {type(content)}")
```

__to_google_messages(prompt_stack)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def __to_google_messages(self, prompt_stack: PromptStack) -> ContentsType:
    types = import_optional_dependency("google.generativeai.types")

    return [
        types.ContentDict(
            {
                "role": self.__to_google_role(message),
                "parts": [self.__to_google_message_content(content) for content in␣
↪message.content],
            },
        )
        for message in prompt_stack.messages
        if not message.is_system()
    ]
```

__to_google_role(message)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def __to_google_role(self, message: Message) -> str:
    if message.is_assistant():
        return "model"
    return "user"
```

__to_google_tools(tools)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def __to_google_tools(self, tools: list[BaseTool]) -> list[dict]:
    types = import_optional_dependency("google.generativeai.types")

    tool_declarations = []
    for tool in tools:
        for activity in tool.activities():
            schema = tool.to_activity_json_schema(activity, "Parameters Schema")

            if "values" in schema["properties"]:
                schema = schema["properties"]["values"]

            schema = remove_key_in_dict_recursively(schema, "additionalProperties")
            tool_declaration = types.FunctionDeclaration(
                name=tool.to_native_tool_name(activity),
                description=tool.activity_description(activity),
                **(
                    {
                        "parameters": {
                            "type": schema["type"],
                            "properties": schema["properties"],
                            "required": schema.get("required", []),
                        }
```

(continues on next page)

(continued from previous page)

```
                    }
                    if schema.get("properties")
                    else {}
                ),
            )

            tool_declarations.append(tool_declaration)

    return tool_declarations
```

__to_prompt_stack_delta_message_content(content)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def __to_prompt_stack_delta_message_content(self, content: Part) ->␣
↪BaseDeltaMessageContent:
    json_format = import_optional_dependency("google.protobuf.json_format")

    if content.text:
        return TextDeltaMessageContent(content.text)
    if content.function_call:
        function_call = content.function_call

        name, path = ToolAction.from_native_tool_name(function_call.name)

        args = json_format.MessageToDict(function_call._pb).get("args", {})
        return ActionCallDeltaMessageContent(
            tag=function_call.name,
            name=name,
            path=path,
            partial_input=json.dumps(args),
        )
    raise ValueError(f"Unsupported message content type {content}")
```

__to_prompt_stack_message_content(content)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def __to_prompt_stack_message_content(self, content: Part) -> BaseMessageContent:
    json_format = import_optional_dependency("google.protobuf.json_format")

    if content.text:
        return TextMessageContent(TextArtifact(content.text))
    if content.function_call:
        function_call = content.function_call

        name, path = ToolAction.from_native_tool_name(function_call.name)

        args = json_format.MessageToDict(function_call._pb).get("args", {})
        return ActionCallMessageContent(
            artifact=ActionArtifact(value=ToolAction(tag=function_call.name,␣
↪name=name, path=path, input=args)),
```

(continues on next page)

(continued from previous page)

```
        )
    raise ValueError(f"Unsupported message content type {content}")
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    types = import_optional_dependency("google.generativeai.types")
    protos = import_optional_dependency("google.generativeai.protos")

    system_messages = prompt_stack.system_messages
    if system_messages:
        self.client._system_instruction = types.ContentDict(
            role="system",
            parts=[protos.Part(text=system_message.to_text()) for system_message in␣
↪system_messages],
        )

    params = {
        "generation_config": types.GenerationConfig(
            **{
                # For some reason, providing stop sequences when streaming breaks␣
↪native functions
                # https://github.com/google-gemini/generative-ai-python/issues/446
                "stop_sequences": [] if self.stream and self.use_native_tools else␣
↪self.tokenizer.stop_sequences,
                "max_output_tokens": self.max_tokens,
                "temperature": self.temperature,
                "top_p": self.top_p,
                "top_k": self.top_k,
                **self.extra_params,
            },
        ),
    }

    if prompt_stack.tools and self.use_native_tools:
        params["tool_config"] = {"function_calling_config": {"mode": self.tool_choice}
↪}

        if prompt_stack.output_schema is not None and self.structured_output_strategy␣
↪== "tool":
            params["tool_config"]["function_calling_config"]["mode"] = "auto"

        params["tools"] = self.__to_google_tools(prompt_stack.tools)

    return params
```

client()

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
@lazy_property()
def client(self) -> GenerativeModel:
    genai = import_optional_dependency("google.generativeai")
    genai.configure(api_key=self.api_key)

    return genai.GenerativeModel(self.model)
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    messages = self.__to_google_messages(prompt_stack)
    params = self._base_params(prompt_stack)
    logger.debug((messages, params["generation_config"].__dict__))
    response: GenerateContentResponse = self.client.generate_content(messages,␣
↪**params)
    logger.debug(response.to_dict())

    usage_metadata = response.usage_metadata

    return Message(
        content=[self.__to_prompt_stack_message_content(part) for part in response.
↪parts],
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(
            input_tokens=usage_metadata.prompt_token_count,
            output_tokens=usage_metadata.candidates_token_count,
        ),
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    messages = self.__to_google_messages(prompt_stack)
    params = {**self._base_params(prompt_stack), "stream": True}
    logger.debug((messages, params))
    response: GenerateContentResponse = self.client.generate_content(
        messages,
        **params,
    )

    prompt_token_count = None
    for chunk in response:
        logger.debug(chunk.to_dict())
        usage_metadata = chunk.usage_metadata
```

(continues on next page)

(continued from previous page)

```
        content = self.__to_prompt_stack_delta_message_content(chunk.parts[0]) if␣
↪chunk.parts else None
        # Only want to output the prompt token count once since it is static each␣
↪chunk
        if prompt_token_count is None:
            prompt_token_count = usage_metadata.prompt_token_count
            yield DeltaMessage(
                content=content,
                usage=DeltaMessage.Usage(
                    input_tokens=usage_metadata.prompt_token_count,
                    output_tokens=usage_metadata.candidates_token_count,
                ),
            )
        else:
            yield DeltaMessage(
                content=content,
                usage=DeltaMessage.Usage(output_tokens=usage_metadata.candidates_
↪token_count),
            )
```

validate*structured_output_strategy(*, value)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
@structured_output_strategy.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
    if value == "native":
        raise ValueError(f"{__class__.__name__} does not support `{value}` structured␣
↪output strategy.")

    return value
```

GoogleWebSearchDriver

Bases: *BaseWebSearchDriver* “)

Source Code in griptape/drivers/web_search/google_web_search_driver.py

```
@define
class GoogleWebSearchDriver(BaseWebSearchDriver):
    api_key: str = field(kw_only=True)
    search_id: str = field(kw_only=True)
    language: str = field(default="en", kw_only=True)
    country: str = field(default="us", kw_only=True)

    def search(self, query: str, **kwargs) -> ListArtifact:
        return ListArtifact([TextArtifact(json.dumps(result)) for result in self._
↪search_google(query, **kwargs)])

    def _search_google(self, query: str, **kwargs) -> list[dict]:
```

(continues on next page)

(continued from previous page)

```
        query_params = {
            "key": self.api_key,
            "cx": self.search_id,
            "q": query,
            "start": 0,
            "lr": f"lang_{self.language}",
            "num": self.results_count,
            "gl": self.country,
            **kwargs,
        }
        response = requests.get("https://www.googleapis.com/customsearch/v1",␣
↪params=query_params)

        if response.status_code == 200:
            data = response.json()

            return [{"url": r["link"], "title": r["title"], "description": r["snippet
↪"]} for r in data["items"]]

        raise Exception(
            f"Google Search API returned an error with status code "
            f"{response.status_code} and reason {response.reason}",
        )
```

- `api_key = field(kw_only=True)` class-attribute instance-attribute
- `country = field(default='us', kw_only=True)` class-attribute instance-attribute
- `language = field(default='en', kw_only=True)` class-attribute instance-attribute
- `search_id = field(kw_only=True)` class-attribute instance-attribute

_search_google(query, **kwargs)

Source Code in griptape/drivers/web_search/google_web_search_driver.py

```
def _search_google(self, query: str, **kwargs) -> list[dict]:
    query_params = {
        "key": self.api_key,
        "cx": self.search_id,
        "q": query,
        "start": 0,
        "lr": f"lang_{self.language}",
        "num": self.results_count,
        "gl": self.country,
        **kwargs,
    }
    response = requests.get("https://www.googleapis.com/customsearch/v1",␣
↪params=query_params)

    if response.status_code == 200:
        data = response.json()

        return [{"url": r["link"], "title": r["title"], "description": r["snippet"]}␣
```

(continues on next page)

(continued from previous page)

```
↪for r in data["items"]]

    raise Exception(
        f"Google Search API returned an error with status code "
        f"{response.status_code} and reason {response.reason}",
    )
```

search(query, **kwargs)

Source Code in griptape/drivers/web_search/google_web_search_driver.py

```
def search(self, query: str, **kwargs) -> ListArtifact:
    return ListArtifact([TextArtifact(json.dumps(result)) for result in self._search_
↪google(query, **kwargs)])
```

GriptapeCloudAssistantDriver

Bases: *BaseAssistantDriver* “)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
@define
class GriptapeCloudAssistantDriver(BaseAssistantDriver):
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]))
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        kw_only=True,
    )
    input: Optional[str] = field(default=None, kw_only=True)
    assistant_id: str = field(kw_only=True)
    thread_id: Optional[str] = field(default=None, kw_only=True)
    thread_alias: Optional[str] = field(default=None, kw_only=True)
    ruleset_ids: Optional[list[str]] = field(default=None, kw_only=True)
    additional_ruleset_ids: list[str] = field(factory=list, kw_only=True)
    knowledge_base_ids: Optional[list[str]] = field(default=None, kw_only=True)
    additional_knowledge_base_ids: list[str] = field(factory=list, kw_only=True)
    structure_ids: Optional[list[str]] = field(default=None, kw_only=True)
    additional_structure_ids: list[str] = field(factory=list, kw_only=True)
    tool_ids: Optional[list[str]] = field(default=None, kw_only=True)
    additional_tool_ids: list[str] = field(factory=list, kw_only=True)
    stream: bool = field(default=False, kw_only=True)
    poll_interval: int = field(default=1, kw_only=True)
    max_attempts: int = field(default=20, kw_only=True)
    auto_create_thread: bool = field(default=True, kw_only=True)

    def try_run(self, *args: BaseArtifact) -> TextArtifact:
        if self.thread_id is None and self.auto_create_thread:
```

(continues on next page)

(continued from previous page)

```
            self._create_or_find_thread(self.thread_alias)
        assistant_run_id = self._create_run(*args)
        run_result = self._get_run_result(assistant_run_id)

        run_result.meta.update(
            {"assistant_id": self.assistant_id, "assistant_run_id": assistant_run_id,
↪"thread_id": self.thread_id}
        )

        return run_result

    def _create_or_find_thread(self, thread_alias: Optional[str] = None) -> None:
        if thread_alias is None:
            self.thread_id = self._create_thread()
        else:
            thread = self._find_thread_by_alias(thread_alias)

            if thread is None:
                self.thread_id = self._create_thread(thread_alias)
            else:
                self.thread_id = thread["thread_id"]

    def _create_thread(self, thread_alias: Optional[str] = None) -> str:
        url = griptape_cloud_url(self.base_url, "api/threads")

        body = {"name": uuid.uuid4().hex}
        if thread_alias is not None:
            body["alias"] = thread_alias

        response = requests.post(url, json=body, headers=self.headers)
        response.raise_for_status()
        return response.json()["thread_id"]

    def _create_run(self, *args: BaseArtifact) -> str:
        url = griptape_cloud_url(self.base_url, f"api/assistants/{self.assistant_id}/
↪runs")

        response = requests.post(
            url,
            json={
                "args": [arg.value for arg in args],
                "stream": self.stream,
                "thread_id": self.thread_id,
                "input": self.input,
                **({"ruleset_ids": self.ruleset_ids} if self.ruleset_ids is not None␣
↪else {}),
                "additional_ruleset_ids": self.additional_ruleset_ids,
                **({"knowledge_base_ids": self.knowledge_base_ids} if self.knowledge_
↪base_ids is not None else {}),
                "additional_knowledge_base_ids": self.additional_knowledge_base_ids,
                **({"structure_ids": self.structure_ids} if self.structure_ids is not␣
↪None else {}),
```

(continues on next page)

(continued from previous page)

```
                "additional_structure_ids": self.additional_structure_ids,
                **({"tool_ids": self.tool_ids} if self.tool_ids is not None else {}),
                "additional_tool_ids": self.additional_tool_ids,
            },
            headers=self.headers,
        )
        response.raise_for_status()
        return response.json()["assistant_run_id"]

    def _get_run_result(self, assistant_run_id: str) -> TextArtifact:
        events = self._get_run_events(assistant_run_id)
        output = None

        for event in events:
            if event["origin"] == "ASSISTANT":
                event_payload = event["payload"]
                try:
                    EventBus.publish_event(BaseEvent.from_dict(event_payload))
                except ValueError as e:
                    logger.warning("Failed to deserialize event: %s", e)
                if event["type"] == "FinishStructureRunEvent":
                    output = TextArtifact.from_dict(event_payload["output_task_output
↪"])

        if output is None:
            raise ValueError("Output not found.")

        return output

    def _get_run_events(self, assistant_run_id: str) -> Iterator[dict]:
        url = griptape_cloud_url(self.base_url, f"api/assistant-runs/{assistant_run_
↪id}/events/stream")
        with requests.get(url, headers=self.headers, stream=True) as response:
            response.raise_for_status()
            for line in response.iter_lines():
                if line:
                    decoded_line = line.decode("utf-8")
                    if decoded_line.startswith("data:"):
                        yield json.loads(decoded_line.removeprefix("data:").strip())

    def _find_thread_by_alias(self, thread_alias: str) -> Optional[dict]:
        url = griptape_cloud_url(self.base_url, "api/threads")
        response = requests.get(url, params={"alias": thread_alias}, headers=self.
↪headers)
        response.raise_for_status()

        threads = response.json()["threads"]

        return next((thread for thread in threads if thread["alias"] == thread_alias),
↪ None)
```

- `additional_knowledge_base_ids = field(factory=list, kw_only=True)` class-attribute instance-attribute

- `additional_ruleset_ids = field(factory=list, kw_only=True)` class-attribute instance-attribute
- `additional_structure_ids = field(factory=list, kw_only=True)` class-attribute instance-attribute
- `additional_tool_ids = field(factory=list, kw_only=True)` class-attribute instance-attribute
- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']))` class-attribute instance-attribute
- `assistant_id = field(kw_only=True)` class-attribute instance-attribute
- `auto_create_thread = field(default=True, kw_only=True)` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')))` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `input = field(default=None, kw_only=True)` class-attribute instance-attribute
- `knowledge_base_ids = field(default=None, kw_only=True)` class-attribute instance-attribute
- `max_attempts = field(default=20, kw_only=True)` class-attribute instance-attribute
- `poll_interval = field(default=1, kw_only=True)` class-attribute instance-attribute
- `ruleset_ids = field(default=None, kw_only=True)` class-attribute instance-attribute
- `stream = field(default=False, kw_only=True)` class-attribute instance-attribute
- `structure_ids = field(default=None, kw_only=True)` class-attribute instance-attribute
- `thread_alias = field(default=None, kw_only=True)` class-attribute instance-attribute
- `thread_id = field(default=None, kw_only=True)` class-attribute instance-attribute
- `tool_ids = field(default=None, kw_only=True)` class-attribute instance-attribute

_create_or_find_thread(thread_alias=None)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def _create_or_find_thread(self, thread_alias: Optional[str] = None) -> None:
    if thread_alias is None:
        self.thread_id = self._create_thread()
    else:
        thread = self._find_thread_by_alias(thread_alias)

        if thread is None:
            self.thread_id = self._create_thread(thread_alias)
        else:
            self.thread_id = thread["thread_id"]
```

_create_run(*args)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def _create_run(self, *args: BaseArtifact) -> str:
    url = griptape_cloud_url(self.base_url, f"api/assistants/{self.assistant_id}/runs
↪")
```

(continues on next page)

(continued from previous page)

```
    response = requests.post(
        url,
        json={
            "args": [arg.value for arg in args],
            "stream": self.stream,
            "thread_id": self.thread_id,
            "input": self.input,
            **({"ruleset_ids": self.ruleset_ids} if self.ruleset_ids is not None else
↪{}),
            "additional_ruleset_ids": self.additional_ruleset_ids,
            **({"knowledge_base_ids": self.knowledge_base_ids} if self.knowledge_base_
↪ids is not None else {}),
            "additional_knowledge_base_ids": self.additional_knowledge_base_ids,
            **({"structure_ids": self.structure_ids} if self.structure_ids is not␣
↪None else {}),
            "additional_structure_ids": self.additional_structure_ids,
            **({"tool_ids": self.tool_ids} if self.tool_ids is not None else {}),
            "additional_tool_ids": self.additional_tool_ids,
        },
        headers=self.headers,
    )
    response.raise_for_status()
    return response.json()["assistant_run_id"]
```

_create_thread(thread_alias=None)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def _create_thread(self, thread_alias: Optional[str] = None) -> str:
    url = griptape_cloud_url(self.base_url, "api/threads")

    body = {"name": uuid.uuid4().hex}
    if thread_alias is not None:
        body["alias"] = thread_alias

    response = requests.post(url, json=body, headers=self.headers)
    response.raise_for_status()
    return response.json()["thread_id"]
```

_find_thread_by_alias(thread_alias)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def _find_thread_by_alias(self, thread_alias: str) -> Optional[dict]:
    url = griptape_cloud_url(self.base_url, "api/threads")
    response = requests.get(url, params={"alias": thread_alias}, headers=self.headers)
    response.raise_for_status()

    threads = response.json()["threads"]
```

(continues on next page)

(continued from previous page)

```
    return next((thread for thread in threads if thread["alias"] == thread_alias),␣
↪None)
```

_get_run_events(assistant_run_id)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def _get_run_events(self, assistant_run_id: str) -> Iterator[dict]:
    url = griptape_cloud_url(self.base_url, f"api/assistant-runs/{assistant_run_id}/
↪events/stream")
    with requests.get(url, headers=self.headers, stream=True) as response:
        response.raise_for_status()
        for line in response.iter_lines():
            if line:
                decoded_line = line.decode("utf-8")
                if decoded_line.startswith("data:"):
                    yield json.loads(decoded_line.removeprefix("data:").strip())
```

_get_run_result(assistant_run_id)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def _get_run_result(self, assistant_run_id: str) -> TextArtifact:
    events = self._get_run_events(assistant_run_id)
    output = None

    for event in events:
        if event["origin"] == "ASSISTANT":
            event_payload = event["payload"]
            try:
                EventBus.publish_event(BaseEvent.from_dict(event_payload))
            except ValueError as e:
                logger.warning("Failed to deserialize event: %s", e)
            if event["type"] == "FinishStructureRunEvent":
                output = TextArtifact.from_dict(event_payload["output_task_output"])

    if output is None:
        raise ValueError("Output not found.")

    return output
```

try_run(*args)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def try_run(self, *args: BaseArtifact) -> TextArtifact:
    if self.thread_id is None and self.auto_create_thread:
        self._create_or_find_thread(self.thread_alias)
    assistant_run_id = self._create_run(*args)
    run_result = self._get_run_result(assistant_run_id)

    run_result.meta.update(
```

(continues on next page)

(continued from previous page)

```
        {"assistant_id": self.assistant_id, "assistant_run_id": assistant_run_id,
↪"thread_id": self.thread_id}
    )

    return run_result
```

GriptapeCloudConversationMemoryDriver

Bases: *BaseConversationMemoryDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.GriptapeCloudConversationMemoryDriver.thread_id" title="thread_id = field(default=None, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.GriptapeCloudConversationMemoryDriver.thread_id)"><code class="language-python">	*Optional[str]*	The ID of the Thread to store the conversation memory in. If not provided, the driver will attempt to retrieve the ID from the environment variable *GT_CLOUD_THREAD_ID*.
<a href="./#griptape.drivers.GriptapeCloudConversationMemoryDriver.alias" title="alias = field(default=None, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.GriptapeCloudConversationMemoryDriver.alias)"><code class="language-python">	*Optional[str]*	The alias of the Thread to store the conversation memory in.

continues on next page

Table 71 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.GriptapeCloudConversationMemoryDriver.base_url" title="base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_ 'https://cloud.griptape.ai'))) class-attribute instance-attribute (griptape.drivers.GriptapeCloudConversationMemoryDriver.base_url)"><code class="language-python">	*str*	The base URL of the Gen AI Builder API. Defaults to the value of the environment variable *GT_CLOUD_BASE_URL* or *https://cloud.griptape.ai*.
<a href="./#griptape.drivers.GriptapeCloudConversationMemoryDriver.api_key" title="api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_K class-attribute instance-attribute (griptape.drivers.GriptapeCloudConversationMemoryDriver.api_key)"><code class="language-python">	*str*	The API key to use for authenticating with the Gen AI Builder API. If not provided, the driver will attempt to retrieve the API key from the environment variable *GT_CLOUD_API_KEY*.

Raises

Source Code in griptape/drivers/memory/conversation/griptape_cloud_conversation_memory_driver.py

```
@define(kw_only=True)
class GriptapeCloudConversationMemoryDriver(BaseConversationMemoryDriver):
    """A driver for storing conversation memory in the Gen AI Builder.

    Attributes:
        thread_id: The ID of the Thread to store the conversation memory in. If not␣
↪provided, the driver will attempt to
            retrieve the ID from the environment variable `GT_CLOUD_THREAD_ID`.
        alias: The alias of the Thread to store the conversation memory in.
        base_url: The base URL of the Gen AI Builder API. Defaults to the value of␣
↪the environment variable
            `GT_CLOUD_BASE_URL` or `https://cloud.griptape.ai`.
```

(continues on next page)

(continued from previous page)

```
        api_key: The API key to use for authenticating with the Gen AI Builder API.␣
↪If not provided, the driver will
            attempt to retrieve the API key from the environment variable `GT_CLOUD_
↪API_KEY`.

    Raises:
        ValueError: If `api_key` is not provided.
    """

    thread_id: Optional[str] = field(
        default=None,
        metadata={"serializable": True},
    )
    alias: Optional[str] = field(
        default=None,
        metadata={"serializable": True},
    )
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]))
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        init=False,
    )
    _thread: Optional[dict] = field(default=None, init=False)

    @api_key.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_api_key(self, _: Attribute, value: Optional[str]) -> str:
        if value is None:
            raise ValueError(f"{self.__class__.__name__} requires an API key")
        return value

    @property
    def thread(self) -> dict:
        """Try to get the Thread by ID, alias, or create a new one."""
        if self._thread is None:
            thread = None
            if self.thread_id is None:
                self.thread_id = os.getenv("GT_CLOUD_THREAD_ID")

            if self.thread_id is not None:
                res = self._call_api("get", f"/threads/{self.thread_id}", raise_for_
↪status=False)
                if res.status_code == 200:
                    thread = res.json()

            # use name as alias to get thread
            if thread is None and self.alias is not None:
                res = self._call_api("get", f"/threads?alias={self.alias}").json()
```

(continues on next page)

(continued from previous page)

```
                if res.get("threads"):
                    thread = res["threads"][0]
                    self.thread_id = thread.get("thread_id")

            # no thread by name or thread_id
            if thread is None:
                data = {"name": uuid.uuid4().hex} if self.alias is None else {"name":␣
↪self.alias, "alias": self.alias}
                thread = self._call_api("post", "/threads", data).json()
                self.thread_id = thread["thread_id"]
                self.alias = thread.get("alias")

            self._thread = thread

        return self._thread  # pyright: ignore[reportReturnType]

    def store(self, runs: list[Run], metadata: dict[str, Any]) -> None:
        # serialize the run artifacts to json strings
        messages = [
            dict_merge(
                {
                    "input": run.input.to_json(),
                    "output": run.output.to_json(),
                    "metadata": {"run_id": run.id},
                },
                run.meta,
            )
            for run in runs
        ]

        body = dict_merge(
            {
                "messages": messages,
            },
            metadata,
        )

        # patch the Thread with the new messages and metadata
        # all old Messages are replaced with the new ones
        thread_id = self.thread["thread_id"] if self.thread_id is None else self.
↪thread_id
        self._call_api("patch", f"/threads/{thread_id}", body)
        self._thread = None

    def load(self) -> tuple[list[Run], dict[str, Any]]:
        from griptape.memory.structure import Run

        thread_id = self.thread["thread_id"] if self.thread_id is None else self.
↪thread_id

        # get the Messages from the Thread
        messages_response = self._call_api("get", f"/threads/{thread_id}/messages").
```

(continues on next page)

(continued from previous page)

```
↪json()

        # retrieve the Thread to get the metadata
        thread_response = self._call_api("get", f"/threads/{thread_id}").json()

        runs = [
            Run(
                **({"id": message["metadata"].pop("run_id", None)} if "run_id" in␣
↪message.get("metadata") else {}),
                meta=message["metadata"],
                input=BaseArtifact.from_json(message["input"]),
                output=BaseArtifact.from_json(message["output"]),
            )
            for message in messages_response.get("messages", [])
        ]

        return runs, thread_response.get("metadata", {})

    def _get_url(self, path: str) -> str:
        path = path.lstrip("/")
        return griptape_cloud_url(self.base_url, f"api/{path}")

    def _call_api(
        self, method: str, path: str, json: Optional[dict] = None, *, raise_for_
↪status: bool = True
    ) -> requests.Response:
        res = requests.request(method, self._get_url(path), json=json, headers=self.
↪headers)
        if raise_for_status:
            res.raise_for_status()
        return res
```

- `_thread = field(default=None, init=False)` class-attribute instance-attribute
- `alias = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']))` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')))` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), init=False)` class-attribute instance-attribute
- `thread` property
- `thread_id = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

_call_api(method, path, json=None, *, raise_for_status=True)

Source Code in griptape/drivers/memory/conversation/griptape_cloud_conversation_memory_driver.py

```
def _call_api(
    self, method: str, path: str, json: Optional[dict] = None, *, raise_for_status:␣
↪bool = True
) -> requests.Response:
    res = requests.request(method, self._get_url(path), json=json, headers=self.
↪headers)
    if raise_for_status:
        res.raise_for_status()
    return res
```

_get_url(path)

Source Code in griptape/drivers/memory/conversation/griptape_cloud_conversation_memory_driver.py

```
def _get_url(self, path: str) -> str:
    path = path.lstrip("/")
    return griptape_cloud_url(self.base_url, f"api/{path}")
```

load()

Source Code in griptape/drivers/memory/conversation/griptape_cloud_conversation_memory_driver.py

```
def load(self) -> tuple[list[Run], dict[str, Any]]:
    from griptape.memory.structure import Run

    thread_id = self.thread["thread_id"] if self.thread_id is None else self.thread_id

    # get the Messages from the Thread
    messages_response = self._call_api("get", f"/threads/{thread_id}/messages").json()

    # retrieve the Thread to get the metadata
    thread_response = self._call_api("get", f"/threads/{thread_id}").json()

    runs = [
        Run(
            **({"id": message["metadata"].pop("run_id", None)} if "run_id" in message.
↪get("metadata") else {}),
            meta=message["metadata"],
            input=BaseArtifact.from_json(message["input"]),
            output=BaseArtifact.from_json(message["output"]),
        )
        for message in messages_response.get("messages", [])
    ]

    return runs, thread_response.get("metadata", {})
```

store(runs, metadata)

Source Code in griptape/drivers/memory/conversation/griptape_cloud_conversation_memory_driver.py

```
def store(self, runs: list[Run], metadata: dict[str, Any]) -> None:
    # serialize the run artifacts to json strings
    messages = [
        dict_merge(
            {
                "input": run.input.to_json(),
                "output": run.output.to_json(),
                "metadata": {"run_id": run.id},
            },
            run.meta,
        )
        for run in runs
    ]

    body = dict_merge(
        {
            "messages": messages,
        },
        metadata,
    )

    # patch the Thread with the new messages and metadata
    # all old Messages are replaced with the new ones
    thread_id = self.thread["thread_id"] if self.thread_id is None else self.thread_id
    self._call_api("patch", f"/threads/{thread_id}", body)
    self._thread = None
```

validate*api_key(*, value)

Source Code in griptape/drivers/memory/conversation/griptape_cloud_conversation_memory_driver.py

```
@api_key.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_api_key(self, _: Attribute, value: Optional[str]) -> str:
    if value is None:
        raise ValueError(f"{self.__class__.__name__} requires an API key")
    return value
```

GriptapeCloudEventListenerDriver

Bases: *BaseEventListenerDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.GriptapeCloudEventListenerDriver.base_url" title="base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_ 'https://cloud.griptape.ai')), kw_only=True) class-attribute instance-attribute (griptape.drivers.GriptapeCloudEventListenerDriver.base_url)"><code class="language-python">	*str*	The base URL of Gen AI Builder. Defaults to the GT_CLOUD_BASE_URL environment variable.
<a href="./#griptape.drivers.GriptapeCloudEventListenerDriver.api_key" title="api_key = field(default=Factory(lambda: os.getenv('GT_CLOUD_ kw_only=True) class-attribute instance-attribute (griptape.drivers.GriptapeCloudEventListenerDriver.api_key)"><code class="language-python">	*Optional[str]*	The API key to authenticate with Gen AI Builder.
<a href="./#griptape.drivers.GriptapeCloudEventListenerDriver.headers" title="headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.drivers.GriptapeCloudEventListenerDriver.headers)"><code class="language-python">	*dict*	The headers to use when making requests to Gen AI Builder. Defaults to include the Authorization header.

continues on next page

Table 72 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.GriptapeCloudEventListenerDriver.structure_run_id" title="structure_run_id = field(default=Factory(lambda: os.getenv('GT_CLOUD_STRUCTURE_RUN_ID')), kw_only=True) class-attribute instance-attribute (griptape.drivers.GriptapeCloudEventListenerDriver.structure_run_id)"><code class="language-python">	*Optional[str]*	The ID of the Structure Run to publish events to. Defaults to the GT_CLOUD_STRUCTURE_RUN_ID environment variable.

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
@define
class GriptapeCloudEventListenerDriver(BaseEventListenerDriver):
    """Driver for publishing events to Gen AI Builder.

    Attributes:
        base_url: The base URL of Gen AI Builder. Defaults to the GT_CLOUD_BASE_URL␣
↪environment variable.
        api_key: The API key to authenticate with Gen AI Builder.
        headers: The headers to use when making requests to Gen AI Builder. Defaults␣
↪to include the Authorization header.
        structure_run_id: The ID of the Structure Run to publish events to. Defaults␣
↪to the GT_CLOUD_STRUCTURE_RUN_ID environment variable.
    """

    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
        kw_only=True,
    )
    api_key: Optional[str] = field(default=Factory(lambda: os.getenv("GT_CLOUD_API_KEY
↪")), kw_only=True)
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        kw_only=True,
    )
    structure_run_id: Optional[str] = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_STRUCTURE_RUN_ID")), kw_only=True
    )
```

(continues on next page)

(continued from previous page)

```
    @structure_run_id.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_run_id(self, _: Attribute, structure_run_id: Optional[str]) -> None:
        if structure_run_id is None:
            raise ValueError(
                "structure_run_id must be set either in the constructor or as an␣
↪environment variable (GT_CLOUD_STRUCTURE_RUN_ID).",
            )

    @api_key.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_api_key(self, _: Attribute, api_key: Optional[str]) -> None:
        if api_key is None:
            raise ValueError(
                "No value was found for the GT_CLOUD_API_KEY environment variable. "
                "This environment variable is required when running in Gen AI Builder␣
↪for authorization. "
                "You can generate a Gen AI Builder API Key by visiting https://cloud.
↪griptape.ai/keys . "
                "Specify it as an environment variable when creating a Managed␣
↪Structure in Gen AI Builder."
            )

    def publish_event(self, event: BaseEvent | dict) -> None:
        from griptape.observability.observability import Observability

        event_payload = event.to_dict() if isinstance(event, BaseEvent) else event

        span_id = Observability.get_span_id()
        if span_id is not None:
            event_payload["span_id"] = span_id

        super().publish_event(event_payload)

    def try_publish_event_payload(self, event_payload: dict) -> None:
        self._post_event(self._get_event_request(event_payload))

    def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None:
        self._post_event([self._get_event_request(event_payload) for event_payload in␣
↪event_payload_batch])

    def _get_event_request(self, event_payload: dict) -> dict:
        return {
            "payload": event_payload,
            "timestamp": event_payload.get("timestamp", time.time()),
            "type": event_payload.get("type", "UserEvent"),
        }

    def _post_event(self, json: list[dict] | dict) -> None:
        requests.post(
```

(continues on next page)

(continued from previous page)

```
            url=griptape_cloud_url(self.base_url, f"api/structure-runs/{self.
↪structure_run_id}/events"),
            json=json,
            headers=self.headers,
        ).raise_for_status()
```

- `api_key = field(default=Factory(lambda: os.getenv('GT_CLOUD_API_KEY')), kw_only=True)` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')), kw_only=True)` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `structure_run_id = field(default=Factory(lambda: os.getenv('GT_CLOUD_STRUCTURE_RUN_ID')), kw_only=True)` class-attribute instance-attribute

_get_event_request(event_payload)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
def _get_event_request(self, event_payload: dict) -> dict:
    return {
        "payload": event_payload,
        "timestamp": event_payload.get("timestamp", time.time()),
        "type": event_payload.get("type", "UserEvent"),
    }
```

_post_event(json)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
def _post_event(self, json: list[dict] | dict) -> None:
    requests.post(
        url=griptape_cloud_url(self.base_url, f"api/structure-runs/{self.structure_
↪run_id}/events"),
        json=json,
        headers=self.headers,
    ).raise_for_status()
```

publish_event(event)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
def publish_event(self, event: BaseEvent | dict) -> None:
    from griptape.observability.observability import Observability

    event_payload = event.to_dict() if isinstance(event, BaseEvent) else event

    span_id = Observability.get_span_id()
    if span_id is not None:
        event_payload["span_id"] = span_id
```

(continues on next page)

(continued from previous page)

```
    super().publish_event(event_payload)
```

try_publish_event_payload(event_payload)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
def try_publish_event_payload(self, event_payload: dict) -> None:
    self._post_event(self._get_event_request(event_payload))
```

try_publish_event_payload_batch(event_payload_batch)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None:
    self._post_event([self._get_event_request(event_payload) for event_payload in␣
↪event_payload_batch])
```

validate*api_key(*, api_key)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
@api_key.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_api_key(self, _: Attribute, api_key: Optional[str]) -> None:
    if api_key is None:
        raise ValueError(
            "No value was found for the GT_CLOUD_API_KEY environment variable. "
            "This environment variable is required when running in Gen AI Builder for␣
↪authorization. "
            "You can generate a Gen AI Builder API Key by visiting https://cloud.
↪griptape.ai/keys . "
            "Specify it as an environment variable when creating a Managed Structure␣
↪in Gen AI Builder."
        )
```

validate*run_id(*, structure_run_id)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
@structure_run_id.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_run_id(self, _: Attribute, structure_run_id: Optional[str]) -> None:
    if structure_run_id is None:
        raise ValueError(
            "structure_run_id must be set either in the constructor or as an␣
↪environment variable (GT_CLOUD_STRUCTURE_RUN_ID).",
        )
```

GriptapeCloudFileManagerDriver

Bases: *BaseFileManagerDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.GriptapeCloudFileManagerDriver.bucket_id" title="bucket_id = field(default=Factory(lambda: os.getenv('GT_CLOUD_ kw_only=True) class-attribute instance-attribute (griptape.drivers.GriptapeCloudFileManagerDriver.bucket_id)"><cod class="language-python">	*Optional[str]*	The ID of the Bucket to list, load, and save Assets in. If not provided, the driver will attempt to retrieve the ID from the environment variable *GT_CLOUD_BUCKET_ID*.
<a href="./#griptape.drivers.GriptapeCloudFileManagerDriver.workdir" title="workdir property writable (griptape.drivers.GriptapeCloudFileManagerDriver.workdir)"><code class="language-python">	*str*	The working directory. List, load, and save operations will be performed relative to this directory.
<a href="./#griptape.drivers.GriptapeCloudFileManagerDriver.base_url" title="base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_ 'https://cloud.griptape.ai'))) class-attribute instance-attribute (griptape.drivers.GriptapeCloudFileManagerDriver.base_url)"><code class="language-python">	*str*	The base URL of the Gen AI Builder API. Defaults to the value of the environment variable *GT_CLOUD_BASE_URL* or *https://cloud.griptape.ai*.

continues on next page

Table 73 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.GriptapeCloudFileManagerDriver.api_key" title="api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_K class-attribute instance-attribute (griptape.drivers.GriptapeCloudFileManagerDriver.api_key)"><code class="language-python">	*str*	The API key to use for authenticating with the Gen AI Builder API. If not provided, the driver will attempt to retrieve the API key from the environment variable *GT_CLOUD_API_KEY*.

Raises

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
@define
class GriptapeCloudFileManagerDriver(BaseFileManagerDriver):
    """GriptapeCloudFileManagerDriver can be used to list, load, and save files as␣
↪Assets in Gen AI Builder Buckets.

    Attributes:
        bucket_id: The ID of the Bucket to list, load, and save Assets in. If not␣
↪provided, the driver will attempt to
            retrieve the ID from the environment variable `GT_CLOUD_BUCKET_ID`.
        workdir: The working directory. List, load, and save operations will be␣
↪performed relative to this directory.
        base_url: The base URL of the Gen AI Builder API. Defaults to the value of␣
↪the environment variable
            `GT_CLOUD_BASE_URL` or `https://cloud.griptape.ai`.
        api_key: The API key to use for authenticating with the Gen AI Builder API.␣
↪If not provided, the driver will
            attempt to retrieve the API key from the environment variable `GT_CLOUD_
↪API_KEY`.

    Raises:
        ValueError: If `api_key` is not provided, if `workdir` does not start with "/"
↪", or invalid `bucket_id` and/or `bucket_name` value(s) are provided.
    """

    bucket_id: Optional[str] = field(default=Factory(lambda: os.getenv("GT_CLOUD_
↪BUCKET_ID")), kw_only=True)
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]))
```

(continues on next page)

(continued from previous page)

```
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        init=False,
    )
    _workdir: str = field(default="/", kw_only=True, alias="workdir")

    @property
    def workdir(self) -> str:
        if self._workdir.startswith("/"):
            return self._workdir
        return f"/{self._workdir}"

    @workdir.setter
    def workdir(self, value: str) -> None:
        self._workdir = value

    @bucket_id.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_bucket_id(self, _: Attribute, value: Optional[str]) -> str:
        if value is None:
            raise ValueError(f"{self.__class__.__name__} requires an Bucket ID")
        return value

    def __attrs_post_init__(self) -> None:
        try:
            self._call_api(method="get", path=f"/buckets/{self.bucket_id}").json()
        except requests.exceptions.HTTPError as e:
            if e.response.status_code == 404:
                raise ValueError(f"No Bucket found with ID: {self.bucket_id}") from e
            raise ValueError(f"Unexpected error when retrieving Bucket with ID: {self.
↪bucket_id}") from e

    def try_list_files(self, path: str, postfix: str = "") -> list[str]:
        full_key = self._to_full_key(path)

        data = {"prefix": full_key}
        if postfix:
            data["postfix"] = postfix
        list_assets_response = self._call_api(
            method="get", path=f"/buckets/{self.bucket_id}/assets", params=data,␣
↪raise_for_status=False
        ).json()

        return [asset["name"] for asset in list_assets_response.get("assets", [])]

    def try_load_file(self, path: str) -> bytes:
        full_key = self._to_full_key(path)

        if self._is_a_directory(full_key):
            raise IsADirectoryError
```

(continues on next page)

(continued from previous page)

```
        try:
            sas_url, headers = self._get_asset_url(full_key)
            response = requests.get(sas_url, headers=headers)
            response.raise_for_status()
            return response.content
        except requests.exceptions.HTTPError as e:
            if e.response.status_code == 404:
                raise FileNotFoundError from e
            raise e

    def try_save_file(self, path: str, value: bytes) -> str:
        full_key = self._to_full_key(path)

        if self._is_a_directory(full_key):
            raise IsADirectoryError

        self._call_api(
            method="put",
            path=f"/buckets/{self.bucket_id}/assets",
            json={"name": full_key},
            raise_for_status=True,
        )

        sas_url, headers = self._get_asset_url(full_key)
        response = requests.put(sas_url, data=value, headers=headers)
        response.raise_for_status()

        return f"buckets/{self.bucket_id}/assets/{full_key}"

    def _get_asset_url(self, full_key: str) -> tuple[str, dict]:
        url_response = self._call_api(
            method="post", path=f"/buckets/{self.bucket_id}/asset-urls/{full_key}",␣
↪raise_for_status=True
        ).json()
        return url_response["url"], url_response.get("headers", {})

    def _get_url(self, path: str) -> str:
        path = path.lstrip("/")
        return griptape_cloud_url(self.base_url, f"api/{path}")

    def _call_api(
        self,
        method: str,
        path: str,
        json: Optional[dict] = None,
        params: Optional[dict] = None,
        *,
        raise_for_status: bool = True,
    ) -> requests.Response:
        res = requests.request(method, self._get_url(path), json=json, params=params,␣
↪headers=self.headers)
        if raise_for_status:
```

(continues on next page)

(continued from previous page)

```
            res.raise_for_status()
        return res

    def _is_a_directory(self, path: str) -> bool:
        return path == "" or path.endswith("/")

    def _to_full_key(self, path: str) -> str:
        path = path.lstrip("/")
        full_key = f"{self.workdir}/{path}"
        return full_key.lstrip("/")
```

- `_workdir = field(default='/', kw_only=True, alias='workdir')` class-attribute instance-attribute
- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']))` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')))` class-attribute instance-attribute
- `bucket_id = field(default=Factory(lambda: os.getenv('GT_CLOUD_BUCKET_ID')), kw_only=True)` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), init=False)` class-attribute instance-attribute
- `workdir` property writable

attrs_post_init()

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def __attrs_post_init__(self) -> None:
    try:
        self._call_api(method="get", path=f"/buckets/{self.bucket_id}").json()
    except requests.exceptions.HTTPError as e:
        if e.response.status_code == 404:
            raise ValueError(f"No Bucket found with ID: {self.bucket_id}") from e
        raise ValueError(f"Unexpected error when retrieving Bucket with ID: {self.
↪bucket_id}") from e
```

_call_api(method, path, json=None, params=None, *, raise_for_status=True)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def _call_api(
    self,
    method: str,
    path: str,
    json: Optional[dict] = None,
    params: Optional[dict] = None,
    *,
    raise_for_status: bool = True,
) -> requests.Response:
    res = requests.request(method, self._get_url(path), json=json, params=params,␣
```

(continues on next page)

(continued from previous page)

```
↪headers=self.headers)
    if raise_for_status:
        res.raise_for_status()
    return res
```

_get_asset_url(full_key)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def _get_asset_url(self, full_key: str) -> tuple[str, dict]:
    url_response = self._call_api(
        method="post", path=f"/buckets/{self.bucket_id}/asset-urls/{full_key}", raise_
↪for_status=True
    ).json()
    return url_response["url"], url_response.get("headers", {})
```

_get_url(path)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def _get_url(self, path: str) -> str:
    path = path.lstrip("/")
    return griptape_cloud_url(self.base_url, f"api/{path}")
```

_is_a_directory(path)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def _is_a_directory(self, path: str) -> bool:
    return path == "" or path.endswith("/")
```

_to_full_key(path)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def _to_full_key(self, path: str) -> str:
    path = path.lstrip("/")
    full_key = f"{self.workdir}/{path}"
    return full_key.lstrip("/")
```

try_list_files(path, postfix='')

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def try_list_files(self, path: str, postfix: str = "") -> list[str]:
    full_key = self._to_full_key(path)

    data = {"prefix": full_key}
    if postfix:
        data["postfix"] = postfix
    list_assets_response = self._call_api(
        method="get", path=f"/buckets/{self.bucket_id}/assets", params=data, raise_
```

(continues on next page)

(continued from previous page)

```
↪for_status=False
    ).json()

    return [asset["name"] for asset in list_assets_response.get("assets", [])]
```

try_load_file(path)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def try_load_file(self, path: str) -> bytes:
    full_key = self._to_full_key(path)

    if self._is_a_directory(full_key):
        raise IsADirectoryError

    try:
        sas_url, headers = self._get_asset_url(full_key)
        response = requests.get(sas_url, headers=headers)
        response.raise_for_status()
        return response.content
    except requests.exceptions.HTTPError as e:
        if e.response.status_code == 404:
            raise FileNotFoundError from e
        raise e
```

try_save_file(path, value)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def try_save_file(self, path: str, value: bytes) -> str:
    full_key = self._to_full_key(path)

    if self._is_a_directory(full_key):
        raise IsADirectoryError

    self._call_api(
        method="put",
        path=f"/buckets/{self.bucket_id}/assets",
        json={"name": full_key},
        raise_for_status=True,
    )

    sas_url, headers = self._get_asset_url(full_key)
    response = requests.put(sas_url, data=value, headers=headers)
    response.raise_for_status()

    return f"buckets/{self.bucket_id}/assets/{full_key}"
```

validate*bucket_id*(, value)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
@bucket_id.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_bucket_id(self, _: Attribute, value: Optional[str]) -> str:
    if value is None:
        raise ValueError(f"{self.__class__.__name__} requires an Bucket ID")
    return value
```

GriptapeCloudImageGenerationDriver

Bases: *BaseImageGenerationDriver* “)

Source Code in griptape/drivers/image_generation/griptape_cloud_image_generation_driver.py

```
@define
class GriptapeCloudImageGenerationDriver(BaseImageGenerationDriver):
    model: Optional[str] = field(default=None, kw_only=True)
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]))
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True), kw_only=True
    )
    style: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    quality: Literal["standard", "hd"] = field(default="standard", kw_only=True,␣
↪metadata={"serializable": True})
    image_size: Literal["1024x1024", "1024x1792", "1792x1024"] = field(
        default="1024x1024", kw_only=True, metadata={"serializable": True}
    )

    def try_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact:
        url = griptape_cloud_url(self.base_url, "api/images/generations")

        response = requests.post(
            url,
            headers=self.headers,
            json={
                "prompts": prompts,
                "driver_configuration": {
                    "model": self.model,
                    "image_size": self.image_size,
                    "quality": self.quality,
                    "style": self.style,
                },
            },
        )
```

(continues on next page)

(continued from previous page)

```
        response.raise_for_status()
        response = response.json()

        return ImageArtifact.from_dict(response["artifact"])

    def try_image_variation(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support image␣
↪variation")

    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪inpainting")

    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪outpainting")
```

- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']))` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')))` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `image_size = field(default='1024x1024', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(default=None, kw_only=True)` class-attribute instance-attribute
- `quality = field(default='standard', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `style = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

try_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/griptape_cloud_image_generation_driver.py

```
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support inpainting
↪")
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/griptape_cloud_image_generation_driver.py

```
def try_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support outpainting
↪")
```

try_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/griptape_cloud_image_generation_driver.py

```
def try_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support image␣
↪variation")
```

try_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/griptape_cloud_image_generation_driver.py

```
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    url = griptape_cloud_url(self.base_url, "api/images/generations")

    response = requests.post(
        url,
        headers=self.headers,
        json={
            "prompts": prompts,
```

(continues on next page)

(continued from previous page)

```
            "driver_configuration": {
                "model": self.model,
                "image_size": self.image_size,
                "quality": self.quality,
                "style": self.style,
            },
        },
    )
    response.raise_for_status()
    response = response.json()

    return ImageArtifact.from_dict(response["artifact"])
```

GriptapeCloudObservabilityDriver

Bases: *OpenTelemetryObservabilityDriver* “)

Source Code in griptape/drivers/observability/griptape_cloud_observability_driver.py

```
@define
class GriptapeCloudObservabilityDriver(OpenTelemetryObservabilityDriver):
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")), kw_only=True
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]), kw_
↪only=True)
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True), kw_only=True
    )
    structure_run_id: Optional[str] = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_STRUCTURE_RUN_ID")), kw_only=True
    )
    span_processor: SpanProcessor = field(
        default=Factory(
            lambda self: import_optional_dependency("opentelemetry.sdk.trace.export").
↪BatchSpanProcessor(
                GriptapeCloudObservabilityDriver.build_span_exporter(
                    base_url=self.base_url,
                    api_key=self.api_key,
                    headers=self.headers,
                    structure_run_id=self.structure_run_id,
                )
            ),
            takes_self=True,
        ),
        kw_only=True,
    )

    @structure_run_id.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
```

(continues on next page)

(continued from previous page)

```
    def validate_run_id(self, _: Attribute, structure_run_id: Optional[str]) -> None:
        if structure_run_id is None:
            raise ValueError(
                "structure_run_id must be set either in the constructor or as an␣
↪environment variable (GT_CLOUD_STRUCTURE_RUN_ID)."
            )

    @staticmethod
    def format_trace_id(trace_id: int) -> str:
        return str(UUID(int=trace_id))

    @staticmethod
    def format_span_id(span_id: int) -> str:
        return str(UUID(int=span_id))

    @staticmethod
    def build_span_exporter(base_url: str, api_key: str, headers: dict, structure_run_
↪id: str) -> SpanExporter:
        @define
        class SpanExporter(import_optional_dependency("opentelemetry.sdk.trace.export
↪").SpanExporter):
            base_url: str = field(kw_only=True)
            api_key: str = field(kw_only=True)
            headers: dict = field(kw_only=True)
            structure_run_id: str = field(kw_only=True)

            def export(self, spans: Sequence[ReadableSpan]) -> SpanExportResult:
                opentelemetry_util = import_optional_dependency("opentelemetry.sdk.
↪util")
                opentelemetry_trace_export = import_optional_dependency(
↪"opentelemetry.sdk.trace.export")

                url = griptape_cloud_url(self.base_url, f"api/structure-runs/{self.
↪structure_run_id}/spans")
                payload = [
                    {
                        "trace_id": GriptapeCloudObservabilityDriver.format_trace_
↪id(span.context.trace_id),
                        "span_id": GriptapeCloudObservabilityDriver.format_span_
↪id(span.context.span_id),
                        "parent_id": GriptapeCloudObservabilityDriver.format_span_
↪id(span.parent.span_id)
                        if span.parent
                        else None,
                        "name": span.name,
                        "start_time": opentelemetry_util.ns_to_iso_str(span.start_
↪time) if span.start_time else None,
                        "end_time": opentelemetry_util.ns_to_iso_str(span.end_time)␣
↪if span.end_time else None,
                        "status": span.status.status_code.name,
                        "attributes": {**span.attributes} if span.attributes else {},
                        "events": [
```

(continues on next page)

(continued from previous page)

```
                            {
                                "name": event.name,
                                "timestamp": opentelemetry_util.ns_to_iso_str(event.
↪timestamp)
                                if event.timestamp
                                else None,
                                "attributes": {**event.attributes} if event.
↪attributes else {},
                            }
                            for event in span.events
                        ],
                    }
                    for span in spans
                    if span.context is not None
                ]
                response = requests.post(url=url, json=payload, headers=self.headers)
                return (
                    opentelemetry_trace_export.SpanExportResult.SUCCESS
                    if response.status_code == 200
                    else opentelemetry_trace_export.SpanExportResult.FAILURE
                )

        return SpanExporter(
            base_url=base_url,
            api_key=api_key,
            headers=headers,
            structure_run_id=structure_run_id,
        )

    def get_span_id(self) -> Optional[str]:
        opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
        span = opentelemetry_trace.get_current_span()
        if span is opentelemetry_trace.INVALID_SPAN:
            return None
        return GriptapeCloudObservabilityDriver.format_span_id(span.get_span_
↪context().span_id)
```

- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']), kw_only=True)` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')), kw_only=True)` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `span_processor = field(default=Factory(lambda self: import_optional_dependency('opentelemetry.sdk.trace.export').BatchSpanProcessor(GriptapeCloudObservabilityDriver.build_span_exporter(base_url=self.base_url, api_key=self.api_key, headers=self.headers, structure_run_id=self.structure_run_id)), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `structure_run_id = field(default=Factory(lambda: os.getenv('GT_CLOUD_STRUCTURE_RUN_ID')), kw_only=True)` class-attribute instance-attribute

build_span_exporter(base_url, api_key, headers, structure_run_id)staticmethod

Source Code in griptape/drivers/observability/griptape_cloud_observability_driver.py

```
@staticmethod
def build_span_exporter(base_url: str, api_key: str, headers: dict, structure_run_id:␣
↪str) -> SpanExporter:
    @define
    class SpanExporter(import_optional_dependency("opentelemetry.sdk.trace.export").
↪SpanExporter):
        base_url: str = field(kw_only=True)
        api_key: str = field(kw_only=True)
        headers: dict = field(kw_only=True)
        structure_run_id: str = field(kw_only=True)

        def export(self, spans: Sequence[ReadableSpan]) -> SpanExportResult:
            opentelemetry_util = import_optional_dependency("opentelemetry.sdk.util")
            opentelemetry_trace_export = import_optional_dependency("opentelemetry.
↪sdk.trace.export")

            url = griptape_cloud_url(self.base_url, f"api/structure-runs/{self.
↪structure_run_id}/spans")
            payload = [
                {
                    "trace_id": GriptapeCloudObservabilityDriver.format_trace_id(span.
↪context.trace_id),
                    "span_id": GriptapeCloudObservabilityDriver.format_span_id(span.
↪context.span_id),
                    "parent_id": GriptapeCloudObservabilityDriver.format_span_id(span.
↪parent.span_id)
                    if span.parent
                    else None,
                    "name": span.name,
                    "start_time": opentelemetry_util.ns_to_iso_str(span.start_time)␣
↪if span.start_time else None,
                    "end_time": opentelemetry_util.ns_to_iso_str(span.end_time) if␣
↪span.end_time else None,
                    "status": span.status.status_code.name,
                    "attributes": {**span.attributes} if span.attributes else {},
                    "events": [
                        {
                            "name": event.name,
                            "timestamp": opentelemetry_util.ns_to_iso_str(event.
↪timestamp)
                            if event.timestamp
                            else None,
                            "attributes": {**event.attributes} if event.attributes␣
↪else {},
                        }
                        for event in span.events
                    ],
                }
                for span in spans
                if span.context is not None
```

(continues on next page)

(continued from previous page)

```
            ]
            response = requests.post(url=url, json=payload, headers=self.headers)
            return (
                opentelemetry_trace_export.SpanExportResult.SUCCESS
                if response.status_code == 200
                else opentelemetry_trace_export.SpanExportResult.FAILURE
            )

    return SpanExporter(
        base_url=base_url,
        api_key=api_key,
        headers=headers,
        structure_run_id=structure_run_id,
    )
```

format_span_id(span_id)staticmethod

Source Code in griptape/drivers/observability/griptape_cloud_observability_driver.py

```
@staticmethod
def format_span_id(span_id: int) -> str:
    return str(UUID(int=span_id))
```

format_trace_id(trace_id)staticmethod

Source Code in griptape/drivers/observability/griptape_cloud_observability_driver.py

```
@staticmethod
def format_trace_id(trace_id: int) -> str:
    return str(UUID(int=trace_id))
```

get_span_id()

Source Code in griptape/drivers/observability/griptape_cloud_observability_driver.py

```
def get_span_id(self) -> Optional[str]:
    opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
    span = opentelemetry_trace.get_current_span()
    if span is opentelemetry_trace.INVALID_SPAN:
        return None
    return GriptapeCloudObservabilityDriver.format_span_id(span.get_span_context().
↪span_id)
```

validate*run_id(*, structure_run_id)

Source Code in griptape/drivers/observability/griptape_cloud_observability_driver.py

```
@structure_run_id.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_run_id(self, _: Attribute, structure_run_id: Optional[str]) -> None:
    if structure_run_id is None:
        raise ValueError(
```

(continues on next page)

(continued from previous page)

```
            "structure_run_id must be set either in the constructor or as an␣
↪environment variable (GT_CLOUD_STRUCTURE_RUN_ID)."
        )
```

GriptapeCloudPromptDriver

Bases: *BasePromptDriver* “)

Source Code in griptape/drivers/prompt/griptape_cloud_prompt_driver.py

```
@define
class GriptapeCloudPromptDriver(BasePromptDriver):
    model: Optional[str] = field(default=None, kw_only=True)
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]))
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True), kw_only=True
    )
    tokenizer: BaseTokenizer = field(
        default=Factory(
            lambda self: SimpleTokenizer(
                characters_per_token=4,
                max_input_tokens=2000,
                max_output_tokens=self.max_tokens,
            ),
            takes_self=True,
        ),
        kw_only=True,
    )
    use_native_tools: bool = field(default=True, kw_only=True)
    structured_output_strategy: StructuredOutputStrategy = field(
        default="native", kw_only=True, metadata={"serializable": True}
    )

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        url = griptape_cloud_url(self.base_url, "api/chat/messages")

        params = self._base_params(prompt_stack)
        logger.debug(params)
        response = requests.post(url, headers=self.headers, json=params)
        response.raise_for_status()
        response_json = response.json()
        logger.debug(response_json)

        return Message.from_dict(response_json)

    @observable
```

(continues on next page)

(continued from previous page)

```
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        url = griptape_cloud_url(self.base_url, "api/chat/messages/stream")
        params = self._base_params(prompt_stack)
        logger.debug(params)
        with requests.post(url, headers=self.headers, json=params, stream=True) as␣
↪response:
            response.raise_for_status()
            for line in response.iter_lines():
                if line:
                    decoded_line = line.decode("utf-8")
                    if decoded_line.startswith("data:"):
                        delta_message_payload = decoded_line.removeprefix("data:").
↪strip()
                        logger.debug(delta_message_payload)
                        yield DeltaMessage.from_json(delta_message_payload)

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        return {
            "messages": prompt_stack.to_dict()["messages"],
            "tools": self.__to_griptape_tools(prompt_stack.tools),
            **({"output_schema": prompt_stack.to_output_json_schema()} if prompt_
↪stack.output_schema else {}),
            "driver_configuration": {
                **({"model": self.model} if self.model else {}),
                "max_tokens": self.max_tokens,
                "use_native_tools": self.use_native_tools,
                "temperature": self.temperature,
                "structured_output_strategy": self.structured_output_strategy,
                "extra_params": self.extra_params,
            },
        }

    def __to_griptape_tools(self, tools: list[BaseTool]) -> list[dict]:
        return [
            {
                "name": tool.name,
                "activities": [
                    {
                        "name": activity.__name__,
                        "description": tool.activity_description(activity),
                        "json_schema": tool.to_activity_json_schema(activity, "Schema
↪"),
                    }
                    for activity in tool.activities()
                ],
            }
            for tool in tools
        ]
```

- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']))`
  class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://`

`cloud.griptape.ai'))` class-attribute instance-attribute

- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `model = field(default=None, kw_only=True)` class-attribute instance-attribute
- `structured_output_strategy = field(default='native', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: SimpleTokenizer(characters_per_token=4, max_input_tokens=2000, max_output_tokens=self.max_tokens), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `use_native_tools = field(default=True, kw_only=True)` class-attribute instance-attribute

__to_griptape_tools(tools)

Source Code in griptape/drivers/prompt/griptape_cloud_prompt_driver.py

```
def __to_griptape_tools(self, tools: list[BaseTool]) -> list[dict]:
    return [
        {
            "name": tool.name,
            "activities": [
                {
                    "name": activity.__name__,
                    "description": tool.activity_description(activity),
                    "json_schema": tool.to_activity_json_schema(activity, "Schema"),
                }
                for activity in tool.activities()
            ],
        }
        for tool in tools
    ]
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/griptape_cloud_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    return {
        "messages": prompt_stack.to_dict()["messages"],
        "tools": self.__to_griptape_tools(prompt_stack.tools),
        **({"output_schema": prompt_stack.to_output_json_schema()} if prompt_stack.
↪output_schema else {}),
        "driver_configuration": {
            **({"model": self.model} if self.model else {}),
            "max_tokens": self.max_tokens,
            "use_native_tools": self.use_native_tools,
            "temperature": self.temperature,
            "structured_output_strategy": self.structured_output_strategy,
            "extra_params": self.extra_params,
        },
    }
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/griptape_cloud_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    url = griptape_cloud_url(self.base_url, "api/chat/messages")

    params = self._base_params(prompt_stack)
    logger.debug(params)
    response = requests.post(url, headers=self.headers, json=params)
    response.raise_for_status()
    response_json = response.json()
    logger.debug(response_json)

    return Message.from_dict(response_json)
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/griptape_cloud_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    url = griptape_cloud_url(self.base_url, "api/chat/messages/stream")
    params = self._base_params(prompt_stack)
    logger.debug(params)
    with requests.post(url, headers=self.headers, json=params, stream=True) as␣
↪response:
        response.raise_for_status()
        for line in response.iter_lines():
            if line:
                decoded_line = line.decode("utf-8")
                if decoded_line.startswith("data:"):
                    delta_message_payload = decoded_line.removeprefix("data:").strip()
                    logger.debug(delta_message_payload)
                    yield DeltaMessage.from_json(delta_message_payload)
```

GriptapeCloudRulesetDriver

Bases: *BaseRulesetDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.GriptapeCloudRulesetDriver.ruleset_id" title="ruleset_id = field(default=None, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.GriptapeCloudRulesetDriver.ruleset_id)"><code class="language-python">	*Optional[str]*	The ID of the Thread to store the conversation memory in. If not provided, the driver will attempt to retrieve the ID from the environment variable *GT_CLOUD_THREAD_ID*. If that is not set, a new Thread will be created.
<a href="./#griptape.drivers.GriptapeCloudRulesetDriver.base_url" title="base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_ 'https://cloud.griptape.ai'))) class-attribute instance-attribute (griptape.drivers.GriptapeCloudRulesetDriver.base_url)"><code class="language-python">	*str*	The base URL of the Gen AI Builder API. Defaults to the value of the environment variable *GT_CLOUD_BASE_URL* or *https://cloud.griptape.ai*.
<a href="./#griptape.drivers.GriptapeCloudRulesetDriver.api_key" title="api_key = field(default=Factory(lambda: os.getenv('GT_CLOUD_ class-attribute instance-attribute (griptape.drivers.GriptapeCloudRulesetDriver.api_key)"><code class="language-python">	*Optional[str]*	The API key to use for authenticating with the Gen AI Builder API. If not provided, the driver will attempt to retrieve the API key from the environment variable *GT_CLOUD_API_KEY*.

Raises

Source Code in griptape/drivers/ruleset/griptape_cloud_ruleset_driver.py

```
@define(kw_only=True)
class GriptapeCloudRulesetDriver(BaseRulesetDriver):
    """A driver for storing conversation memory in the Gen AI Builder.
```

(continues on next page)

(continued from previous page)

```
    Attributes:
        ruleset_id: The ID of the Thread to store the conversation memory in. If not␣
↪provided, the driver will attempt to
            retrieve the ID from the environment variable `GT_CLOUD_THREAD_ID`. If␣
↪that is not set, a new Thread will be
            created.
        base_url: The base URL of the Gen AI Builder API. Defaults to the value of␣
↪the environment variable
            `GT_CLOUD_BASE_URL` or `https://cloud.griptape.ai`.
        api_key: The API key to use for authenticating with the Gen AI Builder API.␣
↪If not provided, the driver will
            attempt to retrieve the API key from the environment variable `GT_CLOUD_
↪API_KEY`.

    Raises:
        ValueError: If `api_key` is not provided.
    """

    ruleset_id: Optional[str] = field(
        default=None,
        metadata={"serializable": True},
    )
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: Optional[str] = field(default=Factory(lambda: os.getenv("GT_CLOUD_API_KEY
↪")))
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        init=False,
    )

    @api_key.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_api_key(self, _: Attribute, value: Optional[str]) -> str:
        if value is None:
            raise ValueError(f"{self.__class__.__name__} requires an API key")
        return value

    def load(self, ruleset_name: str) -> tuple[list[BaseRule], dict[str, Any]]:
        """Load the ruleset from Gen AI Builder, using the ruleset name as an alias␣
↪if ruleset_id is not provided."""
        ruleset = None

        if self.ruleset_id is not None:
            res = self._call_api("get", f"/rulesets/{self.ruleset_id}", raise_for_
↪status=False)
            if res.status_code == 200:
                ruleset = res.json()
```

(continues on next page)

(continued from previous page)

```
        # use name as alias to get ruleset
        if ruleset is None:
            res = self._call_api("get", f"/rulesets?alias={ruleset_name}").json()
            if res.get("rulesets"):
                ruleset = res["rulesets"][0]

        # no ruleset by name or ruleset_id
        if ruleset is None:
            if self.raise_not_found:
                raise ValueError(f"No ruleset found with alias: {ruleset_name} or␣
↪ruleset_id: {self.ruleset_id}")
            return [], {}

        rules = self._call_api("get", f"/rules?ruleset_id={ruleset[ruleset_id]}").
↪json().get("rules", [])

        for rule in rules:
            rule["metadata"] = dict_merge(rule.get("metadata", {}), {"griptape_cloud_
↪rule_id": rule["rule_id"]})

        return [self._get_rule(rule["rule"], rule["metadata"]) for rule in rules],␣
↪ruleset.get("metadata", {})

    def _get_url(self, path: str) -> str:
        path = path.lstrip("/")
        return griptape_cloud_url(self.base_url, f"api/{path}")

    def _call_api(self, method: str, path: str, *, raise_for_status: bool = True) ->␣
↪requests.Response:
        res = requests.request(method, self._get_url(path), headers=self.headers)
        if raise_for_status:
            res.raise_for_status()
        return res
```

- `api_key = field(default=Factory(lambda: os.getenv('GT_CLOUD_API_KEY')))` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')))` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), init=False)` class-attribute instance-attribute
- `ruleset_id = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

_call_api(method, path, *, raise_for_status=True)

Source Code in griptape/drivers/ruleset/griptape_cloud_ruleset_driver.py

```
def _call_api(self, method: str, path: str, *, raise_for_status: bool = True) ->␣
↪requests.Response:
    res = requests.request(method, self._get_url(path), headers=self.headers)
```

(continues on next page)

(continued from previous page)

```
    if raise_for_status:
        res.raise_for_status()
    return res
```

_get_url(path)

Source Code in griptape/drivers/ruleset/griptape_cloud_ruleset_driver.py

```
def _get_url(self, path: str) -> str:
    path = path.lstrip("/")
    return griptape_cloud_url(self.base_url, f"api/{path}")
```

load(ruleset_name)

Source Code in griptape/drivers/ruleset/griptape_cloud_ruleset_driver.py

```
def load(self, ruleset_name: str) -> tuple[list[BaseRule], dict[str, Any]]:
    """Load the ruleset from Gen AI Builder, using the ruleset name as an alias if␣
↪ruleset_id is not provided."""
    ruleset = None

    if self.ruleset_id is not None:
        res = self._call_api("get", f"/rulesets/{self.ruleset_id}", raise_for_
↪status=False)
        if res.status_code == 200:
            ruleset = res.json()

    # use name as alias to get ruleset
    if ruleset is None:
        res = self._call_api("get", f"/rulesets?alias={ruleset_name}").json()
        if res.get("rulesets"):
            ruleset = res["rulesets"][0]

    # no ruleset by name or ruleset_id
    if ruleset is None:
        if self.raise_not_found:
            raise ValueError(f"No ruleset found with alias: {ruleset_name} or ruleset_
↪id: {self.ruleset_id}")
        return [], {}

    rules = self._call_api("get", f"/rules?ruleset_id={ruleset[ruleset_id]}").json().
↪get("rules", [])

    for rule in rules:
        rule["metadata"] = dict_merge(rule.get("metadata", {}), {"griptape_cloud_rule_
↪id": rule["rule_id"]})

    return [self._get_rule(rule["rule"], rule["metadata"]) for rule in rules],␣
↪ruleset.get("metadata", {})
```

validate*api_key*(, value)

Source Code in griptape/drivers/ruleset/griptape_cloud_ruleset_driver.py

```
@api_key.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_api_key(self, _: Attribute, value: Optional[str]) -> str:
    if value is None:
        raise ValueError(f"{self.__class__.__name__} requires an API key")
    return value
```

GriptapeCloudStructureRunDriver

Bases: *BaseStructureRunDriver* “)

Source Code in griptape/drivers/structure_run/griptape_cloud_structure_run_driver.py

```
@define
class GriptapeCloudStructureRunDriver(BaseStructureRunDriver):
    base_url: str = field(default="https://cloud.griptape.ai", kw_only=True)
    api_key: str = field(kw_only=True)
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        kw_only=True,
    )
    structure_id: str = field(kw_only=True)
    structure_run_wait_time_interval: int = field(default=2, kw_only=True)
    structure_run_max_wait_time_attempts: int = field(default=20, kw_only=True)
    async_run: bool = field(default=False, kw_only=True)

    def try_run(self, *args: BaseArtifact) -> BaseArtifact | InfoArtifact:
        structure_run_id = self._create_run(*args)

        if self.async_run:
            return InfoArtifact("Run started successfully")
        return self._get_run_result(structure_run_id)

    def _create_run(self, *args: BaseArtifact) -> str:
        url = griptape_cloud_url(self.base_url, f"api/structures/{self.structure_id}/
↪runs")

        env_vars = [{"name": key, "value": value, "source": "manual"} for key, value␣
↪in self.env.items()]

        response = requests.post(
            url,
            json={"args": [arg.value for arg in args], "env_vars": env_vars},
            headers=self.headers,
        )
        response.raise_for_status()
        response_json = response.json()

        return response_json["structure_run_id"]
```

(continues on next page)

(continued from previous page)

```
    def _get_run_result(self, structure_run_id: str) -> BaseArtifact | InfoArtifact:
        events = self._get_run_events(structure_run_id)
        output = None

        for event in events:
            event_type = event["type"]
            event_payload = event.get("payload", {})
            if event["origin"] == "USER":
                try:
                    if "span_id" in event_payload:
                        span_id = event_payload.pop("span_id")
                        if "meta" in event_payload:
                            event_payload["meta"]["span_id"] = span_id
                        else:
                            event_payload["meta"] = {"span_id": span_id}
                    EventBus.publish_event(BaseEvent.from_dict(event_payload))
                except ValueError as e:
                    logger.warning("Failed to deserialize event: %s", e)
                if event["type"] == "FinishStructureRunEvent":
                    output = BaseArtifact.from_dict(event_payload["output_task_output
↪"])
            elif event["origin"] == "SYSTEM":
                if event_type == "StructureRunError":
                    output = ErrorArtifact(event_payload["status_detail"]["error"])

        if output is None:
            raise ValueError("Output not found.")

        return output

    def _get_run_events(self, structure_run_id: str) -> Iterator[dict]:
        url = griptape_cloud_url(self.base_url, f"api/structure-runs/{structure_run_
↪id}/events/stream")
        with requests.get(url, headers=self.headers, stream=True) as response:
            response.raise_for_status()
            for line in response.iter_lines():
                if line:
                    decoded_line = line.decode("utf-8")
                    if decoded_line.startswith("data:"):
                        yield json.loads(decoded_line.removeprefix("data:").strip())
```

- `api_key = field(kw_only=True)` class-attribute instance-attribute
- `async_run = field(default=False, kw_only=True)` class-attribute instance-attribute
- `base_url = field(default='https://cloud.griptape.ai', kw_only=True)` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `structure_id = field(kw_only=True)` class-attribute instance-attribute
- `structure_run_max_wait_time_attempts = field(default=20, kw_only=True)` class-attribute

instance-attribute

- `structure_run_wait_time_interval = field(default=2, kw_only=True)` class-attribute instance-attribute

_create_run(*args)

Source Code in griptape/drivers/structure_run/griptape_cloud_structure_run_driver.py

```
def _create_run(self, *args: BaseArtifact) -> str:
    url = griptape_cloud_url(self.base_url, f"api/structures/{self.structure_id}/runs
↪")

    env_vars = [{"name": key, "value": value, "source": "manual"} for key, value in␣
↪self.env.items()]

    response = requests.post(
        url,
        json={"args": [arg.value for arg in args], "env_vars": env_vars},
        headers=self.headers,
    )
    response.raise_for_status()
    response_json = response.json()

    return response_json["structure_run_id"]
```

_get_run_events(structure_run_id)

Source Code in griptape/drivers/structure_run/griptape_cloud_structure_run_driver.py

```
def _get_run_events(self, structure_run_id: str) -> Iterator[dict]:
    url = griptape_cloud_url(self.base_url, f"api/structure-runs/{structure_run_id}/
↪events/stream")
    with requests.get(url, headers=self.headers, stream=True) as response:
        response.raise_for_status()
        for line in response.iter_lines():
            if line:
                decoded_line = line.decode("utf-8")
                if decoded_line.startswith("data:"):
                    yield json.loads(decoded_line.removeprefix("data:").strip())
```

_get_run_result(structure_run_id)

Source Code in griptape/drivers/structure_run/griptape_cloud_structure_run_driver.py

```
def _get_run_result(self, structure_run_id: str) -> BaseArtifact | InfoArtifact:
    events = self._get_run_events(structure_run_id)
    output = None

    for event in events:
        event_type = event["type"]
        event_payload = event.get("payload", {})
        if event["origin"] == "USER":
            try:
```

(continues on next page)

(continued from previous page)

```
                if "span_id" in event_payload:
                    span_id = event_payload.pop("span_id")
                    if "meta" in event_payload:
                        event_payload["meta"]["span_id"] = span_id
                    else:
                        event_payload["meta"] = {"span_id": span_id}
                EventBus.publish_event(BaseEvent.from_dict(event_payload))
            except ValueError as e:
                logger.warning("Failed to deserialize event: %s", e)
            if event["type"] == "FinishStructureRunEvent":
                output = BaseArtifact.from_dict(event_payload["output_task_output"])
        elif event["origin"] == "SYSTEM":
            if event_type == "StructureRunError":
                output = ErrorArtifact(event_payload["status_detail"]["error"])

    if output is None:
        raise ValueError("Output not found.")

    return output
```

try_run(*args)

Source Code in griptape/drivers/structure_run/griptape_cloud_structure_run_driver.py

```
def try_run(self, *args: BaseArtifact) -> BaseArtifact | InfoArtifact:
    structure_run_id = self._create_run(*args)

    if self.async_run:
        return InfoArtifact("Run started successfully")
    return self._get_run_result(structure_run_id)
```

GriptapeCloudVectorStoreDriver

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.GriptapeCloudVectorStoreDriver.api_key" title="api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_K class-attribute instance-attribute (griptape.drivers.GriptapeCloudVectorStoreDriver.api_key)"><c class="language-python">	*str*	API Key for Gen AI Builder.
<a href="./#griptape.drivers.GriptapeCloudVectorStoreDriver.knowledge_base_id" title="knowledge_base_id = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.GriptapeCloudVectorStoreDriver.knowledge_base_id)"><code class="language-python">	*str*	Knowledge Base ID for Gen AI Builder.
<a href="./#griptape.drivers.GriptapeCloudVectorStoreDriver.base_url" title="base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_ 'https://cloud.griptape.ai'))) class-attribute instance-attribute (griptape.drivers.GriptapeCloudVectorStoreDriver.base_url)"><class="language-python">	*str*	Base URL for Gen AI Builder.

continues on next page

Table 75 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.GriptapeCloudVectorStoreDriver.headers" title="headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.drivers.GriptapeCloudVectorStoreDriver.headers)"><code class="language-python">	*dict*	Headers for Gen AI Builder.

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
@define
class GriptapeCloudVectorStoreDriver(BaseVectorStoreDriver):
    """A vector store driver for Gen AI Builder Knowledge Bases.

    Attributes:
        api_key: API Key for Gen AI Builder.
        knowledge_base_id: Knowledge Base ID for Gen AI Builder.
        base_url: Base URL for Gen AI Builder.
        headers: Headers for Gen AI Builder.
    """

    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]))
    knowledge_base_id: str = field(kw_only=True, metadata={"serializable": True})
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        kw_only=True,
    )
    embedding_driver: BaseEmbeddingDriver = field(
        default=Factory(lambda: DummyEmbeddingDriver()),
        metadata={"serializable": True},
        kw_only=True,
        init=False,
    )
```

(continues on next page)

(continued from previous page)

```
    def upsert_vector(
        self,
        vector: list[float],
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        raise NotImplementedError(f"{self.__class__.__name__} does not support vector␣
↪upsert.")

    def upsert_text_artifact(
        self,
        artifact: TextArtifact,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        vector_id: Optional[str] = None,
        **kwargs,
    ) -> str:
        raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪artifact upsert.")

    def upsert_text(
        self,
        string: str,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪upsert.")

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪BaseVectorStoreDriver.Entry:
        raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")

    def load_artifacts(self, *, namespace: Optional[str] = None) -> ListArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪Artifact loading.")

    def query(
        self,
        query: str | TextArtifact | ImageArtifact,
        *,
        count: Optional[int] = None,
```

(continues on next page)

(continued from previous page)

```
        namespace: Optional[str] = None,
        include_vectors: Optional[bool] = None,
        distance_metric: Optional[str] = None,
        # GriptapeCloudVectorStoreDriver-specific params:
        filter: Optional[dict] = None,  # noqa: A002
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Performs a query on the Knowledge Base.

        Performs a query on the Knowledge Base and returns Artifacts with close␣
↪vector proximity to the query, optionally filtering to only those that match the␣
↪provided filter(s).
        """
        if isinstance(query, ImageArtifact):
            raise ValueError(f"{self.__class__.__name__} does not support querying␣
↪with Image Artifacts.")
        url = griptape_cloud_url(self.base_url, f"api/knowledge-bases/{self.knowledge_
↪base_id}/query")

        query_args = {
            "count": count,
            "distance_metric": distance_metric,
            "filter": filter,
            "include_vectors": include_vectors,
        }
        query_args = {k: v for k, v in query_args.items() if v is not None}

        request: dict[str, Any] = {
            "query": str(query),
            "query_args": query_args,
        }

        response = requests.post(url, json=request, headers=self.headers).json()
        entries = response.get("entries", [])
        return [BaseVectorStoreDriver.Entry.from_dict(entry) for entry in entries]

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")
```

- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']))` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')))` class-attribute instance-attribute
- `embedding_driver = field(default=Factory(lambda: DummyEmbeddingDriver()), metadata={'serializable': True}, kw_only=True, init=False)` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `knowledge_base_id = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

delete_vector(vector_id)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

load_artifacts(*, namespace=None)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def load_artifacts(self, *, namespace: Optional[str] = None) -> ListArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support Artifact␣
↪loading.")
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")
```

load_entry(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪BaseVectorStoreDriver.Entry:
    raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")
```

query(query, *, count=None, namespace=None, include_vectors=None, distance_metric=None, filter=None, **kwargs)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def query(
    self,
    query: str | TextArtifact | ImageArtifact,
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: Optional[bool] = None,
    distance_metric: Optional[str] = None,
    # GriptapeCloudVectorStoreDriver-specific params:
    filter: Optional[dict] = None,  # noqa: A002
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    """Performs a query on the Knowledge Base.

    Performs a query on the Knowledge Base and returns Artifacts with close vector␣
↪proximity to the query, optionally filtering to only those that match the provided␣
```

(continues on next page)

(continued from previous page)

```
↪filter(s).
    """
    if isinstance(query, ImageArtifact):
        raise ValueError(f"{self.__class__.__name__} does not support querying with␣
↪Image Artifacts.")
    url = griptape_cloud_url(self.base_url, f"api/knowledge-bases/{self.knowledge_
↪base_id}/query")

    query_args = {
        "count": count,
        "distance_metric": distance_metric,
        "filter": filter,
        "include_vectors": include_vectors,
    }
    query_args = {k: v for k, v in query_args.items() if v is not None}

    request: dict[str, Any] = {
        "query": str(query),
        "query_args": query_args,
    }

    response = requests.post(url, json=request, headers=self.headers).json()
    entries = response.get("entries", [])
    return [BaseVectorStoreDriver.Entry.from_dict(entry) for entry in entries]
```

upsert_text(string, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def upsert_text(
    self,
    string: str,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪upsert.")
```

upsert_text_artifact(artifact, namespace=None, meta=None, vector_id=None, **kwargs)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def upsert_text_artifact(
    self,
    artifact: TextArtifact,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    vector_id: Optional[str] = None,
    **kwargs,
) -> str:
```

(continues on next page)

(continued from previous page)

```
    raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪artifact upsert.")
```

upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    raise NotImplementedError(f"{self.__class__.__name__} does not support vector␣
↪upsert.")
```

GrokPromptDriver

Bases: *AzureOpenAiChatPromptDriver* “)

Source Code in griptape/drivers/prompt/grok_prompt_driver.py

```
@define
class GrokPromptDriver(OpenAiChatPromptDriver):
    base_url: str = field(default="https://api.x.ai/v1", kw_only=True, metadata={
↪"serializable": True})
    tokenizer: GrokTokenizer = field(
        default=Factory(
            lambda self: GrokTokenizer(base_url=self.base_url, api_key=self.api_key,␣
↪model=self.model), takes_self=True
        ),
        kw_only=True,
        metadata={"serializable": True},
    )
```

- `base_url = field(default='https://api.x.ai/v1', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: GrokTokenizer(base_url=self.base_url, api_key=self.api_key, model=self.model), takes_self=True), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

HuggingFaceHubEmbeddingDriver

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.HuggingFaceHubEmbeddingDriver.api_token" title="api_token = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.HuggingFaceHubEmbeddingDriver.api_token)"><code class="language-python">	*str*	Hugging Face Hub API token.
<a href="./#griptape.drivers.BaseEmbeddingDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.HuggingFaceHubEmbeddingDriver.model)"><code class="language-python">	*str*	Hugging Face Hub model name.
<a href="./#griptape.drivers.HuggingFaceHubEmbeddingDriver.client" title="client() (griptape.drivers.HuggingFaceHubEmbeddingDriver.client)"><code class="language-python">	*InferenceClient*	Custom *InferenceApi*.

Source Code in griptape/drivers/embedding/huggingface_hub_embedding_driver.py

```
@define
class HuggingFaceHubEmbeddingDriver(BaseEmbeddingDriver):
    """Hugging Face Hub Embedding Driver.

    Attributes:
        api_token: Hugging Face Hub API token.
        model: Hugging Face Hub model name.
        client: Custom `InferenceApi`.
    """

    api_token: str = field(kw_only=True, metadata={"serializable": True})
```

(continues on next page)

(continued from previous page)

```
    _client: Optional[InferenceClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> InferenceClient:
        return import_optional_dependency("huggingface_hub").InferenceClient(
            model=self.model,
            token=self.api_token,
        )

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        response = self.client.feature_extraction(chunk)

        return [float(val) for val in response.flatten().tolist()]
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_token = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/huggingface_hub_embedding_driver.py

```
@lazy_property()
def client(self) -> InferenceClient:
    return import_optional_dependency("huggingface_hub").InferenceClient(
        model=self.model,
        token=self.api_token,
    )
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/huggingface_hub_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    response = self.client.feature_extraction(chunk)

    return [float(val) for val in response.flatten().tolist()]
```

HuggingFaceHubPromptDriver

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.HuggingFaceHubPromptDriver.api_token" title="api_token = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.HuggingFaceHubPromptDriver.api_token)"><code class="language-python">	*str*	Hugging Face Hub API token.
use_gpu	*str*	Use GPU during model run.
<a href="./#griptape.drivers.HuggingFaceHubPromptDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.HuggingFaceHubPromptDriver.model)"><code class="language-python">	*str*	Hugging Face Hub model name.
<a href="./#griptape.drivers.HuggingFaceHubPromptDriver.client" title="client() (griptape.drivers.HuggingFaceHubPromptDriver.client)"><code class="language-python">	*InferenceClient*	Custom *InferenceApi*.

continues on next page

Table 77 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.HuggingFaceHubPromptDriver.tokenizer" title="tokenizer = field(default=Factory(lambda self: HuggingFaceTokenizer(model=self.model, max_output_tokens=self.max_tokens), takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.drivers.HuggingFaceHubPromptDriver.tokenizer)"><code class="language-python">	<a href="../tokenizers#griptape.tokenizers.HuggingFaceTokenizer" title="HuggingFaceTokenizer (griptape.tokenizers.HuggingFaceTokenizer)"><code class="language-python">HuggingFaceTokenizer</coc	Custom *HuggingFaceTokenizer*.

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
@define
class HuggingFaceHubPromptDriver(BasePromptDriver):
    """Hugging Face Hub Prompt Driver.

    Attributes:
        api_token: Hugging Face Hub API token.
        use_gpu: Use GPU during model run.
        model: Hugging Face Hub model name.
        client: Custom `InferenceApi`.
        tokenizer: Custom `HuggingFaceTokenizer`.
    """

    api_token: str = field(kw_only=True, metadata={"serializable": True})
    max_tokens: int = field(default=250, kw_only=True, metadata={"serializable": True}
↪)
    model: str = field(kw_only=True, metadata={"serializable": True})
    structured_output_strategy: StructuredOutputStrategy = field(
        default="native", kw_only=True, metadata={"serializable": True}
    )
    tokenizer: HuggingFaceTokenizer = field(
        default=Factory(
            lambda self: HuggingFaceTokenizer(model=self.model, max_output_
↪tokens=self.max_tokens),
            takes_self=True,
        ),
        kw_only=True,
    )
```

(continues on next page)

(continued from previous page)

```
    _client: Optional[InferenceClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> InferenceClient:
        return import_optional_dependency("huggingface_hub").InferenceClient(
            model=self.model,
            token=self.api_token,
        )

    @structured_output_strategy.validator  # pyright:␣
↪ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
        if value == "tool":
            raise ValueError(f"{__class__.__name__} does not support `{value}`␣
↪structured output strategy.")

        return value

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        prompt = self.prompt_stack_to_string(prompt_stack)
        full_params = self._base_params(prompt_stack)
        logger.debug(
            {
                "prompt": prompt,
                **full_params,
            }
        )

        response = self.client.text_generation(
            prompt,
            **full_params,
        )
        logger.debug(response)
        input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))
        output_tokens = len(self.tokenizer.tokenizer.encode(response))  # pyright:␣
↪ignore[reportArgumentType]

        return Message(
            content=response,
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(input_tokens=input_tokens, output_tokens=output_
↪tokens),
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        prompt = self.prompt_stack_to_string(prompt_stack)
        full_params = {**self._base_params(prompt_stack), "stream": True}
        logger.debug(
```

(continues on next page)

(continued from previous page)

```
            {
                "prompt": prompt,
                **full_params,
            }
        )

        response = self.client.text_generation(prompt, **full_params)

        input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))

        full_text = ""
        for token in response:
            logger.debug(token)
            full_text += token
            yield DeltaMessage(content=TextDeltaMessageContent(token, index=0))

        output_tokens = len(self.tokenizer.tokenizer.encode(full_text))  # pyright:␣
↪ignore[reportArgumentType]
        yield DeltaMessage(usage=DeltaMessage.Usage(input_tokens=input_tokens, output_
↪tokens=output_tokens))

    def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
        return self.tokenizer.tokenizer.decode(self.__prompt_stack_to_tokens(prompt_
↪stack))  # pyright: ignore[reportArgumentType]

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        params = {
            "return_full_text": False,
            "max_new_tokens": self.max_tokens,
            **self.extra_params,
        }

        if prompt_stack.output_schema and self.structured_output_strategy == "native":
            # https://huggingface.co/learn/cookbook/en/structured_generation#-
↪constrained-decoding
            output_schema = prompt_stack.to_output_json_schema()
            # Grammar does not support $schema and $id
            del output_schema["$schema"]
            del output_schema["$id"]
            params["grammar"] = {"type": "json", "value": output_schema}

        return params

    def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
        messages = []
        for message in prompt_stack.messages:
            if len(message.content) == 1:
                messages.append({"role": message.role, "content": message.to_text()})
            else:
                raise ValueError("Invalid input content length.")

        return messages
```

(continues on next page)

(continued from previous page)

```
    def __prompt_stack_to_tokens(self, prompt_stack: PromptStack) -> list[int]:
        messages = self._prompt_stack_to_messages(prompt_stack)
        tokens = self.tokenizer.tokenizer.apply_chat_template(messages, add_
↪generation_prompt=True, tokenize=True)

        if isinstance(tokens, list):
            return tokens  # pyright: ignore[reportReturnType] According to the␣
↪[docs](https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils
↪#transformers.PreTrainedTokenizerBase.apply_chat_template), the return type is␣
↪List[int].
        raise ValueError("Invalid output type.")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_token = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `max_tokens = field(default=250, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='native', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: HuggingFaceTokenizer(model=self.model, max_output_tokens=self.max_tokens), takes_self=True), kw_only=True)` class-attribute instance-attribute

__prompt_stack_to_tokens(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
def __prompt_stack_to_tokens(self, prompt_stack: PromptStack) -> list[int]:
    messages = self._prompt_stack_to_messages(prompt_stack)
    tokens = self.tokenizer.tokenizer.apply_chat_template(messages, add_generation_
↪prompt=True, tokenize=True)

    if isinstance(tokens, list):
        return tokens  # pyright: ignore[reportReturnType] According to the␣
↪[docs](https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils
↪#transformers.PreTrainedTokenizerBase.apply_chat_template), the return type is␣
↪List[int].
    raise ValueError("Invalid output type.")
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    params = {
        "return_full_text": False,
```

(continues on next page)

(continued from previous page)

```
        "max_new_tokens": self.max_tokens,
        **self.extra_params,
    }

    if prompt_stack.output_schema and self.structured_output_strategy == "native":
        # https://huggingface.co/learn/cookbook/en/structured_generation#-constrained-
↪decoding
        output_schema = prompt_stack.to_output_json_schema()
        # Grammar does not support $schema and $id
        del output_schema["$schema"]
        del output_schema["$id"]
        params["grammar"] = {"type": "json", "value": output_schema}

    return params
```

_prompt_stack_to_messages(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
    messages = []
    for message in prompt_stack.messages:
        if len(message.content) == 1:
            messages.append({"role": message.role, "content": message.to_text()})
        else:
            raise ValueError("Invalid input content length.")

    return messages
```

client()

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
@lazy_property()
def client(self) -> InferenceClient:
    return import_optional_dependency("huggingface_hub").InferenceClient(
        model=self.model,
        token=self.api_token,
    )
```

prompt_stack_to_string(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
    return self.tokenizer.tokenizer.decode(self.__prompt_stack_to_tokens(prompt_
↪stack))  # pyright: ignore[reportArgumentType]
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    prompt = self.prompt_stack_to_string(prompt_stack)
    full_params = self._base_params(prompt_stack)
    logger.debug(
        {
            "prompt": prompt,
            **full_params,
        }
    )

    response = self.client.text_generation(
        prompt,
        **full_params,
    )
    logger.debug(response)
    input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))
    output_tokens = len(self.tokenizer.tokenizer.encode(response))  # pyright:␣
↪ignore[reportArgumentType]

    return Message(
        content=response,
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(input_tokens=input_tokens, output_tokens=output_tokens),
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    prompt = self.prompt_stack_to_string(prompt_stack)
    full_params = {**self._base_params(prompt_stack), "stream": True}
    logger.debug(
        {
            "prompt": prompt,
            **full_params,
        }
    )

    response = self.client.text_generation(prompt, **full_params)

    input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))

    full_text = ""
    for token in response:
        logger.debug(token)
        full_text += token
        yield DeltaMessage(content=TextDeltaMessageContent(token, index=0))
```

(continues on next page)

(continued from previous page)

```
    output_tokens = len(self.tokenizer.tokenizer.encode(full_text))  # pyright:␣
↪ignore[reportArgumentType]
    yield DeltaMessage(usage=DeltaMessage.Usage(input_tokens=input_tokens, output_
↪tokens=output_tokens))
```

validate*structured_output_strategy*(, value)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
@structured_output_strategy.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
    if value == "tool":
        raise ValueError(f"{__class__.__name__} does not support `{value}` structured␣
↪output strategy.")

    return value
```

HuggingFacePipelineImageGenerationDriver

Bases: *BaseImageGenerationDriver* “) , ABC

Attributes

Name	Type	Description
<a href="./#griptape.drivers.HuggingFacePipelineImageGenerationDriver.pipeline_driver" title="pipeline_driver = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.HuggingFacePipelineImageGenerationDriver.pipeline_driver" class="language-python">	<a href="image_generation_pipeline#griptape.drivers.image_generation_pipeline.BaseDiffusionImageGenerationPipelineDriver" title="BaseDiffusionImageGenerationPipelineDriver (griptape.drivers.image_generation_pipeline.BaseDiffusionImageGenerationPipelineDriver)"><code class="language-python">BaseDiffusionImageGenerat	A pipeline image generation model driver typed for the specific pipeline required by the model.

continues on next page

Table 78 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.HuggingFacePipelineImageGenerationDriver.device" title="device = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.HuggingFacePipelineImageGenerationDriver.device)"><code class="language-python">	*Optional[str]*	The hardware device used for inference. For example, "cpu", "cuda", or "mps".
<a href="./#griptape.drivers.HuggingFacePipelineImageGenerationDriver.output_format" title="output_format = field(default='png', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.HuggingFacePipelineImageGenerationDriver.output_format)"><code class="language-python">	*str*	The format the generated image is returned in. Defaults to "png".

Source Code in griptape/drivers/image_generation/huggingface_pipeline_image_generation_driver.py

```
@define
class HuggingFacePipelineImageGenerationDriver(BaseImageGenerationDriver, ABC):
    """Image generation driver for models hosted by Hugging Faces Diffusion Pipeline.

    For more information, see the HuggingFace documentation for Diffusers:
            https://huggingface.co/docs/diffusers/en/index

    Attributes:
        pipeline_driver: A pipeline image generation model driver typed for the␣
↪specific pipeline required by the model.
        device: The hardware device used for inference. For example, "cpu", "cuda",␣
↪or "mps".
        output_format: The format the generated image is returned in. Defaults to "png
↪".
    """
```

(continues on next page)

(continued from previous page)

```
    pipeline_driver: BaseDiffusionImageGenerationPipelineDriver = field(kw_only=True,␣
↪metadata={"serializable": True})
    device: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    output_format: str = field(default="png", kw_only=True, metadata={"serializable":␣
↪True})

    def try_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact:
        pipeline = self.pipeline_driver.prepare_pipeline(self.model, self.device)

        prompt = ", ".join(prompts)
        output_image = pipeline(
            prompt, **self.pipeline_driver.make_additional_params(negative_prompts,␣
↪self.device)
        ).images[0]

        buffer = io.BytesIO()
        output_image.save(buffer, format=self.output_format.upper())

        return ImageArtifact(
            value=buffer.getvalue(),
            format=self.output_format.lower(),
            height=output_image.height,
            width=output_image.width,
            meta={"prompt": prompt},
        )

    def try_image_variation(
        self, prompts: list[str], image: ImageArtifact, negative_prompts:␣
↪Optional[list[str]] = None
    ) -> ImageArtifact:
        pil_image = import_optional_dependency("PIL.Image")

        pipeline = self.pipeline_driver.prepare_pipeline(self.model, self.device)

        prompt = ", ".join(prompts)
        input_image = pil_image.open(io.BytesIO(image.value))
        # The size of the input image drives the size of the output image.
        # Resize the input image to the configured dimensions.
        output_width, output_height = self.pipeline_driver.output_image_dimensions
        if input_image.height != output_height or input_image.width != output_width:
            input_image = input_image.resize((output_width, output_height))

        output_image = pipeline(
            prompt,
            **self.pipeline_driver.make_image_param(input_image),
            **self.pipeline_driver.make_additional_params(negative_prompts, self.
↪device),
        ).images[0]

        buffer = io.BytesIO()
```

(continues on next page)

(continued from previous page)

```
        output_image.save(buffer, format=self.output_format.upper())

        return ImageArtifact(
            value=buffer.getvalue(),
            format=self.output_format.lower(),
            height=output_image.height,
            width=output_image.width,
            meta={"prompt": prompt},
        )

    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError("Inpainting is not supported by this driver.")

    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError("Outpainting is not supported by this driver.")
```

- `device = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output_format = field(default='png', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `pipeline_driver = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

try_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/huggingface_pipeline_image_generation_driver.py

```
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError("Inpainting is not supported by this driver.")
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/huggingface_pipeline_image_generation_driver.py

```
def try_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError("Outpainting is not supported by this driver.")
```

try_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/huggingface_pipeline_image_generation_driver.py

```
def try_image_variation(
    self, prompts: list[str], image: ImageArtifact, negative_prompts:␣
↪Optional[list[str]] = None
) -> ImageArtifact:
    pil_image = import_optional_dependency("PIL.Image")

    pipeline = self.pipeline_driver.prepare_pipeline(self.model, self.device)

    prompt = ", ".join(prompts)
    input_image = pil_image.open(io.BytesIO(image.value))
    # The size of the input image drives the size of the output image.
    # Resize the input image to the configured dimensions.
    output_width, output_height = self.pipeline_driver.output_image_dimensions
    if input_image.height != output_height or input_image.width != output_width:
        input_image = input_image.resize((output_width, output_height))

    output_image = pipeline(
        prompt,
        **self.pipeline_driver.make_image_param(input_image),
        **self.pipeline_driver.make_additional_params(negative_prompts, self.device),
    ).images[0]

    buffer = io.BytesIO()
    output_image.save(buffer, format=self.output_format.upper())

    return ImageArtifact(
        value=buffer.getvalue(),
        format=self.output_format.lower(),
        height=output_image.height,
        width=output_image.width,
        meta={"prompt": prompt},
    )
```

try_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/huggingface_pipeline_image_generation_driver.py

```
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    pipeline = self.pipeline_driver.prepare_pipeline(self.model, self.device)

    prompt = ", ".join(prompts)
    output_image = pipeline(
        prompt, **self.pipeline_driver.make_additional_params(negative_prompts, self.
↪device)
    ).images[0]

    buffer = io.BytesIO()
    output_image.save(buffer, format=self.output_format.upper())

    return ImageArtifact(
        value=buffer.getvalue(),
        format=self.output_format.lower(),
        height=output_image.height,
        width=output_image.width,
        meta={"prompt": prompt},
    )
```

HuggingFacePipelinePromptDriver

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.HuggingFacePipelinePromptDriver.model" title="model = field(kw_only=True, metadata={'serializable': True})">class-attribute instance-attribute (griptape.drivers.HuggingFacePipelinePromptDriver.model)"><code class="language-python">	*str*	Hugging Face Hub model name.

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
@define
class HuggingFacePipelinePromptDriver(BasePromptDriver):
    """Hugging Face Pipeline Prompt Driver.
```

(continues on next page)

(continued from previous page)

```
    Attributes:
        model: Hugging Face Hub model name.
    """

    max_tokens: int = field(default=250, kw_only=True, metadata={"serializable": True}
↪)
    model: str = field(kw_only=True, metadata={"serializable": True})
    tokenizer: HuggingFaceTokenizer = field(
        default=Factory(
            lambda self: HuggingFaceTokenizer(model=self.model, max_output_
↪tokens=self.max_tokens),
            takes_self=True,
        ),
        kw_only=True,
    )
    structured_output_strategy: StructuredOutputStrategy = field(
        default="rule", kw_only=True, metadata={"serializable": True}
    )
    _pipeline: Optional[TextGenerationPipeline] = field(
        default=None, kw_only=True, alias="pipeline", metadata={"serializable": False}
    )

    @structured_output_strategy.validator  # pyright:␣
↪ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
        if value in ("native", "tool"):
            raise ValueError(f"{__class__.__name__} does not support `{value}`␣
↪structured output strategy.")

        return value

    @lazy_property()
    def pipeline(self) -> TextGenerationPipeline:
        return import_optional_dependency("transformers").pipeline(
            task="text-generation",
            model=self.model,
            max_new_tokens=self.max_tokens,
            tokenizer=self.tokenizer.tokenizer,
        )

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        messages = self._prompt_stack_to_messages(prompt_stack)
        full_params = self._base_params(prompt_stack)
        logger.debug(
            (
                messages,
                full_params,
            )
        )
```

(continues on next page)

(continued from previous page)

```
        result = self.pipeline(messages, **full_params)
        logger.debug(result)

        if isinstance(result, list):
            if len(result) == 1:
                generated_text = result[0]["generated_text"][-1]["content"]

                input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))
                output_tokens = len(self.tokenizer.tokenizer.encode(generated_text))
↪# pyright: ignore[reportArgumentType]

                return Message(
                    content=[TextMessageContent(TextArtifact(generated_text))],
                    role=Message.ASSISTANT_ROLE,
                    usage=Message.Usage(input_tokens=input_tokens, output_
↪tokens=output_tokens),
                )
            raise Exception("completion with more than one choice is not supported yet
↪")
        raise Exception("invalid output format")

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        raise NotImplementedError("streaming is not supported")

    def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
        return self.tokenizer.tokenizer.decode(self.__prompt_stack_to_tokens(prompt_
↪stack))  # pyright: ignore[reportArgumentType]

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        return {
            "max_new_tokens": self.max_tokens,
            "temperature": self.temperature,
            "do_sample": True,
            **self.extra_params,
        }

    def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
        messages = []

        for message in prompt_stack.messages:
            messages.append({"role": message.role, "content": message.to_text()})

        return messages

    def __prompt_stack_to_tokens(self, prompt_stack: PromptStack) -> list[int]:
        messages = self._prompt_stack_to_messages(prompt_stack)
        tokens = self.tokenizer.tokenizer.apply_chat_template(messages, add_
↪generation_prompt=True, tokenize=True)

        if isinstance(tokens, list):
            return tokens  # pyright: ignore[reportReturnType] According to the␣
```

(continues on next page)

(continued from previous page)

```
↪[docs](https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils
↪#transformers.PreTrainedTokenizerBase.apply_chat_template), the return type is␣
↪List[int].
        raise ValueError("Invalid output type.")
```

- `_pipeline = field(default=None, kw_only=True, alias='pipeline', metadata={'serializable': False})` class-attribute instance-attribute
- `max_tokens = field(default=250, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='rule', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: HuggingFaceTokenizer(model=self.model, max_output_tokens=self.max_tokens), takes_self=True), kw_only=True)` class-attribute instance-attribute

__prompt_stack_to_tokens(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
def __prompt_stack_to_tokens(self, prompt_stack: PromptStack) -> list[int]:
    messages = self._prompt_stack_to_messages(prompt_stack)
    tokens = self.tokenizer.tokenizer.apply_chat_template(messages, add_generation_
↪prompt=True, tokenize=True)

    if isinstance(tokens, list):
        return tokens  # pyright: ignore[reportReturnType] According to the␣
↪[docs](https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils
↪#transformers.PreTrainedTokenizerBase.apply_chat_template), the return type is␣
↪List[int].
    raise ValueError("Invalid output type.")
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    return {
        "max_new_tokens": self.max_tokens,
        "temperature": self.temperature,
        "do_sample": True,
        **self.extra_params,
    }
```

_prompt_stack_to_messages(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
    messages = []

    for message in prompt_stack.messages:
        messages.append({"role": message.role, "content": message.to_text()})

    return messages
```

pipeline()

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
@lazy_property()
def pipeline(self) -> TextGenerationPipeline:
    return import_optional_dependency("transformers").pipeline(
        task="text-generation",
        model=self.model,
        max_new_tokens=self.max_tokens,
        tokenizer=self.tokenizer.tokenizer,
    )
```

prompt_stack_to_string(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
    return self.tokenizer.tokenizer.decode(self.__prompt_stack_to_tokens(prompt_
↪stack))  # pyright: ignore[reportArgumentType]
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    messages = self._prompt_stack_to_messages(prompt_stack)
    full_params = self._base_params(prompt_stack)
    logger.debug(
        (
            messages,
            full_params,
        )
    )

    result = self.pipeline(messages, **full_params)
    logger.debug(result)

    if isinstance(result, list):
        if len(result) == 1:
            generated_text = result[0]["generated_text"][-1]["content"]

            input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))
```

(continues on next page)

(continued from previous page)

```
            output_tokens = len(self.tokenizer.tokenizer.encode(generated_text))  #␣
↪pyright: ignore[reportArgumentType]

            return Message(
                content=[TextMessageContent(TextArtifact(generated_text))],
                role=Message.ASSISTANT_ROLE,
                usage=Message.Usage(input_tokens=input_tokens, output_tokens=output_
↪tokens),
            )
        raise Exception("completion with more than one choice is not supported yet")
    raise Exception("invalid output format")
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    raise NotImplementedError("streaming is not supported")
```

validate*structured_output_strategy(*, value)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
@structured_output_strategy.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
    if value in ("native", "tool"):
        raise ValueError(f"{__class__.__name__} does not support `{value}` structured␣
↪output strategy.")

    return value
```

LeonardoImageGenerationDriver

Bases: *BaseImageGenerationDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.BaseImageGenerationDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.LeonardoImageGenerationDriver.model)"><cod class="language-python">	*str*	The ID of the model to use when generating images.
<a href="./#griptape.drivers.LeonardoImageGenerationDriver.api_key" title="api_key = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.LeonardoImageGenerationDriver.api_key)"><co class="language-python">	*str*	The API key to use when making requests to the Leonardo API.
<a href="./#griptape.drivers.LeonardoImageGenerationDriver.requests_session" title="requests_session = field(default=Factory(lambda: requests.Session()), kw_only=True) class-attribute instance-attribute (griptape.drivers.LeonardoImageGenerationDriver.requests_session)"><code class="language-python">	*Session*	The requests session to use when making requests to the Leonardo API.

continues on next page

Table 80 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.LeonardoImageGenerationDriver.api_base" title="api_base = 'https://cloud.leonardo.ai/api/rest/v1' class-attribute instance-attribute (griptape.drivers.LeonardoImageGenerationDriver.api_base)"><c class="language-python">	*str*	The base URL of the Leonardo API.
<a href="./#griptape.drivers.LeonardoImageGenerationDriver.max_attempts" title="max_attempts = field(default=10, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.LeonardoImageGenerationDriver.max_attempts)"><code class="language-python">	*int*	The maximum number of times to poll the Leonardo API for a completed image.
<a href="./#griptape.drivers.LeonardoImageGenerationDriver.image_width" title="image_width = field(default=512, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.LeonardoImageGenerationDriver.image_width)"><code class="language-python">	*int*	The width of the generated image in the range [32, 1024] and divisible by 8.

continues on next page

Table 80 – continued from previous page

Name	Type	Description
<a href="./#grip-tape.drivers.LeonardoImageGenerationDriver.image_height" title="image_height = field(default=512, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (grip-tape.drivers.LeonardoImageGenerationDriver.image_height)"><code class="language-python">	*int*	The height of the generated image in the range [32, 1024] and divisible by 8.
<a href="./#grip-tape.drivers.LeonardoImageGenerationDriver.steps" title="steps = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (grip-tape.drivers.LeonardoImageGenerationDriver.steps)"><code class="language-python">	*Optional[int]*	Optionally specify the number of inference steps to run for each image generation request, [30, 60].
<a href="./#grip-tape.drivers.LeonardoImageGenerationDriver.seed" title="seed = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (grip-tape.drivers.LeonardoImageGenerationDriver.seed)"><code class="language-python">	*Optional[int]*	Optionally provide a consistent seed to generation requests, increasing consistency in output.

continues on next page

Table 80 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.LeonardoImageGenerationDriver.init_strength" title="init_strength = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.LeonardoImageGenerationDriver.init_strength)" class="language-python">	*Optional[float]*	Optionally specify the strength of the initial image, [0.0, 1.0].

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
@define
class LeonardoImageGenerationDriver(BaseImageGenerationDriver):
    """Driver for the Leonardo image generation API.

    Details on Leonardo image generation parameters can be found here:
    https://docs.leonardo.ai/reference/creategeneration

    Attributes:
        model: The ID of the model to use when generating images.
        api_key: The API key to use when making requests to the Leonardo API.
        requests_session: The requests session to use when making requests to the␣
↪Leonardo API.
        api_base: The base URL of the Leonardo API.
        max_attempts: The maximum number of times to poll the Leonardo API for a␣
↪completed image.
        image_width: The width of the generated image in the range [32, 1024] and␣
↪divisible by 8.
        image_height: The height of the generated image in the range [32, 1024] and␣
↪divisible by 8.
        steps: Optionally specify the number of inference steps to run for each image␣
↪generation request, [30, 60].
        seed: Optionally provide a consistent seed to generation requests, increasing␣
↪consistency in output.
        init_strength: Optionally specify the strength of the initial image, [0.0, 1.
↪0].
    """

    api_key: str = field(kw_only=True, metadata={"serializable": True})
    requests_session: requests.Session = field(default=Factory(lambda: requests.
↪Session()), kw_only=True)
    api_base: str = "https://cloud.leonardo.ai/api/rest/v1"
    max_attempts: int = field(default=10, kw_only=True, metadata={"serializable":␣
↪True})
```

(continues on next page)

(continued from previous page)

```
    image_width: int = field(default=512, kw_only=True, metadata={"serializable":␣
↪True})
    image_height: int = field(default=512, kw_only=True, metadata={"serializable":␣
↪True})
    steps: Optional[int] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    seed: Optional[int] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    init_strength: Optional[float] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    control_net: bool = field(default=False, kw_only=True, metadata={"serializable":␣
↪True})
    control_net_type: Optional[Literal["POSE", "CANNY", "DEPTH"]] = field(
        default=None,
        kw_only=True,
        metadata={"serializable": True},
    )

    def try_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact:
        if negative_prompts is None:
            negative_prompts = []

        generation_id = self._create_generation(prompts=prompts, negative_
↪prompts=negative_prompts)
        image_url = self._get_image_url(generation_id=generation_id)
        image_data = self._download_image(url=image_url)

        return ImageArtifact(
            value=image_data,
            format="png",
            width=self.image_width,
            height=self.image_height,
            meta={
                "model": self.model,
                "prompt": ", ".join(prompts),
            },
        )

    def try_image_variation(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        if negative_prompts is None:
            negative_prompts = []

        init_image_id = self._upload_init_image(image)
        generation_id = self._create_generation(
            prompts=prompts,
            negative_prompts=negative_prompts,
```

(continues on next page)

(continued from previous page)

```
            init_image_id=init_image_id,
        )
        image_url = self._get_image_url(generation_id=generation_id)
        image_data = self._download_image(url=image_url)

        return ImageArtifact(
            value=image_data,
            format="png",
            width=self.image_width,
            height=self.image_height,
            meta={
                "model": self.model,
                "prompt": ", ".join(prompts),
            },
        )

    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪outpainting")

    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪inpainting")

    def _upload_init_image(self, image: ImageArtifact) -> str:
        request = {"extension": image.mime_type.split("/")[1]}

        prep_response = self._make_api_request("/init-image", request=request)
        if prep_response is None or prep_response["uploadInitImage"] is None:
            raise Exception(f"failed to prepare init image: {prep_response}")

        fields = json.loads(prep_response["uploadInitImage"]["fields"])
        pre_signed_url = prep_response["uploadInitImage"]["url"]
        init_image_id = prep_response["uploadInitImage"]["id"]

        files = {"file": image.value}
        upload_response = requests.post(pre_signed_url, data=fields, files=files)
        if not upload_response.ok:
            raise Exception(f"failed to upload init image: {upload_response.text}")
```

(continues on next page)

(continued from previous page)

```
        return init_image_id

    def _create_generation(
        self,
        prompts: list[str],
        negative_prompts: list[str],
        init_image_id: Optional[str] = None,
    ) -> str:
        prompt = ", ".join(prompts)
        negative_prompt = ", ".join(negative_prompts)
        request = {
            "prompt": prompt,
            "negative_prompt": negative_prompt,
            "width": self.image_width,
            "height": self.image_height,
            "num_images": 1,
            "modelId": self.model,
        }

        if init_image_id is not None:
            request["init_image_id"] = init_image_id

        if self.init_strength is not None:
            request["init_strength"] = self.init_strength

        if self.steps:
            request["num_inference_steps"] = self.steps

        if self.seed is not None:
            request["seed"] = self.seed

        if self.control_net:
            request["controlNet"] = self.control_net
            request["controlNetType"] = self.control_net_type

        response = self._make_api_request("/generations", request=request)
        if response is None or response["sdGenerationJob"] is None:
            raise Exception(f"failed to create generation: {response}")

        return response["sdGenerationJob"]["generationId"]

    def _make_api_request(self, endpoint: str, request: dict, method: str = "POST") ->
↪ dict:
        url = f"{self.api_base}{endpoint}"
        headers = {"Authorization": f"Bearer {self.api_key}"}

        response = self.requests_session.request(url=url, method=method, json=request,
↪ headers=headers)
        if not response.ok:
            raise Exception(f"failed to make API request: {response.text}")

        return response.json()
```

(continues on next page)

(continued from previous page)

```
    def _get_image_url(self, generation_id: str) -> str:
        for attempt in range(self.max_attempts):
            response = self.requests_session.get(
                url=f"{self.api_base}/generations/{generation_id}",
                headers={"Authorization": f"Bearer {self.api_key}"},
            ).json()

            if response["generations_by_pk"]["status"] == "PENDING":
                time.sleep(attempt + 1)
                continue

            return response["generations_by_pk"]["generated_images"][0]["url"]
        raise Exception("image generation failed to complete")

    def _download_image(self, url: str) -> bytes:
        response = self.requests_session.get(url=url, headers={"Authorization": f
↪"Bearer {self.api_key}"})

        return response.content
```

- `api_base = 'https://cloud.leonardo.ai/api/rest/v1'` class-attribute instance-attribute
- `api_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `control_net = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `control_net_type = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `image_height = field(default=512, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `image_width = field(default=512, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `init_strength = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `max_attempts = field(default=10, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `requests_session = field(default=Factory(lambda: requests.Session()), kw_only=True)` class-attribute instance-attribute
- `seed = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `steps = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_create_generation(prompts, negative_prompts, init_image_id=None)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def _create_generation(
    self,
    prompts: list[str],
    negative_prompts: list[str],
    init_image_id: Optional[str] = None,
) -> str:
    prompt = ", ".join(prompts)
    negative_prompt = ", ".join(negative_prompts)
    request = {
        "prompt": prompt,
        "negative_prompt": negative_prompt,
        "width": self.image_width,
        "height": self.image_height,
        "num_images": 1,
        "modelId": self.model,
    }

    if init_image_id is not None:
        request["init_image_id"] = init_image_id

    if self.init_strength is not None:
        request["init_strength"] = self.init_strength

    if self.steps:
        request["num_inference_steps"] = self.steps

    if self.seed is not None:
        request["seed"] = self.seed

    if self.control_net:
        request["controlNet"] = self.control_net
        request["controlNetType"] = self.control_net_type

    response = self._make_api_request("/generations", request=request)
    if response is None or response["sdGenerationJob"] is None:
        raise Exception(f"failed to create generation: {response}")

    return response["sdGenerationJob"]["generationId"]
```

_download_image(url)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def _download_image(self, url: str) -> bytes:
    response = self.requests_session.get(url=url, headers={"Authorization": f"Bearer
↪{self.api_key}"})

    return response.content
```

_get_image_url(generation_id)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def _get_image_url(self, generation_id: str) -> str:
    for attempt in range(self.max_attempts):
        response = self.requests_session.get(
            url=f"{self.api_base}/generations/{generation_id}",
            headers={"Authorization": f"Bearer {self.api_key}"},
        ).json()

        if response["generations_by_pk"]["status"] == "PENDING":
            time.sleep(attempt + 1)
            continue

        return response["generations_by_pk"]["generated_images"][0]["url"]
    raise Exception("image generation failed to complete")
```

_make_api_request(endpoint, request, method='POST')

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def _make_api_request(self, endpoint: str, request: dict, method: str = "POST") ->␣
↪dict:
    url = f"{self.api_base}{endpoint}"
    headers = {"Authorization": f"Bearer {self.api_key}"}

    response = self.requests_session.request(url=url, method=method, json=request,␣
↪headers=headers)
    if not response.ok:
        raise Exception(f"failed to make API request: {response.text}")

    return response.json()
```

_upload_init_image(image)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def _upload_init_image(self, image: ImageArtifact) -> str:
    request = {"extension": image.mime_type.split("/")[1]}

    prep_response = self._make_api_request("/init-image", request=request)
    if prep_response is None or prep_response["uploadInitImage"] is None:
        raise Exception(f"failed to prepare init image: {prep_response}")

    fields = json.loads(prep_response["uploadInitImage"]["fields"])
    pre_signed_url = prep_response["uploadInitImage"]["url"]
    init_image_id = prep_response["uploadInitImage"]["id"]

    files = {"file": image.value}
    upload_response = requests.post(pre_signed_url, data=fields, files=files)
    if not upload_response.ok:
        raise Exception(f"failed to upload init image: {upload_response.text}")
```

(continues on next page)

(continued from previous page)

```
    return init_image_id
```

try_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support inpainting
↪")
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def try_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support outpainting
↪")
```

try_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def try_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    if negative_prompts is None:
        negative_prompts = []

    init_image_id = self._upload_init_image(image)
    generation_id = self._create_generation(
        prompts=prompts,
        negative_prompts=negative_prompts,
        init_image_id=init_image_id,
    )
    image_url = self._get_image_url(generation_id=generation_id)
    image_data = self._download_image(url=image_url)
```

(continues on next page)

(continued from previous page)

```
    return ImageArtifact(
        value=image_data,
        format="png",
        width=self.image_width,
        height=self.image_height,
        meta={
            "model": self.model,
            "prompt": ", ".join(prompts),
        },
    )
```

try_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    if negative_prompts is None:
        negative_prompts = []

    generation_id = self._create_generation(prompts=prompts, negative_
↪prompts=negative_prompts)
    image_url = self._get_image_url(generation_id=generation_id)
    image_data = self._download_image(url=image_url)

    return ImageArtifact(
        value=image_data,
        format="png",
        width=self.image_width,
        height=self.image_height,
        meta={
            "model": self.model,
            "prompt": ", ".join(prompts),
        },
    )
```

LocalConversationMemoryDriver

Bases: *BaseConversationMemoryDriver* “)

Source Code in griptape/drivers/memory/conversation/local_conversation_memory_driver.py

```
@define(kw_only=True)
class LocalConversationMemoryDriver(BaseConversationMemoryDriver):
    persist_file: Optional[str] = field(default=None, metadata={"serializable": True})

    def store(self, runs: list[Run], metadata: dict[str, Any]) -> None:
        if self.persist_file is not None:
            Path(self.persist_file).write_text(json.dumps(self._to_params_dict(runs,␣
↪metadata)))
```

(continues on next page)

(continued from previous page)

```
    def load(self) -> tuple[list[Run], dict[str, Any]]:
        if (
            self.persist_file is not None
            and os.path.exists(self.persist_file)
            and (loaded_str := Path(self.persist_file).read_text()) is not None
        ):
            try:
                return self._from_params_dict(json.loads(loaded_str))
            except Exception as e:
                raise ValueError(f"Unable to load data from {self.persist_file}")␣
↪from e

        return [], {}
```

- `persist_file = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

load()

Source Code in griptape/drivers/memory/conversation/local_conversation_memory_driver.py

```
def load(self) -> tuple[list[Run], dict[str, Any]]:
    if (
        self.persist_file is not None
        and os.path.exists(self.persist_file)
        and (loaded_str := Path(self.persist_file).read_text()) is not None
    ):
        try:
            return self._from_params_dict(json.loads(loaded_str))
        except Exception as e:
            raise ValueError(f"Unable to load data from {self.persist_file}") from e

    return [], {}
```

store(runs, metadata)

Source Code in griptape/drivers/memory/conversation/local_conversation_memory_driver.py

```
def store(self, runs: list[Run], metadata: dict[str, Any]) -> None:
    if self.persist_file is not None:
        Path(self.persist_file).write_text(json.dumps(self._to_params_dict(runs,␣
↪metadata)))
```

LocalFileManagerDriver

Bases: *BaseFileManagerDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.LocalFileManagerDriver.workdir" title="workdir property writable (griptape.drivers.LocalFileManagerDriver.workdir)"><code class="language-python">	*str*	The working directory as an absolute path. List, load, and save operations will be performed relative to this directory. Defaults to the current working directory. Setting this to None will disable the working directory and all paths will be treated as absolute paths.

Source Code in griptape/drivers/file_manager/local_file_manager_driver.py

```
@define
class LocalFileManagerDriver(BaseFileManagerDriver):
    """LocalFileManagerDriver can be used to list, load, and save files on the local␣
↪file system.

    Attributes:
        workdir: The working directory as an absolute path. List, load, and save␣
↪operations will be performed relative to this directory.
                 Defaults to the current working directory.
                 Setting this to None will disable the working directory and all␣
↪paths will be treated as absolute paths.
    """

    _workdir: str = field(default=Factory(lambda: os.getcwd()), kw_only=True, alias=
↪"workdir")

    @property
    def workdir(self) -> str:
        if os.path.isabs(self._workdir):
            return self._workdir
        return os.path.join(os.getcwd(), self._workdir)

    @workdir.setter
    def workdir(self, value: str) -> None:
        self._workdir = value

    def try_list_files(self, path: str) -> list[str]:
        full_path = self._full_path(path)
        return os.listdir(full_path)

    def try_load_file(self, path: str) -> bytes:
        full_path = self._full_path(path)
        if self._is_dir(full_path):
            raise IsADirectoryError
        return Path(full_path).read_bytes()

    def try_save_file(self, path: str, value: bytes) -> str:
        full_path = self._full_path(path)
```

(continues on next page)

(continued from previous page)

```
        if self._is_dir(full_path):
            raise IsADirectoryError
        os.makedirs(os.path.dirname(full_path), exist_ok=True)
        Path(full_path).write_bytes(value)
        return full_path

    def _full_path(self, path: str) -> str:
        full_path = path if os.path.isabs(path) else os.path.join(self.workdir, path.
↪lstrip("/"))
        # Need to keep the trailing slash if it was there,
        # because it means the path is a directory.
        ended_with_sep = path.endswith("/")
        full_path = os.path.normpath(full_path)
        if ended_with_sep:
            full_path = full_path.rstrip("/") + "/"
        return full_path

    def _is_dir(self, full_path: str) -> bool:
        return full_path.endswith("/") or Path(full_path).is_dir()
```

- `_workdir = field(default=Factory(lambda: os.getcwd()), kw_only=True, alias='workdir')` class-attribute instance-attribute
- `workdir` property writable

_full_path(path)

Source Code in griptape/drivers/file_manager/local_file_manager_driver.py

```
def _full_path(self, path: str) -> str:
    full_path = path if os.path.isabs(path) else os.path.join(self.workdir, path.
↪lstrip("/"))
    # Need to keep the trailing slash if it was there,
    # because it means the path is a directory.
    ended_with_sep = path.endswith("/")
    full_path = os.path.normpath(full_path)
    if ended_with_sep:
        full_path = full_path.rstrip("/") + "/"
    return full_path
```

_is_dir(full_path)

Source Code in griptape/drivers/file_manager/local_file_manager_driver.py

```
def _is_dir(self, full_path: str) -> bool:
    return full_path.endswith("/") or Path(full_path).is_dir()
```

try_list_files(path)

Source Code in griptape/drivers/file_manager/local_file_manager_driver.py

```
def try_list_files(self, path: str) -> list[str]:
    full_path = self._full_path(path)
    return os.listdir(full_path)
```

try_load_file(path)

Source Code in griptape/drivers/file_manager/local_file_manager_driver.py

```
def try_load_file(self, path: str) -> bytes:
    full_path = self._full_path(path)
    if self._is_dir(full_path):
        raise IsADirectoryError
    return Path(full_path).read_bytes()
```

try_save_file(path, value)

Source Code in griptape/drivers/file_manager/local_file_manager_driver.py

```
def try_save_file(self, path: str, value: bytes) -> str:
    full_path = self._full_path(path)
    if self._is_dir(full_path):
        raise IsADirectoryError
    os.makedirs(os.path.dirname(full_path), exist_ok=True)
    Path(full_path).write_bytes(value)
    return full_path
```

LocalRerankDriver

Bases: *BaseRerankDriver* “) , *FuturesExecutorMixin* ”)

Source Code in griptape/drivers/rerank/local_rerank_driver.py

```
@define(kw_only=True)
class LocalRerankDriver(BaseRerankDriver, FuturesExecutorMixin):
    calculate_relatedness: Callable = field(default=lambda x, y: dot(x, y) / (norm(x)␣
↪* norm(y)))
    embedding_driver: BaseEmbeddingDriver = field(
        kw_only=True, default=Factory(lambda: Defaults.drivers_config.embedding_
↪driver), metadata={"serializable": True}
    )

    def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]:
        query_embedding = self.embedding_driver.embed(query, vector_operation="query")

        with self.create_futures_executor() as futures_executor:
            artifact_embeddings = execute_futures_list(
                [
                    futures_executor.submit(
                        with_contextvars(self.embedding_driver.embed_text_artifact),␣
↪a, vector_operation="upsert"
                    )
                    for a in artifacts
                ],
```

(continues on next page)

(continued from previous page)

```
            )

        artifacts_and_relatednesses = [
            (artifact, self.calculate_relatedness(query_embedding, artifact_
↪embedding))
            for artifact, artifact_embedding in zip(artifacts, artifact_embeddings)
        ]

        artifacts_and_relatednesses.sort(key=operator.itemgetter(1), reverse=True)

        return [artifact for artifact, _ in artifacts_and_relatednesses]
```

- `calculate_relatedness = field(default=lambda x, y: dot(x, y) / norm(x) * norm(y))` class-attribute instance-attribute
- `embedding_driver = field(kw_only=True, default=Factory(lambda: Defaults.drivers_config.embedding_driver), metadata={'serializable': True})` class-attribute instance-attribute

run(query, artifacts)

Source Code in griptape/drivers/rerank/local_rerank_driver.py

```
def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]:
    query_embedding = self.embedding_driver.embed(query, vector_operation="query")

    with self.create_futures_executor() as futures_executor:
        artifact_embeddings = execute_futures_list(
            [
                futures_executor.submit(
                    with_contextvars(self.embedding_driver.embed_text_artifact), a,␣
↪vector_operation="upsert"
                )
                for a in artifacts
            ],
        )

    artifacts_and_relatednesses = [
        (artifact, self.calculate_relatedness(query_embedding, artifact_embedding))
        for artifact, artifact_embedding in zip(artifacts, artifact_embeddings)
    ]

    artifacts_and_relatednesses.sort(key=operator.itemgetter(1), reverse=True)

    return [artifact for artifact, _ in artifacts_and_relatednesses]
```

LocalRulesetDriver

Bases: *BaseRulesetDriver* “)

Source Code in griptape/drivers/ruleset/local_ruleset_driver.py

```
@define(kw_only=True)
class LocalRulesetDriver(BaseRulesetDriver):
    persist_dir: Optional[str] = field(default=None, metadata={"serializable": True})

    def load(self, ruleset_name: str) -> tuple[list[BaseRule], dict[str, Any]]:
        if self.persist_dir is None:
            return [], {}

        file_name = os.path.join(self.persist_dir, ruleset_name)

        if (
            file_name is not None
            and os.path.exists(file_name)
            and (loaded_str := Path(file_name).read_text()) is not None
        ):
            try:
                return self._from_ruleset_dict(json.loads(loaded_str))
            except Exception as e:
                raise ValueError(f"Unable to load data from {file_name}") from e

        if self.raise_not_found:
            raise ValueError(f"Ruleset not found with name {file_name}")
        return [], {}
```

- `persist_dir = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

load(ruleset_name)

Source Code in griptape/drivers/ruleset/local_ruleset_driver.py

```
def load(self, ruleset_name: str) -> tuple[list[BaseRule], dict[str, Any]]:
    if self.persist_dir is None:
        return [], {}

    file_name = os.path.join(self.persist_dir, ruleset_name)

    if (
        file_name is not None
        and os.path.exists(file_name)
        and (loaded_str := Path(file_name).read_text()) is not None
    ):
        try:
            return self._from_ruleset_dict(json.loads(loaded_str))
        except Exception as e:
            raise ValueError(f"Unable to load data from {file_name}") from e

    if self.raise_not_found:
        raise ValueError(f"Ruleset not found with name {file_name}")
    return [], {}
```

LocalStructureRunDriver

Bases: *BaseStructureRunDriver* “)

Source Code in griptape/drivers/structure_run/local_structure_run_driver.py

```
@define
class LocalStructureRunDriver(BaseStructureRunDriver):
    create_structure: Callable[[], Structure] = field(kw_only=True)

    def try_run(self, *args: BaseArtifact) -> BaseArtifact:
        old_env = os.environ.copy()
        try:
            os.environ.update(self.env)
            structure = self.create_structure().run(*[arg.value for arg in args])
        finally:
            os.environ.clear()
            os.environ.update(old_env)

        return structure.output
```

- `create_structure = field(kw_only=True)` class-attribute instance-attribute

try_run(*args)

Source Code in griptape/drivers/structure_run/local_structure_run_driver.py

```
def try_run(self, *args: BaseArtifact) -> BaseArtifact:
    old_env = os.environ.copy()
    try:
        os.environ.update(self.env)
        structure = self.create_structure().run(*[arg.value for arg in args])
    finally:
        os.environ.clear()
        os.environ.update(old_env)

    return structure.output
```

LocalVectorStoreDriver

Bases: *BaseVectorStoreDriver* “)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
@define(kw_only=True)
class LocalVectorStoreDriver(BaseVectorStoreDriver):
    entries: dict[str, BaseVectorStoreDriver.Entry] = field(factory=dict)
    persist_file: Optional[str] = field(default=None)
    calculate_relatedness: Callable = field(default=lambda x, y: dot(x, y) / (norm(x)␣
↪* norm(y)))
    thread_lock: threading.Lock = field(default=Factory(lambda: threading.Lock()))

    def __attrs_post_init__(self) -> None:
        if self.persist_file is not None:
            directory = os.path.dirname(self.persist_file)
```

(continues on next page)

(continued from previous page)

```
            if directory and not os.path.exists(directory):
                os.makedirs(directory)

            if not os.path.isfile(self.persist_file):
                with open(self.persist_file, "w") as file:
                    self.__save_entries_to_file(file)

            with open(self.persist_file, "r+") as file:
                if os.path.getsize(self.persist_file) > 0:
                    self.entries = self.load_entries_from_file(file)
                else:
                    self.__save_entries_to_file(file)

    def load_entries_from_file(self, json_file: TextIO) -> dict[str,␣
↪BaseVectorStoreDriver.Entry]:
        with self.thread_lock:
            data = json.load(json_file)

            return {k: BaseVectorStoreDriver.Entry.from_dict(v) for k, v in data.
↪items()}

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        vector_id = vector_id or utils.str_to_hash(str(vector))

        with self.thread_lock:
            self.entries[self.__namespaced_vector_id(vector_id, namespace=namespace)]␣
↪= self.Entry(
                id=vector_id,
                vector=vector,
                meta=meta,
                namespace=namespace,
            )

        if self.persist_file is not None:
            # TODO: optimize later since it reserializes all entries from memory and␣
↪stores them in the JSON file
            #  every time a new vector is inserted
            with open(self.persist_file, "w") as file:
                self.__save_entries_to_file(file)

        return vector_id

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
```

(continues on next page)

(continued from previous page)

```
↪Optional[BaseVectorStoreDriver.Entry]:
        return self.entries.get(self.__namespaced_vector_id(vector_id,␣
↪namespace=namespace), None)

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        return [entry for key, entry in self.entries.items() if namespace is None or␣
↪entry.namespace == namespace]

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        if namespace:
            entries = {k: v for (k, v) in self.entries.items() if k.startswith(f"
↪{namespace}-")}
        else:
            entries = self.entries

        entries_and_relatednesses = [
            (entry, self.calculate_relatedness(vector, entry.vector)) for entry in␣
↪list(entries.values())
        ]

        entries_and_relatednesses.sort(key=operator.itemgetter(1), reverse=True)

        result = [
            BaseVectorStoreDriver.Entry(
                id=er[0].id, vector=er[0].vector, score=er[1], meta=er[0].meta,␣
↪namespace=er[0].namespace
            )
            for er in entries_and_relatednesses
        ][:count]

        if include_vectors:
            return result
        return [
            BaseVectorStoreDriver.Entry(id=r.id, vector=[], score=r.score, meta=r.
↪meta, namespace=r.namespace)
            for r in result
        ]

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")

    def __save_entries_to_file(self, json_file: TextIO) -> None:
```

(continues on next page)

(continued from previous page)

```
        with self.thread_lock:
            serialized_data = {k: v.to_dict() for k, v in self.entries.items()}

            json.dump(serialized_data, json_file)

    def __namespaced_vector_id(self, vector_id: str, *, namespace: Optional[str]) ->␣
↪str:
        return vector_id if namespace is None else f"{namespace}-{vector_id}"
```

- `calculate_relatedness = field(default=lambda x, y: dot(x, y) / norm(x) * norm(y))` class-attribute instance-attribute
- `entries = field(factory=dict)` class-attribute instance-attribute
- `persist_file = field(default=None)` class-attribute instance-attribute
- `thread_lock = field(default=Factory(lambda: threading.Lock()))` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def __attrs_post_init__(self) -> None:
    if self.persist_file is not None:
        directory = os.path.dirname(self.persist_file)

        if directory and not os.path.exists(directory):
            os.makedirs(directory)

        if not os.path.isfile(self.persist_file):
            with open(self.persist_file, "w") as file:
                self.__save_entries_to_file(file)

        with open(self.persist_file, "r+") as file:
            if os.path.getsize(self.persist_file) > 0:
                self.entries = self.load_entries_from_file(file)
            else:
                self.__save_entries_to_file(file)
```

__namespaced_vector_id(vector_id, *, namespace)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def __namespaced_vector_id(self, vector_id: str, *, namespace: Optional[str]) -> str:
    return vector_id if namespace is None else f"{namespace}-{vector_id}"
```

__save_entries_to_file(json_file)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def __save_entries_to_file(self, json_file: TextIO) -> None:
    with self.thread_lock:
        serialized_data = {k: v.to_dict() for k, v in self.entries.items()}
```

(continues on next page)

(continued from previous page)

```
        json.dump(serialized_data, json_file)
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    return [entry for key, entry in self.entries.items() if namespace is None or␣
↪entry.namespace == namespace]
```

load_entries_from_file(json_file)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def load_entries_from_file(self, json_file: TextIO) -> dict[str,␣
↪BaseVectorStoreDriver.Entry]:
    with self.thread_lock:
        data = json.load(json_file)

        return {k: BaseVectorStoreDriver.Entry.from_dict(v) for k, v in data.items()}
```

load_entry(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    return self.entries.get(self.__namespaced_vector_id(vector_id,␣
↪namespace=namespace), None)
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
```

(continues on next page)

(continued from previous page)

```
    if namespace:
        entries = {k: v for (k, v) in self.entries.items() if k.startswith(f"
↪{namespace}-")}
    else:
        entries = self.entries

    entries_and_relatednesses = [
        (entry, self.calculate_relatedness(vector, entry.vector)) for entry in␣
↪list(entries.values())
    ]

    entries_and_relatednesses.sort(key=operator.itemgetter(1), reverse=True)

    result = [
        BaseVectorStoreDriver.Entry(
            id=er[0].id, vector=er[0].vector, score=er[1], meta=er[0].meta,␣
↪namespace=er[0].namespace
        )
        for er in entries_and_relatednesses
    ][:count]

    if include_vectors:
        return result
    return [
        BaseVectorStoreDriver.Entry(id=r.id, vector=[], score=r.score, meta=r.meta,␣
↪namespace=r.namespace)
        for r in result
    ]
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    vector_id = vector_id or utils.str_to_hash(str(vector))

    with self.thread_lock:
        self.entries[self.__namespaced_vector_id(vector_id, namespace=namespace)] =␣
↪self.Entry(
            id=vector_id,
            vector=vector,
            meta=meta,
            namespace=namespace,
        )
```

(continues on next page)

(continued from previous page)

```
    if self.persist_file is not None:
        # TODO: optimize later since it reserializes all entries from memory and␣
↪stores them in the JSON file
        #  every time a new vector is inserted
        with open(self.persist_file, "w") as file:
            self.__save_entries_to_file(file)

    return vector_id
```

MarkdownifyWebScraperDriver

Bases: *BaseWebScraperDriver* “)

Attributes

Name	Type	Description
<code class="language-python">	*bool*	If *True*, the driver will include link urls in the markdown output.
<code class="language-python">	<code class="language-python">list[str]</code>	Optionally provide custom tags to exclude from the scraped content.

continues on next page

Table 82 – continued from previous page

Name	Type	Description
<code class="language-python">	<code class="language-python">list[str]</code>	Optionally provide custom classes to exclude from the scraped content.
<code class="language-python">	<code class="language-python">list[str]</code>	Optionally provide custom ids to exclude from the scraped content.
<a href="./#griptape.drivers.MarkdownifyWebScraperDriver.timeout" title="timeout = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.drivers.MarkdownifyWebScraperDriver.timeout)"><code class="language-python">	*Optional[int]*	Optionally provide a timeout in milliseconds for the page to continue loading after the browser has emitted the "load" event.

Source Code in griptape/drivers/web_scraper/markdownify_web_scraper_driver.py

```
@define
class MarkdownifyWebScraperDriver(BaseWebScraperDriver):
    """Driver to scrape a webpage and return the content in markdown format.
```

(continues on next page)

(continued from previous page)

```
    As a prerequisite to using MarkdownifyWebScraperDriver, you need to install the␣
↪browsers used by
    playwright. You can do this by running: `poetry run playwright install`.
    For more details about playwright, see https://playwright.dev/python/docs/library.

    Attributes:
        include_links: If `True`, the driver will include link urls in the markdown␣
↪output.
        exclude_tags: Optionally provide custom tags to exclude from the scraped␣
↪content.
        exclude_classes: Optionally provide custom classes to exclude from the␣
↪scraped content.
        exclude_ids: Optionally provide custom ids to exclude from the scraped␣
↪content.
        timeout: Optionally provide a timeout in milliseconds for the page to␣
↪continue loading after
            the browser has emitted the "load" event.
    """

    DEFAULT_EXCLUDE_TAGS = ["script", "style", "head", "audio", "img", "picture",
↪"source", "video"]

    include_links: bool = field(default=True, kw_only=True)
    exclude_tags: list[str] = field(
        default=Factory(lambda self: self.DEFAULT_EXCLUDE_TAGS, takes_self=True),
        kw_only=True,
    )
    exclude_classes: list[str] = field(default=Factory(list), kw_only=True)
    exclude_ids: list[str] = field(default=Factory(list), kw_only=True)
    timeout: Optional[int] = field(default=None, kw_only=True)

    def fetch_url(self, url: str) -> str:
        sync_playwright = import_optional_dependency("playwright.sync_api").sync_
↪playwright

        with sync_playwright() as p, p.chromium.launch(headless=True) as browser:
            page = browser.new_page()

            def skip_loading_images(route: Any) -> Any:
                if route.request.resource_type == "image":
                    return route.abort()
                route.continue_()
                return None

            page.route("**/*", skip_loading_images)

            page.goto(url)

            # Some websites require a delay before the content is fully loaded
            # even after the browser has emitted "load" event.
            if self.timeout:
                page.wait_for_timeout(self.timeout)
```

(continues on next page)

(continued from previous page)

```
            content = page.content()

            if not content:
                raise Exception("cant access URL")

            return content

    def extract_page(self, page: str) -> TextArtifact:
        bs4 = import_optional_dependency("bs4")
        markdownify = import_optional_dependency("markdownify")
        include_links = self.include_links

        # Custom MarkdownConverter to optionally linked urls. If include_links is␣
↪False only
        # the text of the link is returned.
        class OptionalLinksMarkdownConverter(markdownify.MarkdownConverter):
            def convert_a(self, el: Any, text: str, parent_tags: Any) -> str:
                if include_links:
                    return super().convert_a(el, text, parent_tags)
                return text

        soup = bs4.BeautifulSoup(page, "html.parser")

        # Remove unwanted elements
        exclude_selector = ",".join(
            self.exclude_tags + [f".{c}" for c in self.exclude_classes] + [f"#{i}"␣
↪for i in self.exclude_ids],
        )
        if exclude_selector:
            for s in soup.select(exclude_selector):
                s.extract()

        text = OptionalLinksMarkdownConverter().convert_soup(soup)

        # Remove leading and trailing whitespace from the entire text
        text = text.strip()

        # Remove trailing whitespace from each line
        text = re.sub(r"[ \t]+$", "", text, flags=re.MULTILINE)

        # Indent using 2 spaces instead of tabs
        text = re.sub(r"(\n?\s*?)\t", r"  ", text)

        # Remove triple+ newlines (keep double newlines for paragraphs)
        text = re.sub(r"\n\n+", "\n\n", text)

        return TextArtifact(text)
```

- `DEFAULT_EXCLUDE_TAGS = ['script', 'style', 'head', 'audio', 'img', 'picture', 'source', 'video']` class-attribute instance-attribute
- `exclude_classes = field(default=Factory(list), kw_only=True)` class-attribute

instance-attribute

- `exclude_ids = field(default=Factory(list), kw_only=True)` class-attribute instance-attribute
- `exclude_tags = field(default=Factory(lambda self: self.DEFAULT_EXCLUDE_TAGS, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `include_links = field(default=True, kw_only=True)` class-attribute instance-attribute
- `timeout = field(default=None, kw_only=True)` class-attribute instance-attribute

extract_page(page)

Source Code in griptape/drivers/web_scraper/markdownify_web_scraper_driver.py

```
def extract_page(self, page: str) -> TextArtifact:
    bs4 = import_optional_dependency("bs4")
    markdownify = import_optional_dependency("markdownify")
    include_links = self.include_links

    # Custom MarkdownConverter to optionally linked urls. If include_links is False␣
↪only
    # the text of the link is returned.
    class OptionalLinksMarkdownConverter(markdownify.MarkdownConverter):
        def convert_a(self, el: Any, text: str, parent_tags: Any) -> str:
            if include_links:
                return super().convert_a(el, text, parent_tags)
            return text

    soup = bs4.BeautifulSoup(page, "html.parser")

    # Remove unwanted elements
    exclude_selector = ",".join(
        self.exclude_tags + [f".{c}" for c in self.exclude_classes] + [f"#{i}" for i␣
↪in self.exclude_ids],
    )
    if exclude_selector:
        for s in soup.select(exclude_selector):
            s.extract()

    text = OptionalLinksMarkdownConverter().convert_soup(soup)

    # Remove leading and trailing whitespace from the entire text
    text = text.strip()

    # Remove trailing whitespace from each line
    text = re.sub(r"[ \t]+$", "", text, flags=re.MULTILINE)

    # Indent using 2 spaces instead of tabs
    text = re.sub(r"(\n?\s*?)\t", r"  ", text)

    # Remove triple+ newlines (keep double newlines for paragraphs)
    text = re.sub(r"\n\n+", "\n\n", text)

    return TextArtifact(text)
```

fetch_url(url)

Source Code in griptape/drivers/web_scraper/markdownify_web_scraper_driver.py

```
def fetch_url(self, url: str) -> str:
    sync_playwright = import_optional_dependency("playwright.sync_api").sync_
↪playwright

    with sync_playwright() as p, p.chromium.launch(headless=True) as browser:
        page = browser.new_page()

        def skip_loading_images(route: Any) -> Any:
            if route.request.resource_type == "image":
                return route.abort()
            route.continue_()
            return None

        page.route("**/*", skip_loading_images)

        page.goto(url)

        # Some websites require a delay before the content is fully loaded
        # even after the browser has emitted "load" event.
        if self.timeout:
            page.wait_for_timeout(self.timeout)

        content = page.content()

        if not content:
            raise Exception("cant access URL")

        return content
```

MarqoVectorStoreDriver

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.MarqoVectorStoreDriver.api_key" title="api_key = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.MarqoVectorStoreDriver.api_key)"><c class="language-python">	*str*	The API key for the Marqo API.
<a href="./#griptape.drivers.MarqoVectorStoreDriver.url" title="url = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.MarqoVectorStoreDriver.url)"><code class="language-python">	*str*	The URL to the Marqo API.
<a href="./#griptape.drivers.MarqoVectorStoreDriver.client" title="client() (griptape.drivers.MarqoVectorStoreDriver.client)"><coc class="language-python">	*Client*	An optional Marqo client. Defaults to a new client with the given URL and API key.
<a href="./#griptape.drivers.MarqoVectorStoreDriver.index" title="index = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.MarqoVectorStoreDriver.index)"><code class="language-python">	*str*	The name of the index to use.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
@define
```

(continues on next page)

(continued from previous page)

```
class MarqoVectorStoreDriver(BaseVectorStoreDriver):
    """A Vector Store Driver for Marqo.

    Attributes:
        api_key: The API key for the Marqo API.
        url: The URL to the Marqo API.
        client: An optional Marqo client. Defaults to a new client with the given URL␣
↪and API key.
        index: The name of the index to use.
    """

    api_key: str = field(kw_only=True, metadata={"serializable": True})
    url: str = field(kw_only=True, metadata={"serializable": True})
    index: str = field(kw_only=True, metadata={"serializable": True})
    _client: Optional[marqo.Client] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> marqo.Client:
        return import_optional_dependency("marqo").Client(self.url, api_key=self.api_
↪key)

    def upsert(
        self,
        value: str | TextArtifact | ImageArtifact,
        *,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        vector_id: Optional[str] = None,
        **kwargs: Any,
    ) -> str:
        """Upsert a text document into the Marqo index.

        Args:
            value: The value to be indexed.
            namespace: An optional namespace for the document.
            meta: An optional dictionary of metadata for the document.
            vector_id: The ID for the vector. If None, Marqo will generate an ID.
            kwargs: Additional keyword arguments to pass to the Marqo client.

        Returns:
            str: The ID of the document that was added.
        """
        if isinstance(value, TextArtifact):
            artifact_json = value.to_json()
            vector_id = utils.str_to_hash(value.value) if vector_id is None else␣
↪vector_id

            doc = {
                "_id": vector_id,
                "Description": value.value,
```

(continues on next page)

(continued from previous page)

```
                "artifact": str(artifact_json),
            }
        elif isinstance(value, ImageArtifact):
            raise NotImplementedError("`MarqoVectorStoreDriver` does not upserting␣
↪Image Artifacts.")
        else:
            doc = {"_id": vector_id, "Description": value}

        # Non-tensor fields
        if meta:
            doc["meta"] = str(meta)
        if namespace:
            doc["namespace"] = namespace

        response = self.client.index(self.index).add_documents([doc], tensor_fields=[
↪"Description"])
        if isinstance(response, dict) and "items" in response and response["items"]:
            return response["items"][0]["_id"]
        raise ValueError(f"Failed to upsert text: {response}")

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        """Load a document entry from the Marqo index.

        Args:
            vector_id: The ID of the vector to load.
            namespace: The namespace of the vector to load.

        Returns:
            The loaded Entry if found, otherwise None.
        """
        result = self.client.index(self.index).get_document(document_id=vector_id,␣
↪expose_facets=True)

        if result and "_tensor_facets" in result and len(result["_tensor_facets"]) >␣
↪0:
            return BaseVectorStoreDriver.Entry(
                id=result["_id"],
                meta={k: v for k, v in result.items() if k != "_id"},
                vector=result["_tensor_facets"][0]["_embedding"],
            )
        return None

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Load all document entries from the Marqo index.

        Args:
            namespace: The namespace to filter entries by.

        Returns:
            The list of loaded Entries.
```

(continues on next page)

(continued from previous page)

```
        """
        filter_string = f"namespace:{namespace}" if namespace else None

        if filter_string is not None:
            results = self.client.index(self.index).search("", limit=10000, filter_
↪string=filter_string)
        else:
            results = self.client.index(self.index).search("", limit=10000)

        # get all _ids from search results
        ids = [r["_id"] for r in results["hits"]]

        # get documents corresponding to the ids
        documents = self.client.index(self.index).get_documents(document_ids=ids,␣
↪expose_facets=True)

        # for each document, if its found, create an Entry object
        entries = []
        for doc in documents["results"]:
            if doc["_found"]:
                entries.append(
                    BaseVectorStoreDriver.Entry(
                        id=doc["_id"],
                        vector=doc["_tensor_facets"][0]["_embedding"],
                        meta={k: v for k, v in doc.items() if k not in ["_id", "_
↪tensor_facets", "_found"]},
                        namespace=doc.get("namespace"),
                    ),
                )

        return entries

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        include_metadata: bool = True,
        **kwargs: Any,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Query the Marqo index for documents.

        Args:
            vector: The vector to query by.
            count: The maximum number of results to return.
            namespace: The namespace to filter results by.
            include_vectors: Whether to include vector data in the results.
            include_metadata: Whether to include metadata in the results.
            kwargs: Additional keyword arguments to pass to the Marqo client.
```

(continues on next page)

(continued from previous page)

```
        Returns:
            The list of query results.
        """
        params = {
            "limit": count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
            "attributes_to_retrieve": None if include_metadata else ["_id"],
            "filter_string": f"namespace:{namespace}" if namespace else None,
        } | kwargs

        results = self.client.index(self.index).search(**params, context={"tensor":␣
↪[vector], "weight": 1})

        return self.__process_results(results, include_vectors=include_vectors)

    def query(
        self,
        query: str | TextArtifact | ImageArtifact,
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        include_metadata: bool = True,
        **kwargs: Any,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Query the Marqo index for documents.

        Args:
            query: The query string.
            count: The maximum number of results to return.
            namespace: The namespace to filter results by.
            include_vectors: Whether to include vector data in the results.
            include_metadata: Whether to include metadata in the results.
            kwargs: Additional keyword arguments to pass to the Marqo client.

        Returns:
            The list of query results.
        """
        params = {
            "limit": count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
            "attributes_to_retrieve": None if include_metadata else ["_id"],
            "filter_string": f"namespace:{namespace}" if namespace else None,
        } | kwargs

        results = self.client.index(self.index).search(str(query), **params)
        return self.__process_results(results, include_vectors=include_vectors)

    def delete_index(self, name: str) -> dict[str, Any]:
        """Delete an index in the Marqo client.

        Args:
            name: The name of the index to delete.
        """
```

(continues on next page)

(continued from previous page)

```
        return self.client.delete_index(name)

    def get_indexes(self) -> list[str]:
        """Get a list of all indexes in the Marqo client.

        Returns:
            The list of all indexes.
        """
        return [index["index"] for index in self.client.get_indexes()["results"]]

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs: Any,
    ) -> str:
        """Upsert a vector into the Marqo index.

        Args:
            vector: The vector to be indexed.
            vector_id: The ID for the vector. If None, Marqo will generate an ID.
            namespace: An optional namespace for the vector.
            meta: An optional dictionary of metadata for the vector.
            kwargs: Additional keyword arguments to pass to the Marqo client.

        Raises:
            Exception: This function is not yet implemented.

        Returns:
            The ID of the vector that was added.
        """
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪upserting a vector.")

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")

    def __process_results(self, results: dict, *, include_vectors: bool) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        if include_vectors:
            results["hits"] = [
                {**r, **self.client.index(self.index).get_document(r["_id"], expose_
↪facets=True)}
                for r in results["hits"]
            ]

        return [
            BaseVectorStoreDriver.Entry(
```

(continues on next page)

(continued from previous page)

```
                id=r["_id"],
                vector=r["_tensor_facets"][0]["_embedding"] if include_vectors else␣
↪[],
                score=r["_score"],
                meta={k: v for k, v in r.items() if k not in ["_score", "_tensor_
↪facets"]},
            )
            for r in results["hits"]
        ]
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `index = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `url = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__process_results(results, *, include_vectors)

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def __process_results(self, results: dict, *, include_vectors: bool) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    if include_vectors:
        results["hits"] = [
            {**r, **self.client.index(self.index).get_document(r["_id"], expose_
↪facets=True)}
            for r in results["hits"]
        ]

    return [
        BaseVectorStoreDriver.Entry(
            id=r["_id"],
            vector=r["_tensor_facets"][0]["_embedding"] if include_vectors else [],
            score=r["_score"],
            meta={k: v for k, v in r.items() if k not in ["_score", "_tensor_facets"]}
↪,
        )
        for r in results["hits"]
    ]
```

client()

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
@lazy_property()
def client(self) -> marqo.Client:
    return import_optional_dependency("marqo").Client(self.url, api_key=self.api_key)
```

delete_index(name)

Delete an index in the Marqo client.

Parameters

Name	Type	Description	Default
name	*str*	The name of the index to delete. 	*required*

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def delete_index(self, name: str) -> dict[str, Any]:
    """Delete an index in the Marqo client.

    Args:
        name: The name of the index to delete.
    """
    return self.client.delete_index(name)
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

get_indexes()

Get a list of all indexes in the Marqo client.

Returns

Type	Description
<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	The list of all indexes.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def get_indexes(self) -> list[str]:
    """Get a list of all indexes in the Marqo client.

    Returns:
        The list of all indexes.
    """
    return [index["index"] for index in self.client.get_indexes()["results"]]
```

load_entries(*, namespace=None)

Load all document entries from the Marqo index.

Parameters

Name	Type	Description	Default
namespace	*Optional[str]*	The namespace to filter entries by. 	*None*

Returns

Type	Description
<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	The list of loaded Entries.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Load all document entries from the Marqo index.

    Args:
        namespace: The namespace to filter entries by.

    Returns:
        The list of loaded Entries.
    """
    filter_string = f"namespace:{namespace}" if namespace else None

    if filter_string is not None:
        results = self.client.index(self.index).search("", limit=10000, filter_
↪string=filter_string)
    else:
        results = self.client.index(self.index).search("", limit=10000)

    # get all _ids from search results
    ids = [r["_id"] for r in results["hits"]]

    # get documents corresponding to the ids
    documents = self.client.index(self.index).get_documents(document_ids=ids, expose_
↪facets=True)

    # for each document, if its found, create an Entry object
    entries = []
    for doc in documents["results"]:
        if doc["_found"]:
```

(continues on next page)

(continued from previous page)

```
            entries.append(
                BaseVectorStoreDriver.Entry(
                    id=doc["_id"],
                    vector=doc["_tensor_facets"][0]["_embedding"],
                    meta={k: v for k, v in doc.items() if k not in ["_id", "_tensor_
↪facets", "_found"]},
                    namespace=doc.get("namespace"),
                ),
            )

    return entries
```

load_entry(vector_id, *, namespace=None)

Load a document entry from the Marqo index.

Parameters

Name	Type	Description	Default
vector_id	*str*	The ID of the vector to load. 	*required*
namespace	*Optional[str]*	The namespace of the vector to load. 	*None*

Returns

Type	Description
<a href="vector#griptape.drivers.vector.BaseVectorStoreDriver.Entry" title="Entry (griptape.drivers.vector.BaseVectorStoreDriver.Entry)"><code class="language-python">C	The loaded Entry if found, otherwise None.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    """Load a document entry from the Marqo index.

    Args:
        vector_id: The ID of the vector to load.
        namespace: The namespace of the vector to load.

    Returns:
```

(continues on next page)

(continued from previous page)

```
        The loaded Entry if found, otherwise None.
    """
    result = self.client.index(self.index).get_document(document_id=vector_id, expose_
↪facets=True)

    if result and "_tensor_facets" in result and len(result["_tensor_facets"]) > 0:
        return BaseVectorStoreDriver.Entry(
            id=result["_id"],
            meta={k: v for k, v in result.items() if k != "_id"},
            vector=result["_tensor_facets"][0]["_embedding"],
        )
    return None
```

query(query, *, count=None, namespace=None, include_vectors=False, include_metadata=True, **kwargs)

Query the Marqo index for documents.

Parameters

Name	Type	Description	Default
query	<code class="language-pythc I TextArtifact I ImageArtifact</code>	The query string. 	*required*
count	*Optional[int]*	The maximum number of results to return. 	*None*
namespace	*Optional[str]*	The namespace to filter results by. 	*None*
include_vectors	*bool*	Whether to include vector data in the results. 	*False*
include_metadata	*bool*	Whether to include metadata in the results. 	*True*
kwargs	*Any*	Additional keyword arguments to pass to the Marqo client. 	*{}*

Returns

Type	Description
<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	The list of query results.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def query(
    self,
    query: str | TextArtifact | ImageArtifact,
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    include_metadata: bool = True,
    **kwargs: Any,
) -> list[BaseVectorStoreDriver.Entry]:
    """Query the Marqo index for documents.

    Args:
        query: The query string.
        count: The maximum number of results to return.
        namespace: The namespace to filter results by.
        include_vectors: Whether to include vector data in the results.
        include_metadata: Whether to include metadata in the results.
        kwargs: Additional keyword arguments to pass to the Marqo client.

    Returns:
        The list of query results.
    """
    params = {
        "limit": count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
        "attributes_to_retrieve": None if include_metadata else ["_id"],
        "filter_string": f"namespace:{namespace}" if namespace else None,
    } | kwargs

    results = self.client.index(self.index).search(str(query), **params)
    return self.__process_results(results, include_vectors=include_vectors)
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, include_metadata=True, **kwargs)

Query the Marqo index for documents.

Parameters

Name	Type	Description	Default
vector	<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	The vector to query by. 	*required*
count	*Optional[int]*	The maximum number of results to return. 	*None*
namespace	*Optional[str]*	The namespace to filter results by. 	*None*
include_vectors	*bool*	Whether to include vector data in the results. 	*False*
include_metadata	*bool*	Whether to include metadata in the results. 	*True*
kwargs	*Any*	Additional keyword arguments to pass to the Marqo client. 	*{}*

Returns

Type	Description
<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	The list of query results.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    include_metadata: bool = True,
    **kwargs: Any,
) -> list[BaseVectorStoreDriver.Entry]:
    """Query the Marqo index for documents.

    Args:
        vector: The vector to query by.
        count: The maximum number of results to return.
```

(continues on next page)

(continued from previous page)

```
        namespace: The namespace to filter results by.
        include_vectors: Whether to include vector data in the results.
        include_metadata: Whether to include metadata in the results.
        kwargs: Additional keyword arguments to pass to the Marqo client.

    Returns:
        The list of query results.
    """
    params = {
        "limit": count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
        "attributes_to_retrieve": None if include_metadata else ["_id"],
        "filter_string": f"namespace:{namespace}" if namespace else None,
    } | kwargs

    results = self.client.index(self.index).search(**params, context={"tensor":␣
↪[vector], "weight": 1})

    return self.__process_results(results, include_vectors=include_vectors)
```

upsert(value, *, namespace=None, meta=None, vector_id=None, **kwargs)

Upsert a text document into the Marqo index.

Parameters

Name	Type	Description	Default
value	<code class="language-pythc I TextArtifact I ImageArtifact</code>	The value to be indexed. 	*required*
namespace	*Optional[str]*	An optional namespace for the document. 	*None*
meta	*Optional[dict]*	An optional dictionary of metadata for the document. 	*None*
vector_id	*Optional[str]*	The ID for the vector. If None, Marqo will generate an ID. 	*None*

continues on next page

Table 94 – continued from previous page

Name	Type	Description	Default
kwargs	*Any*	Additional keyword arguments to pass to the Marqo client. 	*{}*

Returns

Name	Type	Description
str	*str*	The ID of the document that was added.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def upsert(
    self,
    value: str | TextArtifact | ImageArtifact,
    *,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    vector_id: Optional[str] = None,
    **kwargs: Any,
) -> str:
    """Upsert a text document into the Marqo index.

    Args:
        value: The value to be indexed.
        namespace: An optional namespace for the document.
        meta: An optional dictionary of metadata for the document.
        vector_id: The ID for the vector. If None, Marqo will generate an ID.
        kwargs: Additional keyword arguments to pass to the Marqo client.

    Returns:
        str: The ID of the document that was added.
    """
    if isinstance(value, TextArtifact):
        artifact_json = value.to_json()
        vector_id = utils.str_to_hash(value.value) if vector_id is None else vector_id

        doc = {
            "_id": vector_id,
            "Description": value.value,
            "artifact": str(artifact_json),
        }
    elif isinstance(value, ImageArtifact):
        raise NotImplementedError("`MarqoVectorStoreDriver` does not upserting Image␣
↪Artifacts.")
    else:
        doc = {"_id": vector_id, "Description": value}
```

(continues on next page)

(continued from previous page)

```
    # Non-tensor fields
    if meta:
        doc["meta"] = str(meta)
    if namespace:
        doc["namespace"] = namespace

    response = self.client.index(self.index).add_documents([doc], tensor_fields=[
↪"Description"])
    if isinstance(response, dict) and "items" in response and response["items"]:
        return response["items"][0]["_id"]
    raise ValueError(f"Failed to upsert text: {response}")
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Upsert a vector into the Marqo index.

Parameters

Name	Type	Description	Default
vector	<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	The vector to be indexed. 	*required*
vector_id	*Optional[str]*	The ID for the vector. If None, Marqo will generate an ID. 	*None*
namespace	*Optional[str]*	An optional namespace for the vector. 	*None*
meta	*Optional[dict]*	An optional dictionary of metadata for the vector. 	*None*
kwargs	*Any*	Additional keyword arguments to pass to the Marqo client. 	*{}*

Raises

Returns

Type	Description
str	The ID of the vector that was added.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs: Any,
) -> str:
    """Upsert a vector into the Marqo index.

    Args:
        vector: The vector to be indexed.
        vector_id: The ID for the vector. If None, Marqo will generate an ID.
        namespace: An optional namespace for the vector.
        meta: An optional dictionary of metadata for the vector.
        kwargs: Additional keyword arguments to pass to the Marqo client.

    Raises:
        Exception: This function is not yet implemented.

    Returns:
        The ID of the vector that was added.
    """
    raise NotImplementedError(f"{self.__class__.__name__} does not support upserting␣
↪a vector.")
```

MongoDbAtlasVectorStoreDriver

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.MongoDbAtlasVectorStoreDriver.connection_string" title="connection_string = field(kw_only=True, metadata={'serializable': True})">class-attribute instance-attribute (griptape.drivers.MongoDbAtlasVectorStoreDriver.connection_string)"><code class="language-python">	*str*	The connection string for the MongoDb Atlas cluster.

continues on next page

Table 98 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.MongoDbAtlasVectorStoreDriver.database_nan title="database_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.MongoDbAtlasVectorStoreDriver.database_nan class="language-python">	*str*	The name of the database to use.
<a href="./#griptape.drivers.MongoDbAtlasVectorStoreDriver.collection_name" title="collection_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.MongoDbAtlasVectorStoreDriver.collection_name)"><code class="language-python">	*str*	The name of the collection to use.
<a href="./#griptape.drivers.MongoDbAtlasVectorStoreDriver.index_name" title="index_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.MongoDbAtlasVectorStoreDriver.index_name)"><code class="language-python">	*str*	The name of the index to use.

continues on next page

Table 98 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.MongoDbAtlasVectorStoreDriver.vector_path" title="vector_path = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.MongoDbAtlasVectorStoreDriver.vector_path)"><code class="language-python">	*str*	The path to the vector field in the collection.
<a href="./#griptape.drivers.MongoDbAtlasVectorStoreDriver.client" title="client() (griptape.drivers.MongoDbAtlasVectorStoreDriver.client)"><coc class="language-python">	*MongoClient*	An optional MongoDb client to use. Defaults to a new client using the connection string.

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
@define
class MongoDbAtlasVectorStoreDriver(BaseVectorStoreDriver):
    """A Vector Store Driver for MongoDb Atlas.

    Attributes:
        connection_string: The connection string for the MongoDb Atlas cluster.
        database_name: The name of the database to use.
        collection_name: The name of the collection to use.
        index_name: The name of the index to use.
        vector_path: The path to the vector field in the collection.
        client: An optional MongoDb client to use. Defaults to a new client using the␣
↪connection string.
    """

    MAX_NUM_CANDIDATES = 10000

    connection_string: str = field(kw_only=True, metadata={"serializable": True})
    database_name: str = field(kw_only=True, metadata={"serializable": True})
    collection_name: str = field(kw_only=True, metadata={"serializable": True})
    index_name: str = field(kw_only=True, metadata={"serializable": True})
    vector_path: str = field(kw_only=True, metadata={"serializable": True})
    num_candidates_multiplier: int = field(
```

(continues on next page)

(continued from previous page)

```
        default=10,
        kw_only=True,
        metadata={"serializable": True},
    )  # https://www.mongodb.com/docs/atlas/atlas-vector-search/vector-search-stage/
↪#fields
    _client: Optional[MongoClient] = field(default=None, kw_only=True, alias="client",
↪ metadata={"serializable": False})

    @lazy_property()
    def client(self) -> MongoClient:
        return import_optional_dependency("pymongo").MongoClient(self.connection_
↪string)

    def get_collection(self) -> Collection:
        """Returns the MongoDB Collection instance for the specified database and␣
↪collection name."""
        return self.client[self.database_name][self.collection_name]

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        """Inserts or updates a vector in the collection.

        If a vector with the given vector ID already exists, it is updated; otherwise,
↪ a new vector is inserted.
        """
        collection = self.get_collection()

        if vector_id is None:
            result = collection.insert_one({self.vector_path: vector, "namespace":␣
↪namespace, "meta": meta})
            vector_id = str(result.inserted_id)
        else:
            collection.replace_one(
                {"_id": vector_id},
                {self.vector_path: vector, "namespace": namespace, "meta": meta},
                upsert=True,
            )
        return vector_id

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        """Loads a document entry from the MongoDB collection based on the vector ID.

        Returns:
            The loaded Entry if found; otherwise, None is returned.
```

(continues on next page)

(continued from previous page)

```
        """
        collection = self.get_collection()
        if namespace:
            doc = collection.find_one({"_id": vector_id, "namespace": namespace})
        else:
            doc = collection.find_one({"_id": vector_id})

        if doc is None:
            return doc
        return BaseVectorStoreDriver.Entry(
            id=str(doc["_id"]),
            vector=doc[self.vector_path],
            namespace=doc["namespace"],
            meta=doc["meta"],
        )

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Loads all document entries from the MongoDB collection.

        Entries can optionally be filtered by namespace.
        """
        collection = self.get_collection()
        cursor = collection.find() if namespace is None else collection.find({
↪"namespace": namespace})

        return [
            BaseVectorStoreDriver.Entry(
                id=str(doc["_id"]),
                vector=doc[self.vector_path],
                namespace=doc["namespace"],
                meta=doc["meta"],
            )
            for doc in cursor
        ]

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        offset: Optional[int] = None,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Queries the MongoDB collection for documents that match the provided␣
↪vector list.

        Results can be customized based on parameters like count, namespace,␣
↪inclusion of vectors, offset, and index.
        """
```

(continues on next page)

(continued from previous page)

```
        collection = self.get_collection()

        count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
        offset = offset or 0

        pipeline = [
            {
                "$vectorSearch": {
                    "index": self.index_name,
                    "path": self.vector_path,
                    "queryVector": vector,
                    "numCandidates": min(count * self.num_candidates_multiplier, self.
↪MAX_NUM_CANDIDATES),
                    "limit": count,
                },
            },
            {
                "$project": {
                    "_id": 1,
                    self.vector_path: 1,
                    "namespace": 1,
                    "meta": 1,
                    "score": {"$meta": "vectorSearchScore"},
                },
            },
        ]

        if namespace:
            pipeline[0]["$vectorSearch"]["filter"] = {"namespace": namespace}

        return [
            BaseVectorStoreDriver.Entry(
                id=str(doc["_id"]),
                vector=doc[self.vector_path] if include_vectors else [],
                score=doc["score"],
                meta=doc["meta"],
                namespace=namespace,
            )
            for doc in collection.aggregate(pipeline)
        ]

    def delete_vector(self, vector_id: str) -> None:
        """Deletes the vector from the collection."""
        collection = self.get_collection()
        collection.delete_one({"_id": vector_id})
```

- `MAX_NUM_CANDIDATES = 10000` class-attribute instance-attribute
- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `collection_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

- `connection_string = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `database_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `index_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `num_candidates_multiplier = field(default=10, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `vector_path = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
@lazy_property()
def client(self) -> MongoClient:
    return import_optional_dependency("pymongo").MongoClient(self.connection_string)
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> None:
    """Deletes the vector from the collection."""
    collection = self.get_collection()
    collection.delete_one({"_id": vector_id})
```

get_collection()

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
def get_collection(self) -> Collection:
    """Returns the MongoDB Collection instance for the specified database and␣
↪collection name."""
    return self.client[self.database_name][self.collection_name]
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Loads all document entries from the MongoDB collection.

    Entries can optionally be filtered by namespace.
    """
    collection = self.get_collection()
    cursor = collection.find() if namespace is None else collection.find({"namespace
↪": namespace})
```

(continues on next page)

(continued from previous page)

```
    return [
        BaseVectorStoreDriver.Entry(
            id=str(doc["_id"]),
            vector=doc[self.vector_path],
            namespace=doc["namespace"],
            meta=doc["meta"],
        )
        for doc in cursor
    ]
```

load_entry(vector_id, *, namespace=None)

Loads a document entry from the MongoDB collection based on the vector ID.

Returns

Type	Description
<a href="vector#griptape.drivers.vector.BaseVectorStoreDriver.Entry" title="Entry (griptape.drivers.vector.BaseVectorStoreDriver.Entry)"><code class="language-python">C	The loaded Entry if found; otherwise, None is returned.

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    """Loads a document entry from the MongoDB collection based on the vector ID.

    Returns:
        The loaded Entry if found; otherwise, None is returned.
    """
    collection = self.get_collection()
    if namespace:
        doc = collection.find_one({"_id": vector_id, "namespace": namespace})
    else:
        doc = collection.find_one({"_id": vector_id})

    if doc is None:
        return doc
    return BaseVectorStoreDriver.Entry(
        id=str(doc["_id"]),
        vector=doc[self.vector_path],
        namespace=doc["namespace"],
        meta=doc["meta"],
    )
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, offset=None, **kwargs)

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    offset: Optional[int] = None,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    """Queries the MongoDB collection for documents that match the provided vector␣
↪list.

    Results can be customized based on parameters like count, namespace, inclusion of␣
↪vectors, offset, and index.
    """
    collection = self.get_collection()

    count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
    offset = offset or 0

    pipeline = [
        {
            "$vectorSearch": {
                "index": self.index_name,
                "path": self.vector_path,
                "queryVector": vector,
                "numCandidates": min(count * self.num_candidates_multiplier, self.MAX_
↪NUM_CANDIDATES),
                "limit": count,
            },
        },
        {
            "$project": {
                "_id": 1,
                self.vector_path: 1,
                "namespace": 1,
                "meta": 1,
                "score": {"$meta": "vectorSearchScore"},
            },
        },
    ]

    if namespace:
        pipeline[0]["$vectorSearch"]["filter"] = {"namespace": namespace}

    return [
        BaseVectorStoreDriver.Entry(
            id=str(doc["_id"]),
```

(continues on next page)

(continued from previous page)

```
            vector=doc[self.vector_path] if include_vectors else [],
            score=doc["score"],
            meta=doc["meta"],
            namespace=namespace,
        )
        for doc in collection.aggregate(pipeline)
    ]
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    """Inserts or updates a vector in the collection.

    If a vector with the given vector ID already exists, it is updated; otherwise, a␣
↪new vector is inserted.
    """
    collection = self.get_collection()

    if vector_id is None:
        result = collection.insert_one({self.vector_path: vector, "namespace":␣
↪namespace, "meta": meta})
        vector_id = str(result.inserted_id)
    else:
        collection.replace_one(
            {"_id": vector_id},
            {self.vector_path: vector, "namespace": namespace, "meta": meta},
            upsert=True,
        )
    return vector_id
```

NoOpObservabilityDriver

Bases: *BaseObservabilityDriver* “)

Source Code in griptape/drivers/observability/no_op_observability_driver.py

```
@define
class NoOpObservabilityDriver(BaseObservabilityDriver):
    def observe(self, call: Observable.Call) -> Any:
        return call()

    def get_span_id(self) -> Optional[str]:
```

(continues on next page)

(continued from previous page)

```
        return None
```

get_span_id()

Source Code in griptape/drivers/observability/no_op_observability_driver.py

```
def get_span_id(self) -> Optional[str]:
    return None
```

observe(call)

Source Code in griptape/drivers/observability/no_op_observability_driver.py

```
def observe(self, call: Observable.Call) -> Any:
    return call()
```

OllamaEmbeddingDriver

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.OllamaEmbeddingDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.OllamaEmbeddingDriver.model)"><code class="language-python">	*str*	Ollama embedding model name.
<a href="./#griptape.drivers.OllamaEmbeddingDriver.host" title="host = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.OllamaEmbeddingDriver.host)"><code class="language-python">	*Optional[str]*	Optional Ollama host.

continues on next page

Table 100 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.OllamaEmbeddingDriver.client" title="client() (griptape.drivers.OllamaEmbeddingDriver.client)"><code class="language-python">	*Client*	Ollama *Client*.

Source Code in griptape/drivers/embedding/ollama_embedding_driver.py

```
@define
class OllamaEmbeddingDriver(BaseEmbeddingDriver):
    """Ollama Embedding Driver.

    Attributes:
        model: Ollama embedding model name.
        host: Optional Ollama host.
        client: Ollama `Client`.
    """

    model: str = field(kw_only=True, metadata={"serializable": True})
    host: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    _client: Optional[Client] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> Client:
        return import_optional_dependency("ollama").Client(host=self.host)

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        return list(self.client.embeddings(model=self.model, prompt=chunk)["embedding
↪"])
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `host = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/ollama_embedding_driver.py

```
@lazy_property()
def client(self) -> Client:
    return import_optional_dependency("ollama").Client(host=self.host)
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/ollama_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    return list(self.client.embeddings(model=self.model, prompt=chunk)["embedding"])
```

OllamaPromptDriver

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.OllamaPromptDriver.model" title="model = field(kw_only=True, metadata={'serializable': True})" class="attribute instance-attribute">(griptape.drivers.OllamaPromptDriver.model)"><code class="language-python">	*str*	Model name.

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
@define
class OllamaPromptDriver(BasePromptDriver):
    """Ollama Prompt Driver.

    Attributes:
        model: Model name.
    """

    model: str = field(kw_only=True, metadata={"serializable": True})
    host: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    tokenizer: BaseTokenizer = field(
        default=Factory(
            lambda self: SimpleTokenizer(
                characters_per_token=4,
                max_input_tokens=2000,
                max_output_tokens=self.max_tokens,
            ),
            takes_self=True,
        ),
        kw_only=True,
```

(continues on next page)

(continued from previous page)

```
    )
    options: dict = field(
        default=Factory(
            lambda self: {
                "temperature": self.temperature,
                "stop": self.tokenizer.stop_sequences,
                "num_predict": self.max_tokens,
            },
            takes_self=True,
        ),
        kw_only=True,
    )
    use_native_tools: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})
    _client: Optional[Client] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> Client:
        return import_optional_dependency("ollama").Client(host=self.host)

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        params = self._base_params(prompt_stack)
        logger.debug(params)
        response = self.client.chat(**params)
        logger.debug(response.model_dump())

        return Message(
            content=self.__to_prompt_stack_message_content(response),
            role=Message.ASSISTANT_ROLE,
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        params = {**self._base_params(prompt_stack), "stream": True}
        logger.debug(params)
        stream: Iterator = self.client.chat(**params)

        tool_index = 0
        for chunk in stream:
            logger.debug(chunk)
            message_content = self.__to_prompt_stack_delta_message_content(chunk)
            # Ollama provides multiple Tool calls as separate chunks but with no␣
↪index to differentiate them.
            # So we must keep track of the index ourselves.
            if isinstance(message_content, ActionCallDeltaMessageContent):
                message_content.index = tool_index
                tool_index += 1
            yield DeltaMessage(content=message_content)

    def _base_params(self, prompt_stack: PromptStack) -> dict:
```

(continues on next page)

(continued from previous page)

```
        messages = self._prompt_stack_to_messages(prompt_stack)

        params = {
            "messages": messages,
            "model": self.model,
            "options": self.options,
            **self.extra_params,
        }

        if prompt_stack.output_schema is not None and self.structured_output_strategy␣
↪== "native":
            params["format"] = prompt_stack.to_output_json_schema()

        # Tool calling is only supported when not streaming
        if prompt_stack.tools and self.use_native_tools:
            params["tools"] = self.__to_ollama_tools(prompt_stack.tools)

        return params

    def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
        ollama_messages = []
        for message in prompt_stack.messages:
            action_result_contents = message.get_content_
↪type(ActionResultMessageContent)

            # Function calls need to be handled separately from the rest of the␣
↪message content
            if action_result_contents:
                ollama_messages.extend(
                    [
                        {
                            "role": self.__to_ollama_role(message, action_result_
↪content),
                            "content": self.__to_ollama_message_content(action_result_
↪content),
                        }
                        for action_result_content in action_result_contents
                    ],
                )

                text_contents = message.get_content_type(TextMessageContent)
                if text_contents:
                    ollama_messages.append({"role": self.__to_ollama_role(message),
↪"content": message.to_text()})
            else:
                ollama_message: dict[str, Any] = {
                    "role": self.__to_ollama_role(message),
                    "content": message.to_text(),
                }

                action_call_contents = message.get_content_
↪type(ActionCallMessageContent)
```

(continues on next page)

(continued from previous page)

```
                if action_call_contents:
                    ollama_message["tool_calls"] = [
                        self.__to_ollama_message_content(action_call_content)
                        for action_call_content in action_call_contents
                    ]

                image_contents = message.get_content_type(ImageMessageContent)
                if image_contents:
                    ollama_message["images"] = [
                        self.__to_ollama_message_content(image_content) for image_
↪content in image_contents
                    ]

                ollama_messages.append(ollama_message)

        return ollama_messages

    def __to_ollama_message_content(self, content: BaseMessageContent) -> str | dict:
        if isinstance(content, TextMessageContent):
            return content.artifact.to_text()
        if isinstance(content, ImageMessageContent):
            if isinstance(content.artifact, ImageArtifact):
                return content.artifact.base64
            # TODO: add support for image urls once resolved https://github.com/
↪ollama/ollama/issues/4474
            raise ValueError(f"Unsupported image artifact type: {type(content.
↪artifact)}")
        if isinstance(content, ActionCallMessageContent):
            action = content.artifact.value

            return {
                "type": "function",
                "id": action.tag,
                "function": {"name": action.to_native_tool_name(), "arguments":␣
↪action.input},
            }
        if isinstance(content, ActionResultMessageContent):
            return content.artifact.to_text()
        raise ValueError(f"Unsupported content type: {type(content)}")

    def __to_ollama_tools(self, tools: list[BaseTool]) -> list[dict]:
        ollama_tools = []

        for tool in tools:
            for activity in tool.activities():
                ollama_tool = {
                    "function": {
                        "name": tool.to_native_tool_name(activity),
                        "description": tool.activity_description(activity),
                    },
                    "type": "function",
                }
```

(continues on next page)

(continued from previous page)

```
                activity_schema = tool.activity_schema(activity)
                if activity_schema is not None:
                    ollama_tool["function"]["parameters"] = tool.to_activity_json_
↪schema(activity, "Parameters Schema")[
                        "properties"
                    ]["values"]

                ollama_tools.append(ollama_tool)
        return ollama_tools

    def __to_ollama_role(self, message: Message, message_content:␣
↪Optional[BaseMessageContent] = None) -> str:
        if message.is_system():
            return "system"
        if message.is_assistant():
            return "assistant"
        if isinstance(message_content, ActionResultMessageContent):
            return "tool"
        return "user"

    def __to_prompt_stack_message_content(self, response: ChatResponse) ->␣
↪list[BaseMessageContent]:
        content = []
        message = response["message"]

        if message.get("content"):
            content.append(TextMessageContent(TextArtifact(response["message"][
↪"content"])))
        if "tool_calls" in message:
            content.extend(
                [
                    ActionCallMessageContent(
                        ActionArtifact(
                            ToolAction(
                                tag=tool_call["function"]["name"],
                                name=ToolAction.from_native_tool_name(tool_call[
↪"function"]["name"])[0],
                                path=ToolAction.from_native_tool_name(tool_call[
↪"function"]["name"])[1],
                                input=tool_call["function"]["arguments"],
                            ),
                        )
                    )
                    for tool_call in message["tool_calls"]
                ],
            )

        return content

    def __to_prompt_stack_delta_message_content(self, content_delta: ChatResponse) ->␣
↪BaseDeltaMessageContent:
```

(continues on next page)

(continued from previous page)

```
        message = content_delta["message"]
        if message.get("content"):
            return TextDeltaMessageContent(message["content"])
        if "tool_calls" in message and len(message["tool_calls"]):
            tool_calls = message["tool_calls"]

            # Ollama doesnt _really_ support Tool streaming. They provide the full␣
↪tool call at once.
            # Multiple, parallel, Tool calls are provided as multiple content deltas.
            # Tracking here: https://github.com/ollama/ollama/issues/7886
            tool_call = tool_calls[0]

            return ActionCallDeltaMessageContent(
                tag=tool_call["function"]["name"],
                name=ToolAction.from_native_tool_name(tool_call["function"]["name
↪"])[0],
                path=ToolAction.from_native_tool_name(tool_call["function"]["name
↪"])[1],
                partial_input=json.dumps(tool_call["function"]["arguments"]),
            )
        return TextDeltaMessageContent("")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `host = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `options = field(default=Factory(lambda self: {'temperature': self.temperature, 'stop': self.tokenizer.stop_sequences, 'num_predict': self.max_tokens}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: SimpleTokenizer(characters_per_token=4, max_input_tokens=2000, max_output_tokens=self.max_tokens), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `use_native_tools = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__to_ollama_message_content(content)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def __to_ollama_message_content(self, content: BaseMessageContent) -> str | dict:
    if isinstance(content, TextMessageContent):
        return content.artifact.to_text()
    if isinstance(content, ImageMessageContent):
        if isinstance(content.artifact, ImageArtifact):
            return content.artifact.base64
        # TODO: add support for image urls once resolved https://github.com/ollama/
↪ollama/issues/4474
        raise ValueError(f"Unsupported image artifact type: {type(content.artifact)}")
```

(continues on next page)

(continued from previous page)

```
    if isinstance(content, ActionCallMessageContent):
        action = content.artifact.value

        return {
            "type": "function",
            "id": action.tag,
            "function": {"name": action.to_native_tool_name(), "arguments": action.
↪input},
        }
    if isinstance(content, ActionResultMessageContent):
        return content.artifact.to_text()
    raise ValueError(f"Unsupported content type: {type(content)}")
```

__to_ollama_role(message, message_content=None)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def __to_ollama_role(self, message: Message, message_content:␣
↪Optional[BaseMessageContent] = None) -> str:
    if message.is_system():
        return "system"
    if message.is_assistant():
        return "assistant"
    if isinstance(message_content, ActionResultMessageContent):
        return "tool"
    return "user"
```

__to_ollama_tools(tools)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def __to_ollama_tools(self, tools: list[BaseTool]) -> list[dict]:
    ollama_tools = []

    for tool in tools:
        for activity in tool.activities():
            ollama_tool = {
                "function": {
                    "name": tool.to_native_tool_name(activity),
                    "description": tool.activity_description(activity),
                },
                "type": "function",
            }

            activity_schema = tool.activity_schema(activity)
            if activity_schema is not None:
                ollama_tool["function"]["parameters"] = tool.to_activity_json_
↪schema(activity, "Parameters Schema")[
                    "properties"
                ]["values"]
```

(continues on next page)

(continued from previous page)

```
            ollama_tools.append(ollama_tool)
    return ollama_tools
```

__to_prompt_stack_delta_message_content(content_delta)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def __to_prompt_stack_delta_message_content(self, content_delta: ChatResponse) ->␣
↪BaseDeltaMessageContent:
    message = content_delta["message"]
    if message.get("content"):
        return TextDeltaMessageContent(message["content"])
    if "tool_calls" in message and len(message["tool_calls"]):
        tool_calls = message["tool_calls"]

        # Ollama doesnt _really_ support Tool streaming. They provide the full tool␣
↪call at once.
        # Multiple, parallel, Tool calls are provided as multiple content deltas.
        # Tracking here: https://github.com/ollama/ollama/issues/7886
        tool_call = tool_calls[0]

        return ActionCallDeltaMessageContent(
            tag=tool_call["function"]["name"],
            name=ToolAction.from_native_tool_name(tool_call["function"]["name"])[0],
            path=ToolAction.from_native_tool_name(tool_call["function"]["name"])[1],
            partial_input=json.dumps(tool_call["function"]["arguments"]),
        )
    return TextDeltaMessageContent("")
```

__to_prompt_stack_message_content(response)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def __to_prompt_stack_message_content(self, response: ChatResponse) ->␣
↪list[BaseMessageContent]:
    content = []
    message = response["message"]

    if message.get("content"):
        content.append(TextMessageContent(TextArtifact(response["message"]["content
↪"])))
    if "tool_calls" in message:
        content.extend(
            [
                ActionCallMessageContent(
                    ActionArtifact(
                        ToolAction(
                            tag=tool_call["function"]["name"],
                            name=ToolAction.from_native_tool_name(tool_call["function
↪"]["name"])[0],
                            path=ToolAction.from_native_tool_name(tool_call["function
↪"]["name"])[1],
```

(continues on next page)

(continued from previous page)

```
                            input=tool_call["function"]["arguments"],
                        ),
                    ),
                )
                for tool_call in message["tool_calls"]
            ],
        )

    return content
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    messages = self._prompt_stack_to_messages(prompt_stack)

    params = {
        "messages": messages,
        "model": self.model,
        "options": self.options,
        **self.extra_params,
    }

    if prompt_stack.output_schema is not None and self.structured_output_strategy ==
↪"native":
        params["format"] = prompt_stack.to_output_json_schema()

    # Tool calling is only supported when not streaming
    if prompt_stack.tools and self.use_native_tools:
        params["tools"] = self.__to_ollama_tools(prompt_stack.tools)

    return params
```

_prompt_stack_to_messages(prompt_stack)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
    ollama_messages = []
    for message in prompt_stack.messages:
        action_result_contents = message.get_content_type(ActionResultMessageContent)

        # Function calls need to be handled separately from the rest of the message␣
↪content
        if action_result_contents:
            ollama_messages.extend(
                [
                    {
                        "role": self.__to_ollama_role(message, action_result_content),
                        "content": self.__to_ollama_message_content(action_result_
↪content),
```

(continues on next page)

(continued from previous page)

```
                    }
                    for action_result_content in action_result_contents
                ],
            )

            text_contents = message.get_content_type(TextMessageContent)
            if text_contents:
                ollama_messages.append({"role": self.__to_ollama_role(message),
↪"content": message.to_text()})
        else:
            ollama_message: dict[str, Any] = {
                "role": self.__to_ollama_role(message),
                "content": message.to_text(),
            }

            action_call_contents = message.get_content_type(ActionCallMessageContent)
            if action_call_contents:
                ollama_message["tool_calls"] = [
                    self.__to_ollama_message_content(action_call_content)
                    for action_call_content in action_call_contents
                ]

            image_contents = message.get_content_type(ImageMessageContent)
            if image_contents:
                ollama_message["images"] = [
                    self.__to_ollama_message_content(image_content) for image_content␣
↪in image_contents
                ]

            ollama_messages.append(ollama_message)

    return ollama_messages
```

client()

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
@lazy_property()
def client(self) -> Client:
    return import_optional_dependency("ollama").Client(host=self.host)
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    params = self._base_params(prompt_stack)
    logger.debug(params)
    response = self.client.chat(**params)
    logger.debug(response.model_dump())
```

(continues on next page)

(continued from previous page)

```
    return Message(
        content=self.__to_prompt_stack_message_content(response),
        role=Message.ASSISTANT_ROLE,
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    params = {**self._base_params(prompt_stack), "stream": True}
    logger.debug(params)
    stream: Iterator = self.client.chat(**params)

    tool_index = 0
    for chunk in stream:
        logger.debug(chunk)
        message_content = self.__to_prompt_stack_delta_message_content(chunk)
        # Ollama provides multiple Tool calls as separate chunks but with no index to␣
↪differentiate them.
        # So we must keep track of the index ourselves.
        if isinstance(message_content, ActionCallDeltaMessageContent):
            message_content.index = tool_index
            tool_index += 1
        yield DeltaMessage(content=message_content)
```

OpenAiAssistantDriver

Bases: *BaseAssistantDriver* “)

Source Code in griptape/drivers/assistant/openai_assistant_driver.py

```
@define
class OpenAiAssistantDriver(BaseAssistantDriver):
    class EventHandler(AssistantEventHandler):
        @override
        def on_text_delta(self, delta: TextDelta, snapshot: Text) -> None:
            if delta.value is not None:
                EventBus.publish_event(TextChunkEvent(token=delta.value))

        @override
        def on_tool_call_delta(self, delta: ToolCallDelta, snapshot: ToolCall) ->␣
↪None:
            if delta.type == "code_interpreter" and delta.code_interpreter is not␣
↪None:
                if delta.code_interpreter.input:
                    EventBus.publish_event(TextChunkEvent(token=delta.code_
↪interpreter.input))
                if delta.code_interpreter.outputs:
                    EventBus.publish_event(TextChunkEvent(token="\n\noutput >"))
                    for output in delta.code_interpreter.outputs:
```

(continues on next page)

(continued from previous page)

```
                        if output.type == "logs" and output.logs:
                            EventBus.publish_event(TextChunkEvent(token=output.logs))

    base_url: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    organization: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    thread_id: Optional[str] = field(default=None, kw_only=True)
    assistant_id: str = field(kw_only=True)
    event_handler: AssistantEventHandler = field(
        default=Factory(lambda: OpenAiAssistantDriver.EventHandler()), kw_only=True,␣
↪metadata={"serializable": False}
    )
    auto_create_thread: bool = field(default=True, kw_only=True)

    _client: Optional[openai.OpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.OpenAI:
        return openai.OpenAI(
            base_url=self.base_url,
            api_key=self.api_key,
            organization=self.organization,
        )

    def try_run(self, *args: BaseArtifact) -> TextArtifact:
        if self.thread_id is None:
            if self.auto_create_thread:
                thread_id = self.client.beta.threads.create().id
                self.thread_id = thread_id
            else:
                raise ValueError("Thread ID is required but not provided and auto_
↪create_thread is disabled.")
        else:
            thread_id = self.thread_id

        response = self._create_run(thread_id, *args)

        response.meta.update({"assistant_id": self.assistant_id, "thread_id": self.
↪thread_id})

        return response

    def _create_run(self, thread_id: str, *args: BaseArtifact) -> TextArtifact:
        content = "\n".join(arg.value for arg in args)
        message_id = self.client.beta.threads.messages.create(thread_id=thread_id,␣
↪role="user", content=content)
        with self.client.beta.threads.runs.stream(
```

(continues on next page)

(continued from previous page)

```
            thread_id=thread_id,
            assistant_id=self.assistant_id,
            event_handler=self.event_handler,
        ) as stream:
            stream.until_done()
            last_messages = stream.get_final_messages()

            message_contents = []
            for message in last_messages:
                message_contents.append(
                    "".join(content.text.value for content in message.content if␣
↪content.type == "TextContentBlock")
                )
            message_text = "\n".join(message_contents)

            response = TextArtifact(message_text)

            response.meta.update(
                {"assistant_id": self.assistant_id, "thread_id": self.thread_id,
↪"message_id": message_id}
            )
            return response
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `assistant_id = field(kw_only=True)` class-attribute instance-attribute
- `auto_create_thread = field(default=True, kw_only=True)` class-attribute instance-attribute
- `base_url = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `event_handler = field(default=Factory(lambda: OpenAiAssistantDriver.EventHandler()), kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `organization = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `thread_id = field(default=None, kw_only=True)` class-attribute instance-attribute

EventHandler

Bases:

AssistantEventHandler

Source Code in griptape/drivers/assistant/openai_assistant_driver.py

```
class EventHandler(AssistantEventHandler):
    @override
    def on_text_delta(self, delta: TextDelta, snapshot: Text) -> None:
        if delta.value is not None:
```

(continues on next page)

(continued from previous page)

```
            EventBus.publish_event(TextChunkEvent(token=delta.value))

    @override
    def on_tool_call_delta(self, delta: ToolCallDelta, snapshot: ToolCall) -> None:
        if delta.type == "code_interpreter" and delta.code_interpreter is not None:
            if delta.code_interpreter.input:
                EventBus.publish_event(TextChunkEvent(token=delta.code_interpreter.
↪input))
            if delta.code_interpreter.outputs:
                EventBus.publish_event(TextChunkEvent(token="\n\noutput >"))
                for output in delta.code_interpreter.outputs:
                    if output.type == "logs" and output.logs:
                        EventBus.publish_event(TextChunkEvent(token=output.logs))
```

_create_run(thread_id, *args)

Source Code in griptape/drivers/assistant/openai_assistant_driver.py

```
def _create_run(self, thread_id: str, *args: BaseArtifact) -> TextArtifact:
    content = "\n".join(arg.value for arg in args)
    message_id = self.client.beta.threads.messages.create(thread_id=thread_id, role=
↪"user", content=content)
    with self.client.beta.threads.runs.stream(
        thread_id=thread_id,
        assistant_id=self.assistant_id,
        event_handler=self.event_handler,
    ) as stream:
        stream.until_done()
        last_messages = stream.get_final_messages()

        message_contents = []
        for message in last_messages:
            message_contents.append(
                "".join(content.text.value for content in message.content if content.
↪type == "TextContentBlock")
            )
        message_text = "\n".join(message_contents)

        response = TextArtifact(message_text)

        response.meta.update(
            {"assistant_id": self.assistant_id, "thread_id": self.thread_id, "message_
↪id": message_id}
        )
        return response
```

client()

Source Code in griptape/drivers/assistant/openai_assistant_driver.py

```
@lazy_property()
def client(self) -> openai.OpenAI:
```

(continues on next page)

(continued from previous page)

```
    return openai.OpenAI(
        base_url=self.base_url,
        api_key=self.api_key,
        organization=self.organization,
    )
```

try_run(*args)

Source Code in griptape/drivers/assistant/openai_assistant_driver.py

```
def try_run(self, *args: BaseArtifact) -> TextArtifact:
    if self.thread_id is None:
        if self.auto_create_thread:
            thread_id = self.client.beta.threads.create().id
            self.thread_id = thread_id
        else:
            raise ValueError("Thread ID is required but not provided and auto_create_
↪thread is disabled.")
    else:
        thread_id = self.thread_id

    response = self._create_run(thread_id, *args)

    response.meta.update({"assistant_id": self.assistant_id, "thread_id": self.thread_
↪id})

    return response
```

OpenAiAudioTranscriptionDriver

Bases: *BaseAudioTranscriptionDriver* “)

Source Code in griptape/drivers/audio_transcription/openai_audio_transcription_driver.py

```
@define
class OpenAiAudioTranscriptionDriver(BaseAudioTranscriptionDriver):
    api_type: Optional[str] = field(default=openai.api_type, kw_only=True)
    api_version: Optional[str] = field(default=openai.api_version, kw_only=True,␣
↪metadata={"serializable": True})
    base_url: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    organization: Optional[str] = field(default=openai.organization, kw_only=True,␣
↪metadata={"serializable": True})
    _client: Optional[openai.OpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.OpenAI:
        return openai.OpenAI(api_key=self.api_key, base_url=self.base_url,␣
```

(continues on next page)

(continued from previous page)

```
↪organization=self.organization)

    def try_run(self, audio: AudioArtifact, prompts: Optional[list[str]] = None) ->␣
↪TextArtifact:
        additional_params = {}

        if prompts is not None:
            additional_params["prompt"] = ", ".join(prompts)

        transcription = self.client.audio.transcriptions.create(
            # Even though were not actually providing a file to the client, the API␣
↪still requires that we send a file
            # name. We set the file name to use the same format as the audio file so␣
↪that the API can reject
            # it if the format is unsupported.
            model=self.model,
            file=(f"a.{audio.format}", io.BytesIO(audio.value)),
            response_format="json",
            **additional_params,
        )

        return TextArtifact(value=transcription.text)
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `api_type = field(default=openai.api_type, kw_only=True)` class-attribute instance-attribute
- `api_version = field(default=openai.api_version, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `base_url = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `organization = field(default=openai.organization, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/audio_transcription/openai_audio_transcription_driver.py

```
@lazy_property()
def client(self) -> openai.OpenAI:
    return openai.OpenAI(api_key=self.api_key, base_url=self.base_url,␣
↪organization=self.organization)
```

try_run(audio, prompts=None)

Source Code in griptape/drivers/audio_transcription/openai_audio_transcription_driver.py

```
def try_run(self, audio: AudioArtifact, prompts: Optional[list[str]] = None) ->␣
↪TextArtifact:
```

(continues on next page)

(continued from previous page)

```
    additional_params = {}

    if prompts is not None:
        additional_params["prompt"] = ", ".join(prompts)

    transcription = self.client.audio.transcriptions.create(
        # Even though were not actually providing a file to the client, the API still
↪requires that we send a file
        # name. We set the file name to use the same format as the audio file so that
↪the API can reject
        # it if the format is unsupported.
        model=self.model,
        file=(f"a.{audio.format}", io.BytesIO(audio.value)),
        response_format="json",
        **additional_params,
    )

    return TextArtifact(value=transcription.text)
```

OpenAiChatPromptDriver

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<code class="language-python">	*Optional[str]*	An optional OpenAi API URL.

continues on next page

Table 102 – continued from previous page

Name	Type	Description
<code class="language-python">	*Optional[str]*	An optional OpenAi API key. If not provided, the *OPENAI_API_KEY* environment variable will be used.
<code class="language-python">	*Optional[str]*	An optional OpenAI organization. If not provided, the *OPENAI_ORG_ID* environment variable will be used.
<code class="language-python">	*OpenAI*	An *openai.OpenAI* client.

continues on next page

Table 102 – continued from previous page

Name	Type	Description
<code class="language-python">	*str*	An OpenAI model name.
<code class="language-python">	<code class="language-python">BaseTokenizer</code>	An *OpenAiTokenizer*.
<a href="./#griptape.drivers.OpenAiChatPromptDriver.user" title="user = field(default='', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.OpenAiChatPromptDriver.user)"><code class="language-python">	*str*	A user id. Can be used to track requests by user.

continues on next page

Table 102 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.OpenAiChatPromptDriver.response_format" title="response_format = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.OpenAiChatPromptDriver.response_format)"><code class="language-python">	*Optional[dict]*	An optional OpenAi Chat Completion response format. Currently only supports *json_object* which will enable OpenAi's JSON mode.
<a href="./#griptape.drivers.OpenAiChatPromptDriver.seed" title="seed = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.OpenAiChatPromptDriver.seed)"><code class="language-python">	*Optional[int]*	An optional OpenAi Chat Completion seed.

continues on next page

Table 102 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.OpenAiChatPromptDriver.ignored_exception_types" title="ignored_exception_types = field(default=Factory(lambda: (openai.BadRequestError, openai.AuthenticationError, openai.PermissionDeniedError, openai.NotFoundError, openai.ConflictError, openai.UnprocessableEntityError)), kw_only=True) class-attribute instance-attribute (griptape.drivers.OpenAiChatPromptDriver.ignored_exception_types)"><code class="language-python">	*tuple[type[Exception], ...]*	An optional tuple of exception types to ignore. Defaults to OpenAI's known exception types.
<a href="./#griptape.drivers.OpenAiChatPromptDriver.parallel_tool_calls" title="parallel_tool_calls = field(default=True, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.OpenAiChatPromptDriver.parallel_tool_calls)"><code class="language-python">	*bool*	A flag to enable parallel tool calls. Defaults to *True*.

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
@define
class OpenAiChatPromptDriver(BasePromptDriver):
    """OpenAI Chat Prompt Driver.

    Attributes:
        base_url: An optional OpenAi API URL.
        api_key: An optional OpenAi API key. If not provided, the `OPENAI_API_KEY`␣
```

(continues on next page)

(continued from previous page)

```
↪environment variable will be used.
        organization: An optional OpenAI organization. If not provided, the `OPENAI_
↪ORG_ID` environment variable will be used.
        client: An `openai.OpenAI` client.
        model: An OpenAI model name.
        tokenizer: An `OpenAiTokenizer`.
        user: A user id. Can be used to track requests by user.
        response_format: An optional OpenAi Chat Completion response format.␣
↪Currently only supports `json_object` which will enable OpenAis JSON mode.
        seed: An optional OpenAi Chat Completion seed.
        ignored_exception_types: An optional tuple of exception types to ignore.␣
↪Defaults to OpenAIs known exception types.
        parallel_tool_calls: A flag to enable parallel tool calls. Defaults to `True`.
    """

    base_url: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    organization: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    model: str = field(kw_only=True, metadata={"serializable": True})
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
    user: str = field(default="", kw_only=True, metadata={"serializable": True})
    response_format: Optional[dict] = field(
        default=None,
        kw_only=True,
        metadata={"serializable": True},
    )
    seed: Optional[int] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    tool_choice: str = field(default="auto", kw_only=True, metadata={"serializable":␣
↪False})
    reasoning_effort: Literal["low", "medium", "high"] = field(
        default="medium", kw_only=True, metadata={"serializable": True}
    )
    use_native_tools: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})
    structured_output_strategy: StructuredOutputStrategy = field(
        default="native", kw_only=True, metadata={"serializable": True}
    )
    parallel_tool_calls: bool = field(default=True, kw_only=True, metadata={
↪"serializable": True})
    ignored_exception_types: tuple[type[Exception], ...] = field(
        default=Factory(
            lambda: (
                openai.BadRequestError,
                openai.AuthenticationError,
```

(continues on next page)

(continued from previous page)

```
                openai.PermissionDeniedError,
                openai.NotFoundError,
                openai.ConflictError,
                openai.UnprocessableEntityError,
            ),
        ),
        kw_only=True,
    )
    modalities: list[str] = field(factory=list, kw_only=True, metadata={"serializable
↪": True})
    audio: dict = field(
        default=Factory(lambda: {"voice": "alloy", "format": "pcm16"}), kw_only=True,␣
↪metadata={"serializable": True}
    )
    _client: Optional[openai.OpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.OpenAI:
        return openai.OpenAI(
            base_url=self.base_url,
            api_key=self.api_key,
            organization=self.organization,
        )

    @property
    def is_reasoning_model(self) -> bool:
        return self.model.startswith("o")

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        params = self._base_params(prompt_stack)
        logger.debug(params)
        result = self.client.chat.completions.create(**params)

        logger.debug(result.model_dump())
        return self._to_message(result)

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        params = self._base_params(prompt_stack)
        logger.debug({"stream": True, **params})
        result = self.client.chat.completions.create(**params, stream=True)

        return self._to_delta_message_stream(result)

    def _to_message(self, result: ChatCompletion) -> Message:
        if len(result.choices) == 1:
            choice_message = result.choices[0].message

            message = Message(
```

(continues on next page)

(continued from previous page)

```
                content=self.__to_prompt_stack_message_content(choice_message),
                role=Message.ASSISTANT_ROLE,
            )
            if result.usage is not None:
                message.usage = Message.Usage(
                    input_tokens=result.usage.prompt_tokens,
                    output_tokens=result.usage.completion_tokens,
                )

            return message
        raise Exception("Completion with more than one choice is not supported yet.")

    def _to_delta_message_stream(self, result: Stream[ChatCompletionChunk]) ->␣
↪Iterator[DeltaMessage]:
        for message in result:
            if message.usage is not None:
                yield DeltaMessage(
                    usage=DeltaMessage.Usage(
                        input_tokens=message.usage.prompt_tokens,
                        output_tokens=message.usage.completion_tokens,
                    ),
                )
            if message.choices:
                choice = message.choices[0]
                delta = choice.delta

                content = self.__to_prompt_stack_delta_message_content(delta)

                if content is not None:
                    yield DeltaMessage(content=content)

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        params = {
            "model": self.model,
            **({"user": self.user} if self.user else {}),
            **({"seed": self.seed} if self.seed is not None else {}),
            **({"modalities": self.modalities} if self.modalities and not self.is_
↪reasoning_model else {}),
            **(
                {"reasoning_effort": self.reasoning_effort}
                if self.is_reasoning_model and self.model != "o1-mini"
                else {}
            ),
            **({"temperature": self.temperature} if not self.is_reasoning_model else
↪{}),
            **({"audio": self.audio} if "audio" in self.modalities else {}),
            **(
                {"stop": self.tokenizer.stop_sequences}
                if not self.is_reasoning_model and self.tokenizer.stop_sequences
                else {}
            ),
            **({"max_tokens": self.max_tokens} if self.max_tokens is not None else {}
```

(continues on next page)

(continued from previous page)

```
↪),
            **({"stream_options": {"include_usage": True}} if self.stream else {}),
            **self.extra_params,
        }

        if prompt_stack.tools and self.use_native_tools:
            params["tool_choice"] = self.tool_choice
            params["parallel_tool_calls"] = self.parallel_tool_calls

        if prompt_stack.output_schema is not None:
            if self.structured_output_strategy == "native":
                params["response_format"] = {
                    "type": "json_schema",
                    "json_schema": {
                        "name": "Output",
                        "schema": prompt_stack.to_output_json_schema(),
                        "strict": True,
                    },
                }
            elif self.structured_output_strategy == "tool" and self.use_native_tools:
                params["tool_choice"] = "required"

        if self.response_format is not None:
            if self.response_format == {"type": "json_object"}:
                params["response_format"] = self.response_format
                # JSON mode still requires a system message instructing the LLM to␣
↪output JSON.
                prompt_stack.add_system_message("Provide your response as a valid␣
↪JSON object.")
            else:
                params["response_format"] = self.response_format

        if prompt_stack.tools and self.use_native_tools:
            params["tools"] = self.__to_openai_tools(prompt_stack.tools)

        messages = self.__to_openai_messages(prompt_stack.messages)

        params["messages"] = messages

        return params

    def __to_openai_messages(self, messages: list[Message]) -> list[dict]:
        openai_messages = []

        for message in messages:
            # If the message only contains textual content we can send it as a single␣
↪content.
            if message.has_all_content_type(TextMessageContent):
                openai_messages.append({"role": self.__to_openai_role(message),
↪"content": message.to_text()})
            # Action results must be sent as separate messages.
            elif action_result_contents := message.get_content_
```

(continues on next page)

(continued from previous page)

```
↪type(ActionResultMessageContent):
                openai_messages.extend(
                    {
                        "role": self.__to_openai_role(message, action_result_content),
                        "content": self.__to_openai_message_content(action_result_
↪content),
                        "tool_call_id": action_result_content.action.tag,
                    }
                    for action_result_content in action_result_contents
                )

                if message.has_any_content_type(TextMessageContent):
                    openai_messages.append({"role": self.__to_openai_role(message),
↪"content": message.to_text()})
            else:
                openai_message = {
                    "role": self.__to_openai_role(message),
                    "content": [],
                }

                for content in message.content:
                    if isinstance(content, ActionCallMessageContent):
                        if "tool_calls" not in openai_message:
                            openai_message["tool_calls"] = []
                        openai_message["tool_calls"].append(self.__to_openai_message_
↪content(content))
                    elif (
                        isinstance(content, AudioMessageContent)
                        and message.is_assistant()
                        and time.time() < content.artifact.meta["expires_at"]
                    ):
                        # For assistant audio messages, we reference the audio id␣
↪instead of sending audio message content.
                        openai_message["audio"] = {
                            "id": content.artifact.meta["audio_id"],
                        }
                    else:
                        openai_message["content"].append(self.__to_openai_message_
↪content(content))

                # Some OpenAi-compatible services dont accept an empty array for␣
↪content
                if not openai_message["content"]:
                    del openai_message["content"]

                openai_messages.append(openai_message)

        return openai_messages

    def __to_openai_role(self, message: Message, message_content:␣
↪Optional[BaseMessageContent] = None) -> str:
        if message.is_system():
```

(continues on next page)

(continued from previous page)

```
            if self.is_reasoning_model:
                return "developer"
            return "system"
        if message.is_assistant():
            return "assistant"
        if isinstance(message_content, ActionResultMessageContent):
            return "tool"
        return "user"

    def __to_openai_tools(self, tools: list[BaseTool]) -> list[dict]:
        return [
            {
                "function": {
                    "name": tool.to_native_tool_name(activity),
                    "description": tool.activity_description(activity),
                    "parameters": tool.to_activity_json_schema(activity, "Parameters
↪Schema"),
                },
                "type": "function",
            }
            for tool in tools
            for activity in tool.activities()
        ]

    def __to_openai_message_content(self, content: BaseMessageContent) -> str | dict:
        if isinstance(content, TextMessageContent):
            return {"type": "text", "text": content.artifact.to_text()}
        if isinstance(content, ImageMessageContent):
            if isinstance(content.artifact, ImageArtifact):
                return {
                    "type": "image_url",
                    "image_url": {"url": f"data:{content.artifact.mime_type}; base64,
↪{content.artifact.base64}"},
                }
            if isinstance(content.artifact, ImageUrlArtifact):
                return {
                    "type": "image_url",
                    "image_url": {"url": content.artifact.value},
                }
            raise ValueError(f"Unsupported image artifact type: {type(content.
↪artifact)}")
        if isinstance(content, AudioMessageContent):
            artifact = content.artifact
            metadata = artifact.meta

            # If theres an expiration date, we can assume its an assistant message.
            if "expires_at" in metadata:
                # If its expired, we send the transcript instead.
                if time.time() >= metadata["expires_at"]:
                    return {
                        "type": "text",
                        "text": artifact.meta.get("transcript"),
```

(continues on next page)

(continued from previous page)

```
                    }
                # This should never occur, since a non-expired audio content
                # should have already been referenced by the audio id.
                raise ValueError("Assistant audio messages should be sent as audio␣
↪ids.")
            # If theres no expiration date, we can assume its a user message where we␣
↪send the audio every time.
            return {
                "type": "input_audio",
                "input_audio": {
                    "data": base64.b64encode(artifact.value).decode("utf-8"),
                    "format": artifact.format,
                },
            }
        if isinstance(content, ActionCallMessageContent):
            action = content.artifact.value

            return {
                "type": "function",
                "id": action.tag,
                "function": {"name": action.to_native_tool_name(), "arguments": json.
↪dumps(action.input)},
            }
        if isinstance(content, ActionResultMessageContent):
            return content.artifact.to_text()
        return {"type": "text", "text": content.artifact.to_text()}

    def __to_prompt_stack_message_content(self, response: ChatCompletionMessage) ->␣
↪list[BaseMessageContent]:
        content = []

        if response.content is not None:
            content.append(TextMessageContent(TextArtifact(response.content)))
        if hasattr(response, "audio") and response.audio is not None:
            content.append(
                AudioMessageContent(
                    AudioArtifact(
                        value=base64.b64decode(response.audio.data),
                        format="wav",
                        meta={
                            "audio_id": response.audio.id,
                            "transcript": response.audio.transcript,
                            "expires_at": response.audio.expires_at,
                        },
                    )
                )
            )
        if response.tool_calls is not None:
            content.extend(
                [
                    ActionCallMessageContent(
                        ActionArtifact(
```

(continues on next page)

(continued from previous page)

```
                            ToolAction(
                                tag=tool_call.id,
                                name=ToolAction.from_native_tool_name(tool_call.
↪function.name)[0],
                                path=ToolAction.from_native_tool_name(tool_call.
↪function.name)[1],
                                input=json.loads(tool_call.function.arguments),
                            ),
                        ),
                    )
                    for tool_call in response.tool_calls
                ],
            )

        return content

    def __to_prompt_stack_delta_message_content(self, content_delta: ChoiceDelta) ->␣
↪Optional[BaseDeltaMessageContent]:
        if content_delta.content is not None:
            return TextDeltaMessageContent(content_delta.content)
        if content_delta.tool_calls is not None:
            tool_calls = content_delta.tool_calls

            if len(tool_calls) == 1:
                tool_call = tool_calls[0]
                index = tool_call.index

                if tool_call.function is not None:
                    function_name = tool_call.function.name
                    return ActionCallDeltaMessageContent(
                        index=index,
                        tag=tool_call.id,
                        name=ToolAction.from_native_tool_name(function_name)[0] if␣
↪function_name else None,
                        path=ToolAction.from_native_tool_name(function_name)[1] if␣
↪function_name else None,
                        partial_input=tool_call.function.arguments,
                    )
                raise ValueError(f"Unsupported tool call delta: {tool_call}")
            raise ValueError(f"Unsupported tool call delta length: {len(tool_calls)}")
        # OpenAi doesnt have types for audio deltas so we need to use hasattr and␣
↪getattr.
        if hasattr(content_delta, "audio") and getattr(content_delta, "audio") is not␣
↪None:
            audio_chunk: dict = getattr(content_delta, "audio")
            return AudioDeltaMessageContent(
                id=audio_chunk.get("id"),
                data=audio_chunk.get("data"),
                expires_at=audio_chunk.get("expires_at"),
                transcript=audio_chunk.get("transcript"),
            )
        return None
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `audio = field(default=Factory(lambda: {'voice': 'alloy', 'format': 'pcm16'}), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `base_url = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `ignored_exception_types = field(default=Factory(lambda: (openai.BadRequestError, openai.AuthenticationError, openai.PermissionDeniedError, openai.NotFoundError, openai.ConflictError, openai.UnprocessableEntityError)), kw_only=True)` class-attribute instance-attribute
- `is_reasoning_model` property
- `modalities = field(factory=list, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `organization = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `parallel_tool_calls = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `reasoning_effort = field(default='medium', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `response_format = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `seed = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='native', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `tool_choice = field(default='auto', kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `use_native_tools = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `user = field(default='', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__to_openai_message_content(content)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def __to_openai_message_content(self, content: BaseMessageContent) -> str | dict:
    if isinstance(content, TextMessageContent):
        return {"type": "text", "text": content.artifact.to_text()}
```

(continues on next page)

(continued from previous page)

```
    if isinstance(content, ImageMessageContent):
        if isinstance(content.artifact, ImageArtifact):
            return {
                "type": "image_url",
                "image_url": {"url": f"data:{content.artifact.mime_type}; base64,
↪{content.artifact.base64}"},
            }
        if isinstance(content.artifact, ImageUrlArtifact):
            return {
                "type": "image_url",
                "image_url": {"url": content.artifact.value},
            }
        raise ValueError(f"Unsupported image artifact type: {type(content.artifact)}")
    if isinstance(content, AudioMessageContent):
        artifact = content.artifact
        metadata = artifact.meta

        # If theres an expiration date, we can assume its an assistant message.
        if "expires_at" in metadata:
            # If its expired, we send the transcript instead.
            if time.time() >= metadata["expires_at"]:
                return {
                    "type": "text",
                    "text": artifact.meta.get("transcript"),
                }
            # This should never occur, since a non-expired audio content
            # should have already been referenced by the audio id.
            raise ValueError("Assistant audio messages should be sent as audio ids.")
        # If theres no expiration date, we can assume its a user message where we␣
↪send the audio every time.
        return {
            "type": "input_audio",
            "input_audio": {
                "data": base64.b64encode(artifact.value).decode("utf-8"),
                "format": artifact.format,
            },
        }
    if isinstance(content, ActionCallMessageContent):
        action = content.artifact.value

        return {
            "type": "function",
            "id": action.tag,
            "function": {"name": action.to_native_tool_name(), "arguments": json.
↪dumps(action.input)},
        }
    if isinstance(content, ActionResultMessageContent):
        return content.artifact.to_text()
    return {"type": "text", "text": content.artifact.to_text()}
```

__to_openai_messages(messages)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def __to_openai_messages(self, messages: list[Message]) -> list[dict]:
    openai_messages = []

    for message in messages:
        # If the message only contains textual content we can send it as a single␣
↪content.
        if message.has_all_content_type(TextMessageContent):
            openai_messages.append({"role": self.__to_openai_role(message), "content
↪": message.to_text()})
        # Action results must be sent as separate messages.
        elif action_result_contents := message.get_content_
↪type(ActionResultMessageContent):
            openai_messages.extend(
                {
                    "role": self.__to_openai_role(message, action_result_content),
                    "content": self.__to_openai_message_content(action_result_
↪content),
                    "tool_call_id": action_result_content.action.tag,
                }
                for action_result_content in action_result_contents
            )

            if message.has_any_content_type(TextMessageContent):
                openai_messages.append({"role": self.__to_openai_role(message),
↪"content": message.to_text()})
        else:
            openai_message = {
                "role": self.__to_openai_role(message),
                "content": [],
            }

            for content in message.content:
                if isinstance(content, ActionCallMessageContent):
                    if "tool_calls" not in openai_message:
                        openai_message["tool_calls"] = []
                    openai_message["tool_calls"].append(self.__to_openai_message_
↪content(content))
                elif (
                    isinstance(content, AudioMessageContent)
                    and message.is_assistant()
                    and time.time() < content.artifact.meta["expires_at"]
                ):
                    # For assistant audio messages, we reference the audio id instead␣
↪of sending audio message content.
                    openai_message["audio"] = {
                        "id": content.artifact.meta["audio_id"],
                    }
                else:
                    openai_message["content"].append(self.__to_openai_message_
↪content(content))
```

(continues on next page)

(continued from previous page)

```
            # Some OpenAi-compatible services dont accept an empty array for content
            if not openai_message["content"]:
                del openai_message["content"]

            openai_messages.append(openai_message)

    return openai_messages
```

__to_openai_role(message, message_content=None)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def __to_openai_role(self, message: Message, message_content:␣
↪Optional[BaseMessageContent] = None) -> str:
    if message.is_system():
        if self.is_reasoning_model:
            return "developer"
        return "system"
    if message.is_assistant():
        return "assistant"
    if isinstance(message_content, ActionResultMessageContent):
        return "tool"
    return "user"
```

__to_openai_tools(tools)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def __to_openai_tools(self, tools: list[BaseTool]) -> list[dict]:
    return [
        {
            "function": {
                "name": tool.to_native_tool_name(activity),
                "description": tool.activity_description(activity),
                "parameters": tool.to_activity_json_schema(activity, "Parameters␣
↪Schema"),
            },
            "type": "function",
        }
        for tool in tools
        for activity in tool.activities()
    ]
```

__to_prompt_stack_delta_message_content(content_delta)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def __to_prompt_stack_delta_message_content(self, content_delta: ChoiceDelta) ->␣
↪Optional[BaseDeltaMessageContent]:
    if content_delta.content is not None:
        return TextDeltaMessageContent(content_delta.content)
```

(continues on next page)

(continued from previous page)

```
    if content_delta.tool_calls is not None:
        tool_calls = content_delta.tool_calls

        if len(tool_calls) == 1:
            tool_call = tool_calls[0]
            index = tool_call.index

            if tool_call.function is not None:
                function_name = tool_call.function.name
                return ActionCallDeltaMessageContent(
                    index=index,
                    tag=tool_call.id,
                    name=ToolAction.from_native_tool_name(function_name)[0] if␣
↪function_name else None,
                    path=ToolAction.from_native_tool_name(function_name)[1] if␣
↪function_name else None,
                    partial_input=tool_call.function.arguments,
                )
            raise ValueError(f"Unsupported tool call delta: {tool_call}")
        raise ValueError(f"Unsupported tool call delta length: {len(tool_calls)}")
    # OpenAi doesnt have types for audio deltas so we need to use hasattr and getattr.
    if hasattr(content_delta, "audio") and getattr(content_delta, "audio") is not␣
↪None:
        audio_chunk: dict = getattr(content_delta, "audio")
        return AudioDeltaMessageContent(
            id=audio_chunk.get("id"),
            data=audio_chunk.get("data"),
            expires_at=audio_chunk.get("expires_at"),
            transcript=audio_chunk.get("transcript"),
        )
    return None
```

__to_prompt_stack_message_content(response)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def __to_prompt_stack_message_content(self, response: ChatCompletionMessage) ->␣
↪list[BaseMessageContent]:
    content = []

    if response.content is not None:
        content.append(TextMessageContent(TextArtifact(response.content)))
    if hasattr(response, "audio") and response.audio is not None:
        content.append(
            AudioMessageContent(
                AudioArtifact(
                    value=base64.b64decode(response.audio.data),
                    format="wav",
                    meta={
                        "audio_id": response.audio.id,
                        "transcript": response.audio.transcript,
                        "expires_at": response.audio.expires_at,
```

(continues on next page)

(continued from previous page)

```
                    },
                )
            )
        )
    if response.tool_calls is not None:
        content.extend(
            [
                ActionCallMessageContent(
                    ActionArtifact(
                        ToolAction(
                            tag=tool_call.id,
                            name=ToolAction.from_native_tool_name(tool_call.function.
↪name)[0],
                            path=ToolAction.from_native_tool_name(tool_call.function.
↪name)[1],
                            input=json.loads(tool_call.function.arguments),
                        ),
                    ),
                )
                for tool_call in response.tool_calls
            ],
        )

    return content
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    params = {
        "model": self.model,
        **({"user": self.user} if self.user else {}),
        **({"seed": self.seed} if self.seed is not None else {}),
        **({"modalities": self.modalities} if self.modalities and not self.is_
↪reasoning_model else {}),
        **(
            {"reasoning_effort": self.reasoning_effort}
            if self.is_reasoning_model and self.model != "o1-mini"
            else {}
        ),
        **({"temperature": self.temperature} if not self.is_reasoning_model else {}),
        **({"audio": self.audio} if "audio" in self.modalities else {}),
        **(
            {"stop": self.tokenizer.stop_sequences}
            if not self.is_reasoning_model and self.tokenizer.stop_sequences
            else {}
        ),
        **({"max_tokens": self.max_tokens} if self.max_tokens is not None else {}),
        **({"stream_options": {"include_usage": True}} if self.stream else {}),
        **self.extra_params,
    }
```

(continues on next page)

(continued from previous page)

```
    if prompt_stack.tools and self.use_native_tools:
        params["tool_choice"] = self.tool_choice
        params["parallel_tool_calls"] = self.parallel_tool_calls

    if prompt_stack.output_schema is not None:
        if self.structured_output_strategy == "native":
            params["response_format"] = {
                "type": "json_schema",
                "json_schema": {
                    "name": "Output",
                    "schema": prompt_stack.to_output_json_schema(),
                    "strict": True,
                },
            }
        elif self.structured_output_strategy == "tool" and self.use_native_tools:
            params["tool_choice"] = "required"

    if self.response_format is not None:
        if self.response_format == {"type": "json_object"}:
            params["response_format"] = self.response_format
            # JSON mode still requires a system message instructing the LLM to output␣
↪JSON.
            prompt_stack.add_system_message("Provide your response as a valid JSON␣
↪object.")
        else:
            params["response_format"] = self.response_format

    if prompt_stack.tools and self.use_native_tools:
        params["tools"] = self.__to_openai_tools(prompt_stack.tools)

    messages = self.__to_openai_messages(prompt_stack.messages)

    params["messages"] = messages

    return params
```

_to_delta_message_stream(result)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def _to_delta_message_stream(self, result: Stream[ChatCompletionChunk]) ->␣
↪Iterator[DeltaMessage]:
    for message in result:
        if message.usage is not None:
            yield DeltaMessage(
                usage=DeltaMessage.Usage(
                    input_tokens=message.usage.prompt_tokens,
                    output_tokens=message.usage.completion_tokens,
                ),
            )
        if message.choices:
```

(continues on next page)

(continued from previous page)

```
            choice = message.choices[0]
            delta = choice.delta

            content = self.__to_prompt_stack_delta_message_content(delta)

            if content is not None:
                yield DeltaMessage(content=content)
```

_to_message(result)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def _to_message(self, result: ChatCompletion) -> Message:
    if len(result.choices) == 1:
        choice_message = result.choices[0].message

        message = Message(
            content=self.__to_prompt_stack_message_content(choice_message),
            role=Message.ASSISTANT_ROLE,
        )
        if result.usage is not None:
            message.usage = Message.Usage(
                input_tokens=result.usage.prompt_tokens,
                output_tokens=result.usage.completion_tokens,
            )

        return message
    raise Exception("Completion with more than one choice is not supported yet.")
```

client()

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
@lazy_property()
def client(self) -> openai.OpenAI:
    return openai.OpenAI(
        base_url=self.base_url,
        api_key=self.api_key,
        organization=self.organization,
    )
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    params = self._base_params(prompt_stack)
    logger.debug(params)
    result = self.client.chat.completions.create(**params)
```

(continues on next page)

(continued from previous page)

```
    logger.debug(result.model_dump())
    return self._to_message(result)
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    params = self._base_params(prompt_stack)
    logger.debug({"stream": True, **params})
    result = self.client.chat.completions.create(**params, stream=True)

    return self._to_delta_message_stream(result)
```

OpenAiEmbeddingDriver

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.OpenAiEmbeddingDriver.model" title="model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.OpenAiEmbeddingDriver.model)"><code class="language-python">	*str*	OpenAI embedding model name. Defaults to *text-embedding-3-small*.

continues on next page

Table 103 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.OpenAiEmbeddingDriver.base_url" title="base_url = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.OpenAiEmbeddingDriver.base_url)"><code class="language-python">	*Optional[str]*	API URL. Defaults to OpenAI's v1 API URL.
<a href="./#griptape.drivers.OpenAiEmbeddingDriver.api_key" title="api_key = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.OpenAiEmbeddingDriver.api_key)"><code class="language-python">	*Optional[str]*	API key to pass directly. Defaults to *OPENAI_API_KEY* environment variable.
<a href="./#griptape.drivers.OpenAiEmbeddingDriver.organization" title="organization = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.OpenAiEmbeddingDriver.organization)"><code class="language-python">	*Optional[str]*	OpenAI organization. Defaults to 'OPENAI_ORGANIZATION' environment variable.

continues on next page

Table 103 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.OpenAiEmbeddingDriver.tokenizer" title="tokenizer = field(default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.drivers.OpenAiEmbeddingDriver.tokenizer)"><code class="language-python">	<a href="../tokenizers#griptape.tokenizers.OpenAiTokenizer" title="OpenAiTokenizer (griptape.tokenizers.OpenAiTokenizer)"><code class="language-python">OpenAiTokenizer</code></	Optionally provide custom *OpenAiTokenizer*.
<a href="./#griptape.drivers.OpenAiEmbeddingDriver.client" title="client() (griptape.drivers.OpenAiEmbeddingDriver.client)"><code class="language-python">	*OpenAI*	Optionally provide custom *openai.OpenAI* client.
azure_deployment	*OpenAI*	An Azure OpenAi deployment id.
azure_endpoint	*OpenAI*	An Azure OpenAi endpoint.
azure_ad_token	*OpenAI*	An optional Azure Active Directory token.
azure_ad_token_provider	*OpenAI*	An optional Azure Active Directory token provider.
api_version	*OpenAI*	An Azure OpenAi API version.

Source Code in griptape/drivers/embedding/openai_embedding_driver.py

```
@define
class OpenAiEmbeddingDriver(BaseEmbeddingDriver):
    """OpenAI Embedding Driver.

    Attributes:
        model: OpenAI embedding model name. Defaults to `text-embedding-3-small`.
        base_url: API URL. Defaults to OpenAIs v1 API URL.
        api_key: API key to pass directly. Defaults to `OPENAI_API_KEY` environment␣
↪variable.
        organization: OpenAI organization. Defaults to OPENAI_ORGANIZATION␣
```

(continues on next page)

(continued from previous page)

```
↪environment variable.
        tokenizer: Optionally provide custom `OpenAiTokenizer`.
        client: Optionally provide custom `openai.OpenAI` client.
        azure_deployment: An Azure OpenAi deployment id.
        azure_endpoint: An Azure OpenAi endpoint.
        azure_ad_token: An optional Azure Active Directory token.
        azure_ad_token_provider: An optional Azure Active Directory token provider.
        api_version: An Azure OpenAi API version.
    """

    DEFAULT_MODEL = "text-embedding-3-small"

    model: str = field(default=DEFAULT_MODEL, kw_only=True, metadata={"serializable":␣
↪True})
    base_url: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    organization: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    tokenizer: OpenAiTokenizer = field(
        default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
    _client: Optional[openai.OpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.OpenAI:
        return openai.OpenAI(api_key=self.api_key, base_url=self.base_url,␣
↪organization=self.organization)

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        # Address a performance issue in older ada models
        # https://github.com/openai/openai-python/issues/418#issuecomment-1525939500
        if self.model.endswith("001"):
            chunk = chunk.replace("\n", " ")
        return self.client.embeddings.create(**self._params(chunk)).data[0].embedding

    def _params(self, chunk: str) -> dict:
        return {"input": chunk, "model": self.model}
```

- `DEFAULT_MODEL = 'text-embedding-3-small'` class-attribute instance-attribute
- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `base_url = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

- `model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `organization = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute

_params(chunk)

Source Code in griptape/drivers/embedding/openai_embedding_driver.py

```
def _params(self, chunk: str) -> dict:
    return {"input": chunk, "model": self.model}
```

client()

Source Code in griptape/drivers/embedding/openai_embedding_driver.py

```
@lazy_property()
def client(self) -> openai.OpenAI:
    return openai.OpenAI(api_key=self.api_key, base_url=self.base_url,␣
↪organization=self.organization)
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/openai_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    # Address a performance issue in older ada models
    # https://github.com/openai/openai-python/issues/418#issuecomment-1525939500
    if self.model.endswith("001"):
        chunk = chunk.replace("\n", " ")
    return self.client.embeddings.create(**self._params(chunk)).data[0].embedding
```

OpenAiImageGenerationDriver

Bases: *BaseImageGenerationDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.BaseImageGenerationDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.OpenAiImageGenerationDriver.model)"><cod class="language-python">	*str*	OpenAI model, for example 'dall-e-2' or 'dall-e-3'.
<a href="./#griptape.drivers.OpenAiImageGenerationDriver.api_type" title="api_type = field(default=openai.api_type, kw_only=True) class-attribute instance-attribute (griptape.drivers.OpenAiImageGenerationDriver.api_type)"><c class="language-python">	*Optional[str]*	OpenAI API type, for example 'open_ai' or 'azure'.
<a href="./#griptape.drivers.OpenAiImageGenerationDriver.api_version" title="api_version = field(default=openai.api_version, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.OpenAiImageGenerationDriver.api_version)"><code class="language-python">	*Optional[str]*	API version.

continues on next page

Table 104 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.OpenAiImageGenerationDriver.base_url" title="base_url = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.OpenAiImageGenerationDriver.base_url)"><c class="language-python">	*Optional[str]*	API URL.
<a href="./#griptape.drivers.OpenAiImageGenerationDriver.api_key" title="api_key = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.OpenAiImageGenerationDriver.api_key)"><cc class="language-python">	*Optional[str]*	OpenAI API key.
<a href="./#griptape.drivers.OpenAiImageGenerationDriver.organization" title="organization = field(default=openai.organization, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.OpenAiImageGenerationDriver.organization)"><code class="language-python">	*Optional[str]*	OpenAI organization ID.

continues on next page

Table 104 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.OpenAiImageGenerationDriver.style" title="style = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['dall-e-3']}) class-attribute instance-attribute (griptape.drivers.OpenAiImageGenerationDriver.style)"><code class="language-python">	*Optional[Literal['vivid', 'natural']]*	Optional and only supported for dall-e-3, can be either 'vivid' or 'natural'.
<a href="./#griptape.drivers.OpenAiImageGenerationDriver.quality" title="quality = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.OpenAiImageGenerationDriver.quality)"><code class="language-python">	*Optional[Literal['standard', 'hd', 'low', 'medium', 'high', 'auto']]*	Optional and only supported for dall-e-3. Accepts 'standard', 'hd'.
<a href="./#griptape.drivers.OpenAiImageGenerationDriver.image_size" title="image_size = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.OpenAiImageGenerationDriver.image_size)"><code class="language-python">	*Optional[Literal['256x256', '512x512', '1024x1024', '1024x1792', '1792x1024']]*	Size of the generated image. Must be one of the following, depending on the requested model: dall-e-2: [256x256, 512x512, 1024x1024] dall-e-3: [1024x1024, 1024x1792, 1792x1024] gpt-image-1: [1024x1024, 1536x1024, 1024x1536, auto]

continues on next page

Table 104 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.OpenAiImageGenerationDriver.response_format" title="response_format = field(default='b64_json', kw_only=True, metadata={'serializable': True, 'model_denylist': ['gpt-image-1']}) class-attribute instance-attribute (griptape.drivers.OpenAiImageGenerationDriver.response_format)"><code class="language-python">	*Literal['b64_json']*	The response format. Currently only supports 'b64_json' which will return a base64 encoded image in a JSON object.
<a href="./#griptape.drivers.OpenAiImageGenerationDriver.background" title="background = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']}) class-attribute instance-attribute (griptape.drivers.OpenAiImageGenerationDriver.background)"><code class="language-python">	*Optional[Literal['transparent', 'opaque', 'auto']]*	Optional and only supported for gpt-image-1. Can be either 'transparent', 'opaque', or 'auto'.

continues on next page

Table 104 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.OpenAiImageGenerationDriver.moderation" title="moderation = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']}) class-attribute instance-attribute (griptape.drivers.OpenAiImageGenerationDriver.moderation)"><code class="language-python">	*Optional[Literal['low', 'auto']]*	Optional and only supported for gpt-image-1. Can be either 'low' or 'auto'.
<a href="./#griptape.drivers.OpenAiImageGenerationDriver.output_compression" title="output_compression = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']}) class-attribute instance-attribute (griptape.drivers.OpenAiImageGenerationDriver.output_compression)"><code class="language-python">	*Optional[int]*	Optional and only supported for gpt-image-1. Can be an integer between 0 and 100.

continues on next page

Table 104 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.OpenAiImageGenerationDriver.output_format" title="output_format = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']}) class-attribute instance-attribute (griptape.drivers.OpenAiImageGenerationDriver.output_format)"><code class="language-python">	*Optional[Literal['png', 'jpeg']]*	Optional and only supported for gpt-image-1. Can be either 'png' or 'jpeg'.

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
@define
class OpenAiImageGenerationDriver(BaseImageGenerationDriver):
    """Driver for the OpenAI image generation API.

    Attributes:
        model: OpenAI model, for example dall-e-2 or dall-e-3.
        api_type: OpenAI API type, for example open_ai or azure.
        api_version: API version.
        base_url: API URL.
        api_key: OpenAI API key.
        organization: OpenAI organization ID.
        style: Optional and only supported for dall-e-3, can be either vivid or␣
↪natural.
        quality: Optional and only supported for dall-e-3. Accepts standard, hd.
        image_size: Size of the generated image. Must be one of the following,␣
↪depending on the requested model:
            dall-e-2: [256x256, 512x512, 1024x1024]
            dall-e-3: [1024x1024, 1024x1792, 1792x1024]
            gpt-image-1: [1024x1024, 1536x1024, 1024x1536, auto]
        response_format: The response format. Currently only supports b64_json which␣
↪will return
            a base64 encoded image in a JSON object.
        background: Optional and only supported for gpt-image-1. Can be either␣
↪transparent, opaque, or auto.
        moderation: Optional and only supported for gpt-image-1. Can be either low or␣
↪auto.
        output_compression: Optional and only supported for gpt-image-1. Can be an␣
↪integer between 0 and 100.
        output_format: Optional and only supported for gpt-image-1. Can be either png␣
```

(continues on next page)

(continued from previous page)

```
↪or jpeg.
    """

    api_type: Optional[str] = field(default=openai.api_type, kw_only=True)
    api_version: Optional[str] = field(default=openai.api_version, kw_only=True,␣
↪metadata={"serializable": True})
    base_url: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    organization: Optional[str] = field(default=openai.organization, kw_only=True,␣
↪metadata={"serializable": True})
    style: Optional[Literal["vivid", "natural"]] = field(
        default=None, kw_only=True, metadata={"serializable": True, "model_allowlist
↪": ["dall-e-3"]}
    )
    quality: Optional[Literal["standard", "hd", "low", "medium", "high", "auto"]] =␣
↪field(
        default=None,
        kw_only=True,
        metadata={"serializable": True},
    )
    image_size: Optional[Literal["256x256", "512x512", "1024x1024", "1024x1792",
↪"1792x1024"]] = field(
        default=None,
        kw_only=True,
        metadata={"serializable": True},
    )
    response_format: Literal["b64_json"] = field(
        default="b64_json",
        kw_only=True,
        metadata={"serializable": True, "model_denylist": ["gpt-image-1"]},
    )
    background: Optional[Literal["transparent", "opaque", "auto"]] = field(
        default=None,
        kw_only=True,
        metadata={"serializable": True, "model_allowlist": ["gpt-image-1"]},
    )
    moderation: Optional[Literal["low", "auto"]] = field(
        default=None,
        kw_only=True,
        metadata={"serializable": True, "model_allowlist": ["gpt-image-1"]},
    )
    output_compression: Optional[int] = field(
        default=None,
        kw_only=True,
        metadata={"serializable": True, "model_allowlist": ["gpt-image-1"]},
    )
    output_format: Optional[Literal["png", "jpeg"]] = field(
        default=None,
        kw_only=True,
        metadata={"serializable": True, "model_allowlist": ["gpt-image-1"]},
```

(continues on next page)

(continued from previous page)

```
    )
    _client: Optional[openai.OpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @image_size.validator  # pyright: ignore[reportAttributeAccessIssue,
↪reportOptionalMemberAccess]
    def validate_image_size(self, attribute: str, value: str | None) -> None:
        """Validates the image size based on the model.

        Must be one of `1024x1024`, `1536x1024` (landscape), `1024x1536` (portrait),
↪or `auto` (default value) for
        `gpt-image-1`, one of `256x256`, `512x512`, or `1024x1024` for `dall-e-2`, and
        one of `1024x1024`, `1792x1024`, or `1024x1792` for `dall-e-3`.

        """
        if value is None:
            return

        if self.model.startswith("gpt-image"):
            allowed_sizes = ("1024x1024", "1536x1024", "1024x1536", "auto")
        elif self.model == "dall-e-2":
            allowed_sizes = ("256x256", "512x512", "1024x1024")
        elif self.model == "dall-e-3":
            allowed_sizes = ("1024x1024", "1792x1024", "1024x1792")
        else:
            raise NotImplementedError(f"Image size validation not implemented for
↪model {self.model}")

        if value is not None and value not in allowed_sizes:
            raise ValueError(f"Image size, {value}, must be one of the following:
↪{allowed_sizes}")

    @lazy_property()
    def client(self) -> openai.OpenAI:
        return openai.OpenAI(api_key=self.api_key, base_url=self.base_url,
↪organization=self.organization)

    def try_text_to_image(self, prompts: list[str], negative_prompts:
↪Optional[list[str]] = None) -> ImageArtifact:
        prompt = ", ".join(prompts)

        response = self.client.images.generate(
            model=self.model,
            prompt=prompt,
            n=1,
            **self._build_model_params(
                {
                    "size": "image_size",
                    "quality": "quality",
                    "style": "style",
                    "response_format": "response_format",
```

(continues on next page)

(continued from previous page)

```
                    "background": "background",
                    "moderation": "moderation",
                    "output_compression": "output_compression",
                    "output_format": "output_format",
                }
            ),
        )

        return self._parse_image_response(response, prompt)

    def try_image_variation(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        """Creates a variation of an image.

        Only supported by for dall-e-2. Requires image size to be one of the␣
↪following:
            [256x256, 512x512, 1024x1024]
        """
        if self.model != "dall-e-2":
            raise NotImplementedError("Image variation only supports dall-e-2")
        response = self.client.images.create_variation(
            image=image.value,
            n=1,
            response_format=self.response_format,
            size=self.image_size,  # pyright: ignore[reportArgumentType]
        )

        return self._parse_image_response(response, "")

    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        prompt = ", ".join(prompts)
        response = self.client.images.edit(
            prompt=prompt,
            image=image.value,
            mask=mask.value,
            **self._build_model_params(
                {
                    "size": "image_size",
                    "response_format": "response_format",
                }
            ),
        )
```

(continues on next page)

(continued from previous page)

```
        return self._parse_image_response(response, prompt)

    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪outpainting")

    def _parse_image_response(self, response: ImagesResponse, prompt: str) ->␣
↪ImageArtifact:
        from griptape.loaders.image_loader import ImageLoader

        if response.data is None or response.data[0] is None or response.data[0].b64_
↪json is None:
            raise Exception("Failed to generate image")

        image_data = base64.b64decode(response.data[0].b64_json)

        image_artifact = ImageLoader().parse(image_data)

        image_artifact.meta["prompt"] = prompt
        image_artifact.meta["model"] = self.model

        return image_artifact

    def _build_model_params(self, values: dict) -> dict:
        """Builds parameters while considering field metadata and None values.

        Args:
            values: A dictionary mapping parameter names to field names.

        Field will be added to the params dictionary if all conditions are met:
            - The field value is not None
            - The model_allowlist is None or the model is in the allowlist
            - The model_denylist is None or the model is not in the denylist
        """
        params = {}

        fields = fields_dict(self.__class__)
        for param_name, field_name in values.items():
            metadata = fields[field_name].metadata
            model_allowlist = metadata.get("model_allowlist")
            model_denylist = metadata.get("model_denylist")

            field_value = getattr(self, field_name, None)

            allowlist_condition = model_allowlist is None or self.model in model_
```

(continues on next page)

(continued from previous page)

```
↪allowlist
            denylist_condition = model_denylist is None or self.model not in model_
↪denylist

            if field_value is not None and allowlist_condition and denylist_condition:
                params[param_name] = field_value
        return params
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `api_type = field(default=openai.api_type, kw_only=True)` class-attribute instance-attribute
- `api_version = field(default=openai.api_version, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `background = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']})` class-attribute instance-attribute
- `base_url = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `image_size = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `moderation = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']})` class-attribute instance-attribute
- `organization = field(default=openai.organization, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output_compression = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']})` class-attribute instance-attribute
- `output_format = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']})` class-attribute instance-attribute
- `quality = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `response_format = field(default='b64_json', kw_only=True, metadata={'serializable': True, 'model_denylist': ['gpt-image-1']})` class-attribute instance-attribute
- `style = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['dall-e-3']})` class-attribute instance-attribute

_build_model_params(values)

Builds parameters while considering field metadata and None values.

Parameters

Name	Type	Description	Default
values	*dict*	A dictionary mapping parameter names to field names. 	*required*

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
def _build_model_params(self, values: dict) -> dict:
    """Builds parameters while considering field metadata and None values.

    Args:
        values: A dictionary mapping parameter names to field names.

    Field will be added to the params dictionary if all conditions are met:
        - The field value is not None
        - The model_allowlist is None or the model is in the allowlist
        - The model_denylist is None or the model is not in the denylist
    """
    params = {}

    fields = fields_dict(self.__class__)
    for param_name, field_name in values.items():
        metadata = fields[field_name].metadata
        model_allowlist = metadata.get("model_allowlist")
        model_denylist = metadata.get("model_denylist")

        field_value = getattr(self, field_name, None)

        allowlist_condition = model_allowlist is None or self.model in model_allowlist
        denylist_condition = model_denylist is None or self.model not in model_
↪denylist

        if field_value is not None and allowlist_condition and denylist_condition:
            params[param_name] = field_value
    return params
```

_parse_image_response(response, prompt)

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
def _parse_image_response(self, response: ImagesResponse, prompt: str) ->␣
↪ImageArtifact:
    from griptape.loaders.image_loader import ImageLoader

    if response.data is None or response.data[0] is None or response.data[0].b64_json␣
↪is None:
        raise Exception("Failed to generate image")

    image_data = base64.b64decode(response.data[0].b64_json)

    image_artifact = ImageLoader().parse(image_data)
```

(continues on next page)

(continued from previous page)

```
    image_artifact.meta["prompt"] = prompt
    image_artifact.meta["model"] = self.model

    return image_artifact
```

client()

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
@lazy_property()
def client(self) -> openai.OpenAI:
    return openai.OpenAI(api_key=self.api_key, base_url=self.base_url,␣
↪organization=self.organization)
```

try_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    prompt = ", ".join(prompts)
    response = self.client.images.edit(
        prompt=prompt,
        image=image.value,
        mask=mask.value,
        **self._build_model_params(
            {
                "size": "image_size",
                "response_format": "response_format",
            }
        ),
    )

    return self._parse_image_response(response, prompt)
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
def try_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
```

(continues on next page)

(continued from previous page)

```
    raise NotImplementedError(f"{self.__class__.__name__} does not support outpainting
↪")
```

try_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
def try_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    """Creates a variation of an image.

    Only supported by for dall-e-2. Requires image size to be one of the following:
        [256x256, 512x512, 1024x1024]
    """
    if self.model != "dall-e-2":
        raise NotImplementedError("Image variation only supports dall-e-2")
    response = self.client.images.create_variation(
        image=image.value,
        n=1,
        response_format=self.response_format,
        size=self.image_size,  # pyright: ignore[reportArgumentType]
    )

    return self._parse_image_response(response, "")
```

try_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    prompt = ", ".join(prompts)

    response = self.client.images.generate(
        model=self.model,
        prompt=prompt,
        n=1,
        **self._build_model_params(
            {
                "size": "image_size",
                "quality": "quality",
                "style": "style",
                "response_format": "response_format",
                "background": "background",
                "moderation": "moderation",
                "output_compression": "output_compression",
                "output_format": "output_format",
            }
```

(continues on next page)

(continued from previous page)

```
        ),
    )

    return self._parse_image_response(response, prompt)
```

validate_image_size(attribute, value)

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
@image_size.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_image_size(self, attribute: str, value: str | None) -> None:
    """Validates the image size based on the model.

    Must be one of `1024x1024`, `1536x1024` (landscape), `1024x1536` (portrait), or␣
↪`auto` (default value) for
    `gpt-image-1`, one of `256x256`, `512x512`, or `1024x1024` for `dall-e-2`, and
    one of `1024x1024`, `1792x1024`, or `1024x1792` for `dall-e-3`.

    """
    if value is None:
        return

    if self.model.startswith("gpt-image"):
        allowed_sizes = ("1024x1024", "1536x1024", "1024x1536", "auto")
    elif self.model == "dall-e-2":
        allowed_sizes = ("256x256", "512x512", "1024x1024")
    elif self.model == "dall-e-3":
        allowed_sizes = ("1024x1024", "1792x1024", "1024x1792")
    else:
        raise NotImplementedError(f"Image size validation not implemented for model
↪{self.model}")

    if value is not None and value not in allowed_sizes:
        raise ValueError(f"Image size, {value}, must be one of the following:
↪{allowed_sizes}")
```

OpenAiTextToSpeechDriver

Bases: *BaseTextToSpeechDriver* “)

Source Code in griptape/drivers/text_to_speech/openai_text_to_speech_driver.py

```
@define
class OpenAiTextToSpeechDriver(BaseTextToSpeechDriver):
    model: str = field(default="tts-1", kw_only=True, metadata={"serializable": True})
    voice: Literal["alloy", "echo", "fable", "onyx", "nova", "shimmer"] = field(
        default="alloy",
        kw_only=True,
        metadata={"serializable": True},
    )
    format: Literal["mp3", "opus", "aac", "flac"] = field(default="mp3", kw_only=True,
```

(continues on next page)

(continued from previous page)

```
↪ metadata={"serializable": True})
    api_type: Optional[str] = field(default=openai.api_type, kw_only=True)
    api_version: Optional[str] = field(default=openai.api_version, kw_only=True,␣
↪metadata={"serializable": True})
    base_url: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    api_key: Optional[str] = field(default=None, kw_only=True)
    organization: Optional[str] = field(default=openai.organization, kw_only=True,␣
↪metadata={"serializable": True})
    _client: Optional[openai.OpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.OpenAI:
        return openai.OpenAI(
            api_key=self.api_key,
            base_url=self.base_url,
            organization=self.organization,
        )

    def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
        response = self.client.audio.speech.create(
            input=". ".join(prompts),
            voice=self.voice,
            model=self.model,
            response_format=self.format,
        )

        return AudioArtifact(value=response.content, format=self.format)
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True)` class-attribute instance-attribute
- `api_type = field(default=openai.api_type, kw_only=True)` class-attribute instance-attribute
- `api_version = field(default=openai.api_version, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `base_url = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `format = field(default='mp3', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(default='tts-1', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `organization = field(default=openai.organization, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `voice = field(default='alloy', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/text_to_speech/openai_text_to_speech_driver.py

```
@lazy_property()
def client(self) -> openai.OpenAI:
    return openai.OpenAI(
        api_key=self.api_key,
        base_url=self.base_url,
        organization=self.organization,
    )
```

try_text_to_audio(prompts)

Source Code in griptape/drivers/text_to_speech/openai_text_to_speech_driver.py

```
def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
    response = self.client.audio.speech.create(
        input=". ".join(prompts),
        voice=self.voice,
        model=self.model,
        response_format=self.format,
    )

    return AudioArtifact(value=response.content, format=self.format)
```

OpenSearchVectorStoreDriver

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.OpenSearchVectorStoreDriver.host" title="host = field(kw_only=True, metadata={'serializable': True})">class-attribute instance-attribute (griptape.drivers.OpenSearchVector-StoreDriver.host)"><code class="language-python">	*str*	The host of the OpenSearch cluster.

continues on next page

Table 106 – continued from previous page

Name	Type	Description
<a href="./#grip-tape.drivers.OpenSearch\torStoreDriver.port" title="port = field(default=443, kw_only=True, meta-data={'serializable': True}) class-attribute instance-attribute (grip-tape.drivers.OpenSearch\tor-StoreDriver.port)"><code class="language-python">	*int*	The port of the OpenSearch cluster.
<a href="./#grip-tape.drivers.OpenSearch\tor-StoreDriver.http_auth" title="http_auth = field(default=None, kw_only=True, meta-data={'serializable': True}) class-attribute instance-attribute (grip-tape.drivers.OpenSearch\tor-StoreDriver.http_auth)"><class="language-python">	*str I tuple[str, Optional[str]]*	The HTTP authentication credentials to use.
<a href="./#grip-tape.drivers.OpenSearch\torStoreDriver.use_ssl" title="use_ssl = field(default=True, kw_only=True, meta-data={'serializable': True}) class-attribute instance-attribute (grip-tape.drivers.OpenSearch\tor-StoreDriver.use_ssl)"><c class="language-python">	*bool*	Whether to use SSL.

continues on next page

Table 106 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.OpenSearchVectorStoreDriver.verify_certs" title="verify_certs = field(default=True, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.OpenSearchVectorStoreDriver.verify_certs)"><code class="language-python">	*bool*	Whether to verify SSL certificates.
<a href="./#griptape.drivers.OpenSearchVectorStoreDriver.index_name" title="index_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.OpenSearchVectorStoreDriver.index_name)"><code class="language-python">	*str*	The name of the index to use.

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
@define
class OpenSearchVectorStoreDriver(BaseVectorStoreDriver):
    """A Vector Store Driver for OpenSearch.

    Attributes:
        host: The host of the OpenSearch cluster.
        port: The port of the OpenSearch cluster.
        http_auth: The HTTP authentication credentials to use.
        use_ssl: Whether to use SSL.
        verify_certs: Whether to verify SSL certificates.
        index_name: The name of the index to use.
    """

    host: str = field(kw_only=True, metadata={"serializable": True})
    port: int = field(default=443, kw_only=True, metadata={"serializable": True})
    http_auth: str | tuple[str, Optional[str]] = field(default=None, kw_only=True,␣
↪metadata={"serializable": True})
    use_ssl: bool = field(default=True, kw_only=True, metadata={"serializable": True})
    verify_certs: bool = field(default=True, kw_only=True, metadata={"serializable":␣
```

(continues on next page)

(continued from previous page)

```
↪True})
    index_name: str = field(kw_only=True, metadata={"serializable": True})
    _client: Optional[OpenSearch] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> OpenSearch:
        opensearchpy = import_optional_dependency("opensearchpy")

        return opensearchpy.OpenSearch(
            hosts=[{"host": self.host, "port": self.port}],
            http_auth=self.http_auth,
            use_ssl=self.use_ssl,
            verify_certs=self.verify_certs,
            connection_class=opensearchpy.RequestsHttpConnection,
        )

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        """Inserts or updates a vector in OpenSearch.

        If a vector with the given vector ID already exists, it is updated; otherwise,
↪ a new vector is inserted.
        Metadata associated with the vector can also be provided.
        """
        vector_id = vector_id or utils.str_to_hash(str(vector))
        doc = {"vector": vector, "namespace": namespace, "metadata": meta}
        doc.update(kwargs)
        response = self.client.index(index=self.index_name, id=vector_id, body=doc)

        return response["_id"]

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        """Retrieves a specific vector entry from OpenSearch based on its identifier␣
↪and optional namespace.

        Returns:
            If the entry is found, it returns an instance of BaseVectorStoreDriver.
↪Entry; otherwise, None is returned.
        """
        try:
            query = {"bool": {"must": [{"term": {"_id": vector_id}}]}}

            if namespace:
```

(continues on next page)

(continued from previous page)

```
                query["bool"]["must"].append({"term": {"namespace": namespace}})

            response = self.client.search(index=self.index_name, body={"query": query,
↪ "size": 1})

            if response["hits"]["total"]["value"] > 0:
                vector_data = response["hits"]["hits"][0]["_source"]
                return BaseVectorStoreDriver.Entry(
                    id=vector_id,
                    meta=vector_data.get("metadata"),
                    vector=vector_data.get("vector"),
                    namespace=vector_data.get("namespace"),
                )
            return None
        except Exception as e:
            logging.exception("Error while loading entry: %s", e)
            return None

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Retrieves all vector entries from OpenSearch that match the optional␣
↪namespace.

        Returns:
            A list of BaseVectorStoreDriver.Entry objects.
        """
        query_body = {"size": 10000, "query": {"match_all": {}}}

        if namespace:
            query_body["query"] = {"match": {"namespace": namespace}}

        response = self.client.search(index=self.index_name, body=query_body)

        return [
            BaseVectorStoreDriver.Entry(
                id=hit["_id"],
                vector=hit["_source"].get("vector"),
                meta=hit["_source"].get("metadata"),
                namespace=hit["_source"].get("namespace"),
            )
            for hit in response["hits"]["hits"]
        ]

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        include_metadata: bool = True,
        field_name: str = "vector",
```

(continues on next page)

(continued from previous page)

```
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Performs a nearest neighbor search on OpenSearch to find vectors similar␣
↪to the provided vector list.

        Results can be limited using the count parameter and optionally filtered by a␣
↪namespace.

        Returns:
            A list of BaseVectorStoreDriver.Entry objects, each encapsulating the␣
↪retrieved vector, its similarity score, metadata, and namespace.
        """
        count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
        # Base k-NN query
        query_body = {"size": count, "query": {"knn": {field_name: {"vector": vector,
↪"k": count}}}}

        if namespace:
            query_body["query"] = {
                "bool": {
                    "must": [
                        {"match": {"namespace": namespace}},
                        {"knn": {field_name: {"vector": vector, "k": count}}},
                    ],
                },
            }

        response = self.client.search(index=self.index_name, body=query_body)

        return [
            BaseVectorStoreDriver.Entry(
                id=hit["_id"],
                namespace=hit["_source"].get("namespace") if namespace else None,
                score=hit["_score"],
                vector=hit["_source"].get("vector") if include_vectors else None,
                meta=hit["_source"].get("metadata") if include_metadata else None,
            )
            for hit in response["hits"]["hits"]
        ]

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `host = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `http_auth = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `index_name = field(kw_only=True, metadata={'serializable': True})` class-attribute

instance-attribute

- `port = field(default=443, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute
- `use_ssl = field(default=True, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute
- `verify_certs = field(default=True, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute

client()

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
@lazy_property()
def client(self) -> OpenSearch:
    opensearchpy = import_optional_dependency("opensearchpy")

    return opensearchpy.OpenSearch(
        hosts=[{"host": self.host, "port": self.port}],
        http_auth=self.http_auth,
        use_ssl=self.use_ssl,
        verify_certs=self.verify_certs,
        connection_class=opensearchpy.RequestsHttpConnection,
    )
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

load_entries(*, namespace=None)

Retrieves all vector entries from OpenSearch that match the optional namespace.

Returns

Type	Description
<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	A list of BaseVectorStoreDriver.Entry objects.

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Retrieves all vector entries from OpenSearch that match the optional namespace.

    Returns:
```

(continues on next page)

(continued from previous page)

```
        A list of BaseVectorStoreDriver.Entry objects.
    """
    query_body = {"size": 10000, "query": {"match_all": {}}}

    if namespace:
        query_body["query"] = {"match": {"namespace": namespace}}

    response = self.client.search(index=self.index_name, body=query_body)

    return [
        BaseVectorStoreDriver.Entry(
            id=hit["_id"],
            vector=hit["_source"].get("vector"),
            meta=hit["_source"].get("metadata"),
            namespace=hit["_source"].get("namespace"),
        )
        for hit in response["hits"]["hits"]
    ]
```

load_entry(vector_id, *, namespace=None)

Retrieves a specific vector entry from OpenSearch based on its identifier and optional namespace.

Returns

Type	Description
<a href="vector#griptape.drivers.vector.BaseVectorStoreDriver.Entry" title="Entry (griptape.drivers.vector.BaseVectorStoreDriver.Entry)"><code class="language-python">C	If the entry is found, it returns an instance of BaseVectorStoreDriver.Entry; otherwise, None is returned.

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    """Retrieves a specific vector entry from OpenSearch based on its identifier and␣
↪optional namespace.

    Returns:
        If the entry is found, it returns an instance of BaseVectorStoreDriver.Entry;␣
↪otherwise, None is returned.
    """
    try:
        query = {"bool": {"must": [{"term": {"_id": vector_id}}]}}

        if namespace:
```

(continues on next page)

(continued from previous page)

```
            query["bool"]["must"].append({"term": {"namespace": namespace}})

        response = self.client.search(index=self.index_name, body={"query": query,
↪"size": 1})

        if response["hits"]["total"]["value"] > 0:
            vector_data = response["hits"]["hits"][0]["_source"]
            return BaseVectorStoreDriver.Entry(
                id=vector_id,
                meta=vector_data.get("metadata"),
                vector=vector_data.get("vector"),
                namespace=vector_data.get("namespace"),
            )
        return None
    except Exception as e:
        logging.exception("Error while loading entry: %s", e)
        return None
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, include_metadata=True, field_name='vector', **kwargs)

Performs a nearest neighbor search on OpenSearch to find vectors similar to the provided vector list.

Results can be limited using the count parameter and optionally filtered by a namespace.

Returns

Type	Description
<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	A list of BaseVectorStoreDriver.Entry objects, each encapsulating the retrieved vector, its similarity score, metadata, and namespace.

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    include_metadata: bool = True,
    field_name: str = "vector",
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    """Performs a nearest neighbor search on OpenSearch to find vectors similar to␣
↪the provided vector list.

    Results can be limited using the count parameter and optionally filtered by a␣
↪namespace.
```

(continues on next page)

(continued from previous page)

```
    Returns:
        A list of BaseVectorStoreDriver.Entry objects, each encapsulating the␣
↪retrieved vector, its similarity score, metadata, and namespace.
    """
    count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
    # Base k-NN query
    query_body = {"size": count, "query": {"knn": {field_name: {"vector": vector, "k
↪": count}}}}

    if namespace:
        query_body["query"] = {
            "bool": {
                "must": [
                    {"match": {"namespace": namespace}},
                    {"knn": {field_name: {"vector": vector, "k": count}}},
                ],
            },
        }

    response = self.client.search(index=self.index_name, body=query_body)

    return [
        BaseVectorStoreDriver.Entry(
            id=hit["_id"],
            namespace=hit["_source"].get("namespace") if namespace else None,
            score=hit["_score"],
            vector=hit["_source"].get("vector") if include_vectors else None,
            meta=hit["_source"].get("metadata") if include_metadata else None,
        )
        for hit in response["hits"]["hits"]
    ]
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    """Inserts or updates a vector in OpenSearch.

    If a vector with the given vector ID already exists, it is updated; otherwise, a␣
↪new vector is inserted.
    Metadata associated with the vector can also be provided.
    """
```

(continues on next page)

(continued from previous page)

```
    vector_id = vector_id or utils.str_to_hash(str(vector))
    doc = {"vector": vector, "namespace": namespace, "metadata": meta}
    doc.update(kwargs)
    response = self.client.index(index=self.index_name, id=vector_id, body=doc)

    return response["_id"]
```

OpenTelemetryObservabilityDriver

Bases: *BaseObservabilityDriver* “)

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
@define
class OpenTelemetryObservabilityDriver(BaseObservabilityDriver):
    service_name: str = field(default="griptape", kw_only=True)
    span_processor: SpanProcessor = field(kw_only=True)
    service_version: Optional[str] = field(default=None, kw_only=True)
    deployment_env: Optional[str] = field(default=None, kw_only=True)
    trace_provider: TracerProvider = field(
        default=Factory(
            lambda self: self._trace_provider_factory(),
            takes_self=True,
        ),
        kw_only=True,
    )
    _tracer: Tracer = field(init=False)
    _root_span_context_manager: Any = None

    def _trace_provider_factory(self) -> TracerProvider:
        opentelemetry_trace = import_optional_dependency("opentelemetry.sdk.trace")

        attributes = {"service.name": self.service_name}
        if self.service_version is not None:
            attributes["service.version"] = self.service_version
        if self.deployment_env is not None:
            attributes["deployment.environment"] = self.deployment_env
        return opentelemetry_trace.TracerProvider(
            resource=import_optional_dependency("opentelemetry.sdk.resources").
↪Resource(attributes=attributes)
        )  # pyright: ignore[reportArgumentType]

    def __attrs_post_init__(self) -> None:
        opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
        self.trace_provider.add_span_processor(self.span_processor)
        self._tracer = opentelemetry_trace.get_tracer(self.service_name, tracer_
↪provider=self.trace_provider)

    def __enter__(self) -> None:
        opentelemetry_instrumentation_threading = import_optional_dependency(
↪"opentelemetry.instrumentation.threading")
```

(continues on next page)

(continued from previous page)

```
        opentelemetry_instrumentation_threading.ThreadingInstrumentor().instrument()
        self._root_span_context_manager = self._tracer.start_as_current_span("main")
↪# pyright: ignore[reportCallIssue]
        self._root_span_context_manager.__enter__()

    def __exit__(
        self,
        exc_type: Optional[type[BaseException]],
        exc_value: Optional[BaseException],
        exc_traceback: Optional[TracebackType],
    ) -> bool:
        opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
        opentelemetry_instrumentation_threading = import_optional_dependency(
↪"opentelemetry.instrumentation.threading")
        root_span = opentelemetry_trace.get_current_span()
        if exc_value:
            root_span = opentelemetry_trace.get_current_span()
            root_span.set_status(opentelemetry_trace.Status(opentelemetry_trace.
↪StatusCode.ERROR))
            root_span.record_exception(exc_value)
        else:
            root_span.set_status(opentelemetry_trace.Status(opentelemetry_trace.
↪StatusCode.OK))
        if self._root_span_context_manager:
            self._root_span_context_manager.__exit__(exc_type, exc_value, exc_
↪traceback)
            self._root_span_context_manager = None
        self.trace_provider.force_flush()
        opentelemetry_instrumentation_threading.ThreadingInstrumentor().uninstrument()
        return False

    def observe(self, call: Observable.Call) -> Any:
        open_telemetry_trace = import_optional_dependency("opentelemetry.trace")
        func = call.func
        instance = call.instance
        tags = call.tags

        class_name = f"{instance.__class__.__name__}." if instance else ""
        span_name = f"{class_name}{func.__name__}()"
        with self._tracer.start_as_current_span(span_name) as span:  # pyright:␣
↪ignore[reportCallIssue]
            if tags is not None:
                span.set_attribute("tags", tags)

            try:
                result = call()
                span.set_status(open_telemetry_trace.Status(open_telemetry_trace.
↪StatusCode.OK))
                return result
            except Exception as e:
                span.set_status(open_telemetry_trace.Status(open_telemetry_trace.
↪StatusCode.ERROR))
```

(continues on next page)

(continued from previous page)

```
                span.record_exception(e)
                raise e

    def get_span_id(self) -> Optional[str]:
        opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
        span = opentelemetry_trace.get_current_span()
        if span is opentelemetry_trace.INVALID_SPAN:
            return None
        return opentelemetry_trace.format_span_id(span.get_span_context().span_id)
```

- `_root_span_context_manager = None` class-attribute instance-attribute
- `_tracer = field(init=False)` class-attribute instance-attribute
- `deployment_env = field(default=None, kw_only=True)` class-attribute instance-attribute
- `service_name = field(default='griptape', kw_only=True)` class-attribute instance-attribute
- `service_version = field(default=None, kw_only=True)` class-attribute instance-attribute
- `span_processor = field(kw_only=True)` class-attribute instance-attribute
- `trace_provider = field(default=Factory(lambda self: self._trace_provider_factory(), takes_self=True), kw_only=True)` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
def __attrs_post_init__(self) -> None:
    opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
    self.trace_provider.add_span_processor(self.span_processor)
    self._tracer = opentelemetry_trace.get_tracer(self.service_name, tracer_
↪provider=self.trace_provider)
```

enter()

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
def __enter__(self) -> None:
    opentelemetry_instrumentation_threading = import_optional_dependency(
↪"opentelemetry.instrumentation.threading")

    opentelemetry_instrumentation_threading.ThreadingInstrumentor().instrument()
    self._root_span_context_manager = self._tracer.start_as_current_span("main")  #␣
↪pyright: ignore[reportCallIssue]
    self._root_span_context_manager.__enter__()
```

exit(exc_type, exc_value, exc_traceback)

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
def __exit__(
    self,
    exc_type: Optional[type[BaseException]],
```

(continues on next page)

(continued from previous page)

```
    exc_value: Optional[BaseException],
    exc_traceback: Optional[TracebackType],
) -> bool:
    opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
    opentelemetry_instrumentation_threading = import_optional_dependency(
↪"opentelemetry.instrumentation.threading")
    root_span = opentelemetry_trace.get_current_span()
    if exc_value:
        root_span = opentelemetry_trace.get_current_span()
        root_span.set_status(opentelemetry_trace.Status(opentelemetry_trace.
↪StatusCode.ERROR))
        root_span.record_exception(exc_value)
    else:
        root_span.set_status(opentelemetry_trace.Status(opentelemetry_trace.
↪StatusCode.OK))
    if self._root_span_context_manager:
        self._root_span_context_manager.__exit__(exc_type, exc_value, exc_traceback)
        self._root_span_context_manager = None
    self.trace_provider.force_flush()
    opentelemetry_instrumentation_threading.ThreadingInstrumentor().uninstrument()
    return False
```

_trace_provider_factory()

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
def _trace_provider_factory(self) -> TracerProvider:
    opentelemetry_trace = import_optional_dependency("opentelemetry.sdk.trace")

    attributes = {"service.name": self.service_name}
    if self.service_version is not None:
        attributes["service.version"] = self.service_version
    if self.deployment_env is not None:
        attributes["deployment.environment"] = self.deployment_env
    return opentelemetry_trace.TracerProvider(
        resource=import_optional_dependency("opentelemetry.sdk.resources").
↪Resource(attributes=attributes)
    )  # pyright: ignore[reportArgumentType]
```

get_span_id()

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
def get_span_id(self) -> Optional[str]:
    opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
    span = opentelemetry_trace.get_current_span()
    if span is opentelemetry_trace.INVALID_SPAN:
        return None
    return opentelemetry_trace.format_span_id(span.get_span_context().span_id)
```

observe(call)

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
def observe(self, call: Observable.Call) -> Any:
    open_telemetry_trace = import_optional_dependency("opentelemetry.trace")
    func = call.func
    instance = call.instance
    tags = call.tags

    class_name = f"{instance.__class__.__name__}." if instance else ""
    span_name = f"{class_name}{func.__name__}()"
    with self._tracer.start_as_current_span(span_name) as span:  # pyright:␣
↪ignore[reportCallIssue]
        if tags is not None:
            span.set_attribute("tags", tags)

        try:
            result = call()
            span.set_status(open_telemetry_trace.Status(open_telemetry_trace.
↪StatusCode.OK))
            return result
        except Exception as e:
            span.set_status(open_telemetry_trace.Status(open_telemetry_trace.
↪StatusCode.ERROR))
            span.record_exception(e)
            raise e
```

PerplexityPromptDriver

Bases: *AzureOpenAiChatPromptDriver* “)

Source Code in griptape/drivers/prompt/perplexity_prompt_driver.py

```
@define
class PerplexityPromptDriver(OpenAiChatPromptDriver):
    base_url: str = field(default="https://api.perplexity.ai", kw_only=True, metadata=
↪{"serializable": True})
    structured_output_strategy: str = field(default="native", kw_only=True, metadata={
↪"serializable": True})

    @override
    def _to_message(self, result: ChatCompletion) -> Message:
        message = super()._to_message(result)

        message.content[0].artifact.meta["citations"] = getattr(result, "citations",␣
↪[])

        return message

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        params = super()._base_params(prompt_stack)

        if "stop" in params:
```

(continues on next page)

(continued from previous page)

```
            del params["stop"]

        return params
```

- `base_url = field(default='https://api.perplexity.ai', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='native', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/perplexity_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    params = super()._base_params(prompt_stack)

    if "stop" in params:
        del params["stop"]

    return params
```

_to_message(result)

Source Code in griptape/drivers/prompt/perplexity_prompt_driver.py

```
@override
def _to_message(self, result: ChatCompletion) -> Message:
    message = super()._to_message(result)

    message.content[0].artifact.meta["citations"] = getattr(result, "citations", [])

    return message
```

PerplexityWebSearchDriver

Bases: *BaseWebSearchDriver* “)

Source Code in griptape/drivers/web_search/perplexity_web_search_driver.py

```
@define
class PerplexityWebSearchDriver(BaseWebSearchDriver):
    model: str = field(default="sonar-pro", kw_only=True, metadata={"serializable":␣
↪True})
    api_key: Optional[str] = field(kw_only=True, default=None)
    _prompt_driver: Optional[PerplexityPromptDriver] = field(default=None, alias=
↪"prompt_driver")

    @lazy_property()
    def prompt_driver(self) -> PerplexityPromptDriver:
        if self.api_key is None:
            raise ValueError("api_key is required")
        return PerplexityPromptDriver(model=self.model, api_key=self.api_key)
```

(continues on next page)

(continued from previous page)

```
    def search(self, query: str, **kwargs) -> ListArtifact:
        message = self.prompt_driver.run(PromptStack.from_
↪artifact(TextArtifact(query)))

        return ListArtifact([message.to_artifact()])
```

- `_prompt_driver = field(default=None, alias='prompt_driver')` class-attribute instance-attribute
- `api_key = field(kw_only=True, default=None)` class-attribute instance-attribute
- `model = field(default='sonar-pro', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

prompt_driver()

Source Code in griptape/drivers/web_search/perplexity_web_search_driver.py

```
@lazy_property()
def prompt_driver(self) -> PerplexityPromptDriver:
    if self.api_key is None:
        raise ValueError("api_key is required")
    return PerplexityPromptDriver(model=self.model, api_key=self.api_key)
```

search(query, **kwargs)

Source Code in griptape/drivers/web_search/perplexity_web_search_driver.py

```
def search(self, query: str, **kwargs) -> ListArtifact:
    message = self.prompt_driver.run(PromptStack.from_artifact(TextArtifact(query)))

    return ListArtifact([message.to_artifact()])
```

PgAiKnowledgeBaseVectorStoreDriver

Bases: *BaseVectorStoreDriver* “)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
@define
class PgAiKnowledgeBaseVectorStoreDriver(BaseVectorStoreDriver):
    connection_string: str = field(kw_only=True, metadata={"serializable": True})
    knowledge_base_name: str = field(kw_only=True, metadata={"serializable": True})
    embedding_driver: BaseEmbeddingDriver = field(
        default=Factory(lambda: DummyEmbeddingDriver()),
        metadata={"serializable": True},
        kw_only=True,
        init=False,
    )
    _engine: sqlalchemy.Engine = field(default=None, kw_only=True, alias="engine",␣
↪metadata={"serializable": False})
```

(continues on next page)

(continued from previous page)

```
    @lazy_property()
    def engine(self) -> sqlalchemy.Engine:
        return import_optional_dependency("sqlalchemy").create_engine(self.connection_
↪string)

    def query(
        self,
        query: str | TextArtifact | ImageArtifact,
        *,
        count: Optional[int] = BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        if isinstance(query, ImageArtifact):
            raise ValueError(f"{self.__class__.__name__} does not support querying␣
↪with Image Artifacts.")

        sqlalchemy = import_optional_dependency("sqlalchemy")

        with sqlalchemy.orm.Session(self.engine) as session:
            rows = session.query(sqlalchemy.func.aidb.retrieve_text(self.knowledge_
↪base_name, query, count)).all()

        entries = []
        for (row,) in rows:
            # Remove the first and last parentheses from the row and list by commas
            # Example: (foo,bar) -> [foo, bar]
            row_list = "".join(row.replace("(", "", 1).rsplit(")", 1)).split(",")
            entries.append(
                BaseVectorStoreDriver.Entry(
                    id=row_list[0],
                    score=float(row_list[2]),
                    meta={"artifact": TextArtifact(row_list[1]).to_json()},
                )
            )

        return entries

    def upsert_vector(
        self,
        vector: list[float],
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        raise NotImplementedError(f"{self.__class__.__name__} does not support vector␣
↪upsert.")

    def upsert_text_artifact(
        self,
        artifact: TextArtifact,
        namespace: Optional[str] = None,
```

(continues on next page)

(continued from previous page)

```
        meta: Optional[dict] = None,
        vector_id: Optional[str] = None,
        **kwargs,
    ) -> str:
        raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪artifact upsert.")

    def upsert_text(
        self,
        string: str,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪upsert.")

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪BaseVectorStoreDriver.Entry:
        raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")

    def load_artifacts(self, *, namespace: Optional[str] = None) -> ListArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪Artifact loading.")

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")
```

- `_engine = field(default=None, kw_only=True, alias='engine', metadata={'serializable': False})` class-attribute instance-attribute
- `connection_string = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `embedding_driver = field(default=Factory(lambda: DummyEmbeddingDriver()), metadata={'serializable': True}, kw_only=True, init=False)` class-attribute instance-attribute
- `knowledge_base_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

delete_vector(vector_id)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

engine()

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
@lazy_property()
def engine(self) -> sqlalchemy.Engine:
    return import_optional_dependency("sqlalchemy").create_engine(self.connection_
↪string)
```

load_artifacts(*, namespace=None)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def load_artifacts(self, *, namespace: Optional[str] = None) -> ListArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support Artifact␣
↪loading.")
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")
```

load_entry(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪BaseVectorStoreDriver.Entry:
    raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")
```

query(query, *, count=BaseVectorStoreDriver.DEFAULT_QUERY_COUNT, **kwargs)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def query(
    self,
    query: str | TextArtifact | ImageArtifact,
    *,
    count: Optional[int] = BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    if isinstance(query, ImageArtifact):
        raise ValueError(f"{self.__class__.__name__} does not support querying with␣
↪Image Artifacts.")

    sqlalchemy = import_optional_dependency("sqlalchemy")

    with sqlalchemy.orm.Session(self.engine) as session:
        rows = session.query(sqlalchemy.func.aidb.retrieve_text(self.knowledge_base_
```

(continues on next page)

(continued from previous page)

```
↪name, query, count)).all()

    entries = []
    for (row,) in rows:
        # Remove the first and last parentheses from the row and list by commas
        # Example: (foo,bar) -> [foo, bar]
        row_list = "".join(row.replace("(", "", 1).rsplit(")", 1)).split(",")
        entries.append(
            BaseVectorStoreDriver.Entry(
                id=row_list[0],
                score=float(row_list[2]),
                meta={"artifact": TextArtifact(row_list[1]).to_json()},
            )
        )

    return entries
```

upsert_text(string, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def upsert_text(
    self,
    string: str,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪upsert.")
```

upsert_text_artifact(artifact, namespace=None, meta=None, vector_id=None, **kwargs)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def upsert_text_artifact(
    self,
    artifact: TextArtifact,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    vector_id: Optional[str] = None,
    **kwargs,
) -> str:
    raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪artifact upsert.")
```

upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    raise NotImplementedError(f"{self.__class__.__name__} does not support vector␣
↪upsert.")
```

PgVectorVectorStoreDriver

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.PgVectorVectorStoreDriver.connection_string" title="connection_string = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.PgVectorVectorStoreDriver.connection_string)"><code class="language-python">	*Optional[str]*	An optional string describing the target Postgres database instance.
<a href="./#griptape.drivers.PgVectorVectorStoreDriver.create_engine_params" title="create_engine_params = field(factory=dict, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.PgVectorVectorStoreDriver.create_engine_params)"><code class="language-python">	*dict*	Additional configuration params passed when creating the database connection.

continues on next page

Table 110 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.PgVectorVectorStoreDriver.engine" title="engine() (griptape.drivers.PgVectorVectorStoreDriver.engine)"><code class="language-python">	*Engine*	An optional sqlalchemy Postgres engine to use.
<a href="./#griptape.drivers.PgVectorVectorStoreDriver.table_name" title="table_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.PgVectorVectorStoreDriver.table_name)"><code class="language-python">	*str*	Optionally specify the name of the table to used to store vectors.

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
@define
class PgVectorVectorStoreDriver(BaseVectorStoreDriver):
    """A vector store driver to Postgres using the PGVector extension.

    Attributes:
        connection_string: An optional string describing the target Postgres database instance.
        create_engine_params: Additional configuration params passed when creating the database connection.
        engine: An optional sqlalchemy Postgres engine to use.
        table_name: Optionally specify the name of the table to used to store vectors.
    """

    connection_string: Optional[str] = field(default=None, kw_only=True, metadata={"serializable": True})
    create_engine_params: dict = field(factory=dict, kw_only=True, metadata={"serializable": True})
    table_name: str = field(kw_only=True, metadata={"serializable": True})
    _model: Any = field(default=Factory(lambda self: self.default_vector_model(), takes_self=True))
    _engine: Optional[sqlalchemy.Engine] = field(
```

(continues on next page)

(continued from previous page)

```
        default=None, kw_only=True, alias="engine", metadata={"serializable": False}
    )

    @connection_string.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_connection_string(self, _: Attribute, connection_string:␣
↪Optional[str]) -> None:
        # If an engine is provided, the connection string is not used.
        if self._engine is not None:
            return

        # If an engine is not provided, a connection string is required.
        if connection_string is None:
            raise ValueError("An engine or connection string is required")

        if not connection_string.startswith("postgresql://"):
            raise ValueError("The connection string must describe a Postgres database␣
↪connection")

    @lazy_property()
    def engine(self) -> sqlalchemy.Engine:
        return import_optional_dependency("sqlalchemy").create_engine(
            self.connection_string, **self.create_engine_params
        )

    def setup(
        self,
        *,
        create_schema: bool = True,
        install_uuid_extension: bool = True,
        install_vector_extension: bool = True,
    ) -> None:
        """Provides a mechanism to initialize the database schema and extensions."""
        sqlalchemy_sql = import_optional_dependency("sqlalchemy.sql")

        if install_uuid_extension:
            with self.engine.begin() as conn:
                conn.execute(sqlalchemy_sql.text(CREATE EXTENSION IF NOT EXISTS "uuid-
↪ossp";))

        if install_vector_extension:
            with self.engine.begin() as conn:
                conn.execute(sqlalchemy_sql.text(CREATE EXTENSION IF NOT EXISTS
↪"vector";))

        if create_schema:
            self._model.metadata.create_all(self.engine)

    def upsert_vector(
        self,
        vector: list[float],
        *,
```

(continues on next page)

(continued from previous page)

```
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        """Inserts or updates a vector in the collection."""
        sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

        with sqlalchemy_orm.Session(self.engine) as session:
            obj = self._model(id=vector_id, vector=vector, namespace=namespace,␣
↪meta=meta, **kwargs)

            obj = session.merge(obj)
            session.commit()

            return str(obj.id)

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪BaseVectorStoreDriver.Entry:
        """Retrieves a specific vector entry from the collection based on its␣
↪identifier and optional namespace."""
        sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

        with sqlalchemy_orm.Session(self.engine) as session:
            result = session.get(self._model, vector_id)

            return BaseVectorStoreDriver.Entry(
                id=result.id,
                vector=result.vector,
                namespace=result.namespace,
                meta=result.meta,
            )

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Retrieves all vector entries from the collection, optionally filtering to␣
↪only those that match the provided namespace."""
        sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

        with sqlalchemy_orm.Session(self.engine) as session:
            query = session.query(self._model)
            if namespace:
                query = query.filter_by(namespace=namespace)

            results = query.all()

            return [
                BaseVectorStoreDriver.Entry(
                    id=str(result.id),
                    vector=result.vector,
                    namespace=result.namespace,
                    meta=result.meta,
```

(continues on next page)

(continued from previous page)

```
                )
                for result in results
            ]

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        distance_metric: str = "cosine_distance",
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Performs a search on the collection to find vectors similar to the provided input vector, optionally filtering to only those that match the provided namespace."""
        sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

        count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT

        distance_metrics = {
            "cosine_distance": self._model.vector.cosine_distance,
            "l2_distance": self._model.vector.l2_distance,
            "inner_product": self._model.vector.max_inner_product,
        }

        if distance_metric not in distance_metrics:
            raise ValueError("Invalid distance metric provided")

        op = distance_metrics[distance_metric]

        with sqlalchemy_orm.Session(self.engine) as session:
            # The query should return both the vector and the distance metric score.
            query_result = session.query(self._model, op(vector).label("score")).order_by(op(vector))  # pyright: ignore[reportOptionalCall]

            filter_kwargs: Optional[OrderedDict] = None

            if namespace is not None:
                filter_kwargs = OrderedDict(namespace=namespace)

            if "filter" in kwargs and isinstance(kwargs["filter"], dict):
                filter_kwargs = filter_kwargs or OrderedDict()
                filter_kwargs.update(kwargs["filter"])

            if filter_kwargs is not None:
                query_result = query_result.filter_by(**filter_kwargs)

            results = query_result.limit(count).all()

            return [
```

(continues on next page)

(continued from previous page)

```
                BaseVectorStoreDriver.Entry(
                    id=str(result[0].id),
                    vector=result[0].vector if include_vectors else None,
                    score=result[1],
                    meta=result[0].meta,
                    namespace=result[0].namespace,
                )
                for result in results
            ]

    def default_vector_model(self) -> Any:
        pgvector_sqlalchemy = import_optional_dependency("pgvector.sqlalchemy")
        sqlalchemy = import_optional_dependency("sqlalchemy")
        sqlalchemy_dialects_postgresql = import_optional_dependency("sqlalchemy.
↪dialects.postgresql")
        sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

        @dataclass
        class VectorModel(sqlalchemy_orm.declarative_base()):
            __tablename__ = self.table_name

            id = sqlalchemy.Column(
                sqlalchemy_dialects_postgresql.UUID(as_uuid=True),
                primary_key=True,
                default=uuid.uuid4,
                unique=True,
                nullable=False,
            )
            vector = sqlalchemy.Column(pgvector_sqlalchemy.Vector())
            namespace = sqlalchemy.Column(sqlalchemy.String)
            meta = sqlalchemy.Column(sqlalchemy.JSON)

        return VectorModel

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")
```

- `_engine = field(default=None, kw_only=True, alias='engine', metadata={'serializable': False})` class-attribute instance-attribute
- `_model = field(default=Factory(lambda self: self.default_vector_model(), takes_self=True))` class-attribute instance-attribute
- `connection_string = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `create_engine_params = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `table_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

default_vector_model()

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def default_vector_model(self) -> Any:
    pgvector_sqlalchemy = import_optional_dependency("pgvector.sqlalchemy")
    sqlalchemy = import_optional_dependency("sqlalchemy")
    sqlalchemy_dialects_postgresql = import_optional_dependency("sqlalchemy.dialects.
↪postgresql")
    sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

    @dataclass
    class VectorModel(sqlalchemy_orm.declarative_base()):
        __tablename__ = self.table_name

        id = sqlalchemy.Column(
            sqlalchemy_dialects_postgresql.UUID(as_uuid=True),
            primary_key=True,
            default=uuid.uuid4,
            unique=True,
            nullable=False,
        )
        vector = sqlalchemy.Column(pgvector_sqlalchemy.Vector())
        namespace = sqlalchemy.Column(sqlalchemy.String)
        meta = sqlalchemy.Column(sqlalchemy.JSON)

    return VectorModel
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

engine()

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
@lazy_property()
def engine(self) -> sqlalchemy.Engine:
    return import_optional_dependency("sqlalchemy").create_engine(
        self.connection_string, **self.create_engine_params
    )
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Retrieves all vector entries from the collection, optionally filtering to only␣
↪those that match the provided namespace."""
    sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")
```

(continues on next page)

(continued from previous page)

```
    with sqlalchemy_orm.Session(self.engine) as session:
        query = session.query(self._model)
        if namespace:
            query = query.filter_by(namespace=namespace)

        results = query.all()

        return [
            BaseVectorStoreDriver.Entry(
                id=str(result.id),
                vector=result.vector,
                namespace=result.namespace,
                meta=result.meta,
            )
            for result in results
        ]
```

load_entry(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪BaseVectorStoreDriver.Entry:
    """Retrieves a specific vector entry from the collection based on its identifier␣
↪and optional namespace."""
    sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

    with sqlalchemy_orm.Session(self.engine) as session:
        result = session.get(self._model, vector_id)

        return BaseVectorStoreDriver.Entry(
            id=result.id,
            vector=result.vector,
            namespace=result.namespace,
            meta=result.meta,
        )
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, distance_metric='cosine_distance', **kwargs)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    distance_metric: str = "cosine_distance",
    **kwargs,
```

(continues on next page)

(continued from previous page)

```
) -> list[BaseVectorStoreDriver.Entry]:
    """Performs a search on the collection to find vectors similar to the provided input vector, optionally filtering to only those that match the provided namespace."""
    sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

    count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT

    distance_metrics = {
        "cosine_distance": self._model.vector.cosine_distance,
        "l2_distance": self._model.vector.l2_distance,
        "inner_product": self._model.vector.max_inner_product,
    }

    if distance_metric not in distance_metrics:
        raise ValueError("Invalid distance metric provided")

    op = distance_metrics[distance_metric]

    with sqlalchemy_orm.Session(self.engine) as session:
        # The query should return both the vector and the distance metric score.
        query_result = session.query(self._model, op(vector).label("score")).order_by(op(vector))  # pyright: ignore[reportOptionalCall]

        filter_kwargs: Optional[OrderedDict] = None

        if namespace is not None:
            filter_kwargs = OrderedDict(namespace=namespace)

        if "filter" in kwargs and isinstance(kwargs["filter"], dict):
            filter_kwargs = filter_kwargs or OrderedDict()
            filter_kwargs.update(kwargs["filter"])

        if filter_kwargs is not None:
            query_result = query_result.filter_by(**filter_kwargs)

        results = query_result.limit(count).all()

        return [
            BaseVectorStoreDriver.Entry(
                id=str(result[0].id),
                vector=result[0].vector if include_vectors else None,
                score=result[1],
                meta=result[0].meta,
                namespace=result[0].namespace,
            )
            for result in results
        ]
```

setup(*, create_schema=True, install_uuid_extension=True, install_vector_extension=True)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def setup(
    self,
    *,
    create_schema: bool = True,
    install_uuid_extension: bool = True,
    install_vector_extension: bool = True,
) -> None:
    """Provides a mechanism to initialize the database schema and extensions."""
    sqlalchemy_sql = import_optional_dependency("sqlalchemy.sql")

    if install_uuid_extension:
        with self.engine.begin() as conn:
            conn.execute(sqlalchemy_sql.text(CREATE EXTENSION IF NOT EXISTS "uuid-ossp
↪";))

    if install_vector_extension:
        with self.engine.begin() as conn:
            conn.execute(sqlalchemy_sql.text(CREATE EXTENSION IF NOT EXISTS "vector";
↪))

    if create_schema:
        self._model.metadata.create_all(self.engine)
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    """Inserts or updates a vector in the collection."""
    sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

    with sqlalchemy_orm.Session(self.engine) as session:
        obj = self._model(id=vector_id, vector=vector, namespace=namespace, meta=meta,
↪ **kwargs)

        obj = session.merge(obj)
        session.commit()

        return str(obj.id)
```

validate*connection_string(*, connection_string)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
@connection_string.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_connection_string(self, _: Attribute, connection_string: Optional[str]) -
↪> None:
    # If an engine is provided, the connection string is not used.
    if self._engine is not None:
        return

    # If an engine is not provided, a connection string is required.
    if connection_string is None:
        raise ValueError("An engine or connection string is required")

    if not connection_string.startswith("postgresql://"):
        raise ValueError("The connection string must describe a Postgres database␣
↪connection")
```

PineconeVectorStoreDriver

Bases: *BaseVectorStoreDriver* “)

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
@define
class PineconeVectorStoreDriver(BaseVectorStoreDriver):
    api_key: str = field(kw_only=True, metadata={"serializable": True})
    index_name: str = field(kw_only=True, metadata={"serializable": True})
    environment: str = field(kw_only=True, metadata={"serializable": True})
    project_name: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    _client: Optional[pinecone.Pinecone] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )
    _index: Optional[pinecone.Index] = field(
        default=None, kw_only=True, alias="index", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> pinecone.Pinecone:
        return import_optional_dependency("pinecone").Pinecone(
            api_key=self.api_key,
            environment=self.environment,
            project_name=self.project_name,
        )

    @lazy_property()
    def index(self) -> pinecone.data.index.Index:
        return self.client.Index(self.index_name)

    def upsert_vector(
        self,
```

(continues on next page)

(continued from previous page)

```
        vector: list[float],
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        vector_id = vector_id or str_to_hash(str(vector))

        params: dict[str, Any] = {"namespace": namespace} | kwargs

        self.index.upsert(vectors=[(vector_id, vector, meta)], **params)  # pyright:␣
↪ignore[reportArgumentType]

        return vector_id

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        result = self.index.fetch(ids=[vector_id], namespace=namespace).to_dict()  #␣
↪pyright: ignore[reportAttributeAccessIssue]
        vectors = list(result["vectors"].values())

        if len(vectors) > 0:
            vector = vectors[0]

            return BaseVectorStoreDriver.Entry(
                id=vector["id"],
                meta=vector["metadata"],
                vector=vector["values"],
                namespace=result["namespace"],
            )
        return None

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        # This is a hacky way to query up to 10,000 values from Pinecone. Waiting on␣
↪an official API for fetching
        # all values from a namespace:
        # https://community.pinecone.io/t/is-there-a-way-to-query-all-the-vectors-and-
↪or-metadata-from-a-namespace/797/5

        results = self.index.query(
            vector=self.embedding_driver.embed("", vector_operation="query"),
            top_k=10000,
            include_metadata=True,
            namespace=namespace,
        )

        return [
            BaseVectorStoreDriver.Entry(
                id=r["id"],
                vector=r["values"],
                meta=r["metadata"],
```

(continues on next page)

(continued from previous page)

```
                namespace=results["namespace"],  # pyright: ignore[reportIndexIssue]
            )
            for r in results["matches"]  # pyright: ignore[reportIndexIssue]
        ]

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        include_metadata: bool = True,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        params = {
            "top_k": count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
            "namespace": namespace,
            "include_values": include_vectors,
            "include_metadata": include_metadata,
        } | kwargs

        results = self.index.query(vector=vector, **params)

        return [
            BaseVectorStoreDriver.Entry(
                id=r["id"],
                vector=r["values"],
                score=r["score"],
                meta=r["metadata"],
                namespace=results["namespace"],  # pyright: ignore[reportIndexIssue]
            )
            for r in results["matches"]  # pyright: ignore[reportIndexIssue]
        ]

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `_index = field(default=None, kw_only=True, alias='index', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `environment = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `index_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `project_name = field(default=None, kw_only=True, metadata={'serializable': True})`

class-attribute instance-attribute

client()

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
@lazy_property()
def client(self) -> pinecone.Pinecone:
    return import_optional_dependency("pinecone").Pinecone(
        api_key=self.api_key,
        environment=self.environment,
        project_name=self.project_name,
    )
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

index()

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
@lazy_property()
def index(self) -> pinecone.data.index.Index:
    return self.client.Index(self.index_name)
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    # This is a hacky way to query up to 10,000 values from Pinecone. Waiting on an␣
↪official API for fetching
    # all values from a namespace:
    # https://community.pinecone.io/t/is-there-a-way-to-query-all-the-vectors-and-or-
↪metadata-from-a-namespace/797/5

    results = self.index.query(
        vector=self.embedding_driver.embed("", vector_operation="query"),
        top_k=10000,
        include_metadata=True,
        namespace=namespace,
    )

    return [
        BaseVectorStoreDriver.Entry(
            id=r["id"],
            vector=r["values"],
            meta=r["metadata"],
```

(continues on next page)

(continued from previous page)

```
            namespace=results["namespace"],  # pyright: ignore[reportIndexIssue]
        )
        for r in results["matches"]  # pyright: ignore[reportIndexIssue]
    ]
```

load_entry(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    result = self.index.fetch(ids=[vector_id], namespace=namespace).to_dict()  #␣
↪pyright: ignore[reportAttributeAccessIssue]
    vectors = list(result["vectors"].values())

    if len(vectors) > 0:
        vector = vectors[0]

        return BaseVectorStoreDriver.Entry(
            id=vector["id"],
            meta=vector["metadata"],
            vector=vector["values"],
            namespace=result["namespace"],
        )
    return None
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, include_metadata=True, **kwargs)

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    include_metadata: bool = True,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    params = {
        "top_k": count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
        "namespace": namespace,
        "include_values": include_vectors,
        "include_metadata": include_metadata,
    } | kwargs

    results = self.index.query(vector=vector, **params)

    return [
        BaseVectorStoreDriver.Entry(
```

(continues on next page)

(continued from previous page)

```
            id=r["id"],
            vector=r["values"],
            score=r["score"],
            meta=r["metadata"],
            namespace=results["namespace"],  # pyright: ignore[reportIndexIssue]
        )
        for r in results["matches"]  # pyright: ignore[reportIndexIssue]
    ]
```

upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    vector_id = vector_id or str_to_hash(str(vector))

    params: dict[str, Any] = {"namespace": namespace} | kwargs

    self.index.upsert(vectors=[(vector_id, vector, meta)], **params)  # pyright:␣
↪ignore[reportArgumentType]

    return vector_id
```

ProxyWebScraperDriver

Bases: *BaseWebScraperDriver* “)

Source Code in griptape/drivers/web_scraper/proxy_web_scraper_driver.py

```
@define
class ProxyWebScraperDriver(BaseWebScraperDriver):
    proxies: dict = field(kw_only=True, metadata={"serializable": False})
    params: dict = field(default=Factory(dict), kw_only=True, metadata={"serializable
↪": True})

    def fetch_url(self, url: str) -> str:
        response = requests.get(url, proxies=self.proxies, **self.params)

        return response.text

    def extract_page(self, page: str) -> TextArtifact:
        return TextArtifact(page)
```

- `params = field(default=Factory(dict), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

- `proxies = field(kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute

extract_page(page)

Source Code in griptape/drivers/web_scraper/proxy_web_scraper_driver.py

```
def extract_page(self, page: str) -> TextArtifact:
    return TextArtifact(page)
```

fetch_url(url)

Source Code in griptape/drivers/web_scraper/proxy_web_scraper_driver.py

```
def fetch_url(self, url: str) -> str:
    response = requests.get(url, proxies=self.proxies, **self.params)

    return response.text
```

PusherEventListenerDriver

Bases: *BaseEventListenerDriver* “)

Source Code in griptape/drivers/event_listener/pusher_event_listener_driver.py

```
@define
class PusherEventListenerDriver(BaseEventListenerDriver):
    app_id: str = field(kw_only=True, metadata={"serializable": True})
    key: str = field(kw_only=True, metadata={"serializable": True})
    secret: str = field(kw_only=True, metadata={"serializable": False})
    cluster: str = field(kw_only=True, metadata={"serializable": True})
    channel: str = field(kw_only=True, metadata={"serializable": True})
    event_name: str = field(kw_only=True, metadata={"serializable": True})
    ssl: bool = field(default=True, kw_only=True, metadata={"serializable": True})
    _client: Optional[Pusher] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> Pusher:
        return import_optional_dependency("pusher").Pusher(
            app_id=self.app_id,
            key=self.key,
            secret=self.secret,
            cluster=self.cluster,
            ssl=self.ssl,
        )

    def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None:
        data = [
            {"channel": self.channel, "name": self.event_name, "data": event_payload}
            for event_payload in event_payload_batch
        ]
```

(continues on next page)

(continued from previous page)

```
        self.client.trigger_batch(data)

    def try_publish_event_payload(self, event_payload: dict) -> None:
        self.client.trigger(channels=self.channel, event_name=self.event_name,␣
↪data=event_payload)
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `app_id = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `channel = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `cluster = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `event_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `secret = field(kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `ssl = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/event_listener/pusher_event_listener_driver.py

```
@lazy_property()
def client(self) -> Pusher:
    return import_optional_dependency("pusher").Pusher(
        app_id=self.app_id,
        key=self.key,
        secret=self.secret,
        cluster=self.cluster,
        ssl=self.ssl,
    )
```

try_publish_event_payload(event_payload)

Source Code in griptape/drivers/event_listener/pusher_event_listener_driver.py

```
def try_publish_event_payload(self, event_payload: dict) -> None:
    self.client.trigger(channels=self.channel, event_name=self.event_name, data=event_
↪payload)
```

try_publish_event_payload_batch(event_payload_batch)

Source Code in griptape/drivers/event_listener/pusher_event_listener_driver.py

```
def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None:
    data = [
        {"channel": self.channel, "name": self.event_name, "data": event_payload}
        for event_payload in event_payload_batch
    ]

    self.client.trigger_batch(data)
```

QdrantVectorStoreDriver

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.QdrantVectorStoreDriver.location" title="location = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.QdrantVectorStoreDriver.location)"><code class="language-python">	*Optional[str]*	An optional location for the Qdrant client. If set to ':memory:', an in-memory client is used.
<a href="./#griptape.drivers.QdrantVectorStoreDriver.url" title="url = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.QdrantVectorStoreDriver.url)"><code class="language-python">	*Optional[str]*	An optional Qdrant API URL.

continues on next page

Table 111 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.QdrantVectorStoreDriver.host" title="host = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.QdrantVectorStoreDriver.host)"><code class="language-python">	*Optional[str]*	An optional Qdrant host.
<a href="./#griptape.drivers.QdrantVectorStoreDriver.path" title="path = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.QdrantVectorStoreDriver.path)"><code class="language-python">	*Optional[str]*	Persistence path for QdrantLocal. Default: None
<a href="./#griptape.drivers.QdrantVectorStoreDriver.port" title="port = field(default=6333, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.QdrantVectorStoreDriver.port)"><code class="language-python">	*int*	The port number for the Qdrant client. Defaults: 6333.

continues on next page

Table 111 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.QdrantVectorStoreDriver.grpc_port" title="grpc_port = field(default=6334, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.QdrantVectorStoreDriver.grpc_port)"><code class="language-python">	*int*	The gRPC port number for the Qdrant client. Defaults: 6334.
<a href="./#griptape.drivers.QdrantVectorStoreDriver.prefer_grpc" title="prefer_grpc = field(default=False, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.QdrantVectorStoreDriver.prefer_grpc)"><code class="language-python">	*bool*	A boolean indicating whether to prefer gRPC over HTTP. Defaults: False.
<a href="./#griptape.drivers.QdrantVectorStoreDriver.force_disable_check_same_thread' title="force_disable_check_same_thread = field(default=False, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.QdrantVectorStoreDriver.force_disable_check_same_thread) class="language-python">	*Optional[bool]*	For QdrantLocal, force disable check_same_thread. Default: False Only use this if you can guarantee that you can resolve the thread safety outside QdrantClient.

continues on next page

Table 111 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.QdrantVectorStoreDriver.timeout" title="timeout = field(default=5, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.QdrantVectorStoreDriver.timeout)"><code class="language-python">	*Optional[int]*	Timeout for REST and gRPC API requests. Default: 5 seconds for REST and unlimited for gRPC
<a href="./#griptape.drivers.QdrantVectorStoreDriver.api_key" title="api_key = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.QdrantVectorStoreDriver.api_key)"><c class="language-python">	*Optional[str]*	API key for authentication in Qdrant Cloud. Defaults: False
<a href="./#griptape.drivers.QdrantVectorStoreDriver.https" title="https = field(default=True, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.QdrantVectorStoreDriver.https)"><cod class="language-python">	*bool*	If true - use HTTPS(SSL) protocol. Default: True

continues on next page

Table 111 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.QdrantVectorStoreDriver.prefix" title="prefix = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.QdrantVectorStoreDriver.prefix)"><code class="language-python">	*Optional[str]*	Add prefix to the REST URL path. Example: service/v1 will result in Example: service/v1 will result in http://localhost:6333/service/v1/{qdrant-endpoint} for REST API. Defaults: None
<a href="./#griptape.drivers.QdrantVectorStoreDriver.distance" title="distance = field(default=DEFAULT_DISTANCE, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.QdrantVectorStoreDriver.distance)"><code class="language-python">	*str*	The distance metric to be used for the vectors. Defaults: 'COSINE'.
<a href="./#griptape.drivers.QdrantVectorStoreDriver.collection_name" title="collection_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.QdrantVectorStoreDriver.collection_name)"><code class="language-python">	*str*	The name of the Qdrant collection.

continues on next page

Table 111 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.QdrantVectorStoreDriver.vector_name" title="vector_name = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.QdrantVectorStoreDriver.vector_name)"><code class="language-python">	*Optional[str]*	An optional name for the vectors.
<a href="./#griptape.drivers.QdrantVectorStoreDriver.content_payload_key" title="content_payload_key = field(default=CONTENT_PAYLOAD_KEY, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.QdrantVectorStoreDriver.content_payload_key)"><code class="language-python">	*str*	The key for the content payload in the metadata. Defaults: 'data'.

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
@define
class QdrantVectorStoreDriver(BaseVectorStoreDriver):
    """Vector Store Driver for Qdrant.

    Attributes:
        location: An optional location for the Qdrant client. If set to :memory:, an␣
↪in-memory client is used.
        url: An optional Qdrant API URL.
        host: An optional Qdrant host.
        path: Persistence path for QdrantLocal. Default: None
        port: The port number for the Qdrant client. Defaults: 6333.
        grpc_port: The gRPC port number for the Qdrant client. Defaults: 6334.
        prefer_grpc: A boolean indicating whether to prefer gRPC over HTTP. Defaults:␣
↪False.
        force_disable_check_same_thread: For QdrantLocal, force disable check_same_
```

(continues on next page)

(continued from previous page)

```
↪thread. Default: False Only use this if you can guarantee that you can resolve the␣
↪thread safety outside QdrantClient.
        timeout: Timeout for REST and gRPC API requests. Default: 5 seconds for REST␣
↪and unlimited for gRPC
        api_key: API key for authentication in Qdrant Cloud. Defaults: False
        https: If true - use HTTPS(SSL) protocol. Default: True
        prefix: Add prefix to the REST URL path. Example: service/v1 will result in␣
↪Example: service/v1 will result in http://localhost:6333/service/v1/{qdrant-
↪endpoint} for REST API. Defaults: None
        distance: The distance metric to be used for the vectors. Defaults: COSINE.
        collection_name: The name of the Qdrant collection.
        vector_name: An optional name for the vectors.
        content_payload_key: The key for the content payload in the metadata.␣
↪Defaults: data.
    """

    location: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    url: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    host: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    path: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    port: int = field(default=6333, kw_only=True, metadata={"serializable": True})
    grpc_port: int = field(default=6334, kw_only=True, metadata={"serializable": True}
↪)
    prefer_grpc: bool = field(default=False, kw_only=True, metadata={"serializable":␣
↪True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    https: bool = field(default=True, kw_only=True, metadata={"serializable": True})
    prefix: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    force_disable_check_same_thread: Optional[bool] = field(
        default=False,
        kw_only=True,
        metadata={"serializable": True},
    )
    timeout: Optional[int] = field(default=5, kw_only=True, metadata={"serializable":␣
↪True})
    distance: str = field(default=DEFAULT_DISTANCE, kw_only=True, metadata={
↪"serializable": True})
    collection_name: str = field(kw_only=True, metadata={"serializable": True})
    vector_name: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    content_payload_key: str = field(default=CONTENT_PAYLOAD_KEY, kw_only=True,␣
↪metadata={"serializable": True})
    _client: Optional[QdrantClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )
```

(continues on next page)

(continued from previous page)

```
    @lazy_property()
    def client(self) -> QdrantClient:
        return import_optional_dependency("qdrant_client").QdrantClient(
            location=self.location,
            url=self.url,
            host=self.host,
            path=self.path,
            port=self.port,
            prefer_grpc=self.prefer_grpc,
            grpc_port=self.grpc_port,
            api_key=self.api_key,
            https=self.https,
            prefix=self.prefix,
            force_disable_check_same_thread=self.force_disable_check_same_thread,
            timeout=self.timeout,
        )

    def delete_vector(self, vector_id: str) -> None:
        """Delete a vector from the Qdrant collection based on its ID.

        Parameters:
            vector_id (str | id): ID of the vector to delete.
        """
        deletion_response = self.client.delete(
            collection_name=self.collection_name,
            points_selector=import_optional_dependency("qdrant_client.http.models").
↪PointIdsList(points=[vector_id]),
        )
        if deletion_response.status == import_optional_dependency("qdrant_client.http.
↪models").UpdateStatus.COMPLETED:
            logging.info("ID %s is successfully deleted", vector_id)

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Query the Qdrant collection based on a query vector.

        Parameters:
            vector (list[float]): Query vector.
            count (Optional[int]): Optional number of results to return.
            namespace (Optional[str]): Optional namespace of the vectors.
            include_vectors (bool): Whether to include vectors in the results.

        Returns:
            list[BaseVectorStoreDriver.Entry]: List of Entry objects.
        """
```

(continues on next page)

(continued from previous page)

```
        # Create a search request
        request = {"collection_name": self.collection_name, "query_vector": vector,
↪"limit": count}
        request = {k: v for k, v in request.items() if v is not None}
        results = self.client.search(**request)

        # Convert results to QueryResult objects
        return [
            BaseVectorStoreDriver.Entry(
                id=str(result.id),
                vector=result.vector if include_vectors else [],  # pyright:␣
↪ignore[reportArgumentType]
                score=result.score,
                meta={k: v for k, v in result.payload.items() if k not in ["_score",
↪"_tensor_facets"]},
            )
            for result in results
            if result.payload is not None
        ]

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        content: Optional[str] = None,
        **kwargs,
    ) -> str:
        """Upsert vectors into the Qdrant collection.

        Parameters:
            vector (list[float]): The vector to be upserted.
            vector_id (Optional[str]): Optional vector ID.
            namespace (Optional[str]): Optional namespace for the vector.
            meta (Optional[dict]): Optional dictionary containing metadata.
            content (Optional[str]): The text content to be included in the payload.

        Returns:
            str: The ID of the upserted vector.
        """
        if vector_id is None:
            vector_id = str(uuid.uuid5(uuid.NAMESPACE_DNS, str(vector)))

        if meta is None:
            meta = {}

        if content:
            meta[self.content_payload_key] = content

        points = import_optional_dependency("qdrant_client.http.models").Batch(
```

(continues on next page)

(continued from previous page)

```
            ids=[vector_id],
            vectors=[vector],
            payloads=[meta] if meta else None,
        )

        self.client.upsert(collection_name=self.collection_name, points=points)
        return vector_id

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        """Load a vector entry from the Qdrant collection based on its ID.

        Parameters:
            vector_id (str): ID of the vector to load.
            namespace (str, optional): Optional namespace of the vector.

        Returns:
            Optional[BaseVectorStoreDriver.Entry]: Vector entry if found, else None.
        """
        results = self.client.retrieve(collection_name=self.collection_name,␣
↪ids=[vector_id])
        if results:
            entry = results[0]
            if entry.payload is None:
                entry.payload = {}
            return BaseVectorStoreDriver.Entry(
                id=str(entry.id),
                vector=entry.vector if entry.vector is not None else [],  # pyright:␣
↪ignore[reportArgumentType]
                meta={k: v for k, v in entry.payload.items() if k not in ["_score", "_
↪tensor_facets"]},
            )
        return None

    def load_entries(self, *, namespace: Optional[str] = None, **kwargs) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Load vector entries from the Qdrant collection.

        Parameters:
            namespace: Optional namespace of the vectors.

        Returns:
            List of points.
        """
        results = self.client.retrieve(
            collection_name=self.collection_name,
            ids=kwargs.get("ids", []),
            with_payload=kwargs.get("with_payload", True),
            with_vectors=kwargs.get("with_vectors", True),
        )

        return [
```

(continues on next page)

(continued from previous page)

```
            BaseVectorStoreDriver.Entry(
                id=str(entry.id),
                vector=entry.vector if kwargs.get("with_vectors", True) else [],  #␣
↪pyright: ignore[reportArgumentType]
                meta={k: v for k, v in entry.payload.items() if k not in ["_score", "_
↪tensor_facets"]},
            )
            for entry in results
            if entry.payload is not None
        ]
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `collection_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `content_payload_key = field(default=CONTENT_PAYLOAD_KEY, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `distance = field(default=DEFAULT_DISTANCE, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `force_disable_check_same_thread = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `grpc_port = field(default=6334, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `host = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `https = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `location = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `path = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `port = field(default=6333, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `prefer_grpc = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `prefix = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `timeout = field(default=5, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `url = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `vector_name = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
@lazy_property()
def client(self) -> QdrantClient:
    return import_optional_dependency("qdrant_client").QdrantClient(
        location=self.location,
        url=self.url,
        host=self.host,
        path=self.path,
        port=self.port,
        prefer_grpc=self.prefer_grpc,
        grpc_port=self.grpc_port,
        api_key=self.api_key,
        https=self.https,
        prefix=self.prefix,
        force_disable_check_same_thread=self.force_disable_check_same_thread,
        timeout=self.timeout,
    )
```

delete_vector(vector_id)

Delete a vector from the Qdrant collection based on its ID.

Parameters

Name	Type	Description	Default
vector_id	*str I id*	ID of the vector to delete. 	*required*

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> None:
    """Delete a vector from the Qdrant collection based on its ID.

    Parameters:
        vector_id (str | id): ID of the vector to delete.
    """
    deletion_response = self.client.delete(
        collection_name=self.collection_name,
        points_selector=import_optional_dependency("qdrant_client.http.models").
↪PointIdsList(points=[vector_id]),
    )
    if deletion_response.status == import_optional_dependency("qdrant_client.http.
↪models").UpdateStatus.COMPLETED:
        logging.info("ID %s is successfully deleted", vector_id)
```

load_entries(*, namespace=None, **kwargs)

Load vector entries from the Qdrant collection.

Parameters

Name	Type	Description	Default
namespace	*Optional[str]*	Optional namespace of the vectors. 	*None*

Returns

Type	Description
<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	List of points.

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None, **kwargs) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Load vector entries from the Qdrant collection.

    Parameters:
        namespace: Optional namespace of the vectors.

    Returns:
        List of points.
    """
    results = self.client.retrieve(
        collection_name=self.collection_name,
        ids=kwargs.get("ids", []),
        with_payload=kwargs.get("with_payload", True),
        with_vectors=kwargs.get("with_vectors", True),
    )

    return [
        BaseVectorStoreDriver.Entry(
            id=str(entry.id),
            vector=entry.vector if kwargs.get("with_vectors", True) else [],  #␣
↪pyright: ignore[reportArgumentType]
            meta={k: v for k, v in entry.payload.items() if k not in ["_score", "_
↪tensor_facets"]},
        )
        for entry in results
        if entry.payload is not None
    ]
```

load_entry(vector_id, *, namespace=None)

Load a vector entry from the Qdrant collection based on its ID.

Parameters

Name	Type	Description	Default
vector_id	*str*	ID of the vector to load. 	*required*
namespace	*str*	Optional namespace of the vector. 	*None*

Returns

Type	Description
<a href="vector#griptape.drivers.vector.BaseVectorStoreDriver.Entry" title="Entry (griptape.drivers.vector.BaseVectorStoreDriver.Entry)"><code class="language-python">C	Optional[BaseVectorStoreDriver.Entry]: Vector entry if found, else None.

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    """Load a vector entry from the Qdrant collection based on its ID.

    Parameters:
        vector_id (str): ID of the vector to load.
        namespace (str, optional): Optional namespace of the vector.

    Returns:
        Optional[BaseVectorStoreDriver.Entry]: Vector entry if found, else None.
    """
    results = self.client.retrieve(collection_name=self.collection_name, ids=[vector_
↪id])
    if results:
        entry = results[0]
        if entry.payload is None:
            entry.payload = {}
        return BaseVectorStoreDriver.Entry(
            id=str(entry.id),
            vector=entry.vector if entry.vector is not None else [],  # pyright:␣
↪ignore[reportArgumentType]
            meta={k: v for k, v in entry.payload.items() if k not in ["_score", "_
↪tensor_facets"]},
        )
    return None
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs)

Query the Qdrant collection based on a query vector.

Parameters

Name	Type	Description	Default
vector	<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	Query vector. 	*required*
count	*Optional[int]*	Optional number of results to return. 	*None*
namespace	*Optional[str]*	Optional namespace of the vectors. 	*None*
include_vectors	*bool*	Whether to include vectors in the results. 	*False*

Returns

Type	Description
<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	list[BaseVectorStoreDriver.Entry]: List of Entry objects.

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    """Query the Qdrant collection based on a query vector.

    Parameters:
        vector (list[float]): Query vector.
        count (Optional[int]): Optional number of results to return.
        namespace (Optional[str]): Optional namespace of the vectors.
        include_vectors (bool): Whether to include vectors in the results.

    Returns:
```

(continues on next page)

(continued from previous page)

```
        list[BaseVectorStoreDriver.Entry]: List of Entry objects.
    """
    # Create a search request
    request = {"collection_name": self.collection_name, "query_vector": vector, "limit
↪": count}
    request = {k: v for k, v in request.items() if v is not None}
    results = self.client.search(**request)

    # Convert results to QueryResult objects
    return [
        BaseVectorStoreDriver.Entry(
            id=str(result.id),
            vector=result.vector if include_vectors else [],  # pyright:␣
↪ignore[reportArgumentType]
            score=result.score,
            meta={k: v for k, v in result.payload.items() if k not in ["_score", "_
↪tensor_facets"]},
        )
        for result in results
        if result.payload is not None
    ]
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, content=None, **kwargs)

Upsert vectors into the Qdrant collection.

Parameters

Name	Type	Description	Default
vector	<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	The vector to be upserted. 	*required*
vector_id	*Optional[str]*	Optional vector ID. 	*None*
namespace	*Optional[str]*	Optional namespace for the vector. 	*None*
meta	*Optional[dict]*	Optional dictionary containing metadata. 	*None*
content	*Optional[str]*	The text content to be included in the payload. 	*None*

Returns

Name	Type	Description
str	*str*	The ID of the upserted vector.

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    content: Optional[str] = None,
    **kwargs,
) -> str:
    """Upsert vectors into the Qdrant collection.

    Parameters:
        vector (list[float]): The vector to be upserted.
        vector_id (Optional[str]): Optional vector ID.
        namespace (Optional[str]): Optional namespace for the vector.
        meta (Optional[dict]): Optional dictionary containing metadata.
        content (Optional[str]): The text content to be included in the payload.

    Returns:
        str: The ID of the upserted vector.
    """
    if vector_id is None:
        vector_id = str(uuid.uuid5(uuid.NAMESPACE_DNS, str(vector)))

    if meta is None:
        meta = {}

    if content:
        meta[self.content_payload_key] = content

    points = import_optional_dependency("qdrant_client.http.models").Batch(
        ids=[vector_id],
        vectors=[vector],
        payloads=[meta] if meta else None,
    )

    self.client.upsert(collection_name=self.collection_name, points=points)
    return vector_id
```

RedisConversationMemoryDriver

Bases: *BaseConversationMemoryDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.RedisConversationMemoryDriver.host" title="host = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.RedisConversationMemoryDriver.host)"><code class="language-python">	*str*	The host of the Redis instance.
<a href="./#griptape.drivers.RedisConversationMemoryDriver.port" title="port = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.RedisConversationMemoryDriver.port)"><code class="language-python">	*int*	The port of the Redis instance.
<a href="./#griptape.drivers.RedisConversationMemoryDriver.db" title="db = field(kw_only=True, default=0, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.RedisConversationMemoryDriver.db)"><code class="language-python">	*int*	The database of the Redis instance.

continues on next page

Table 121 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.RedisConversationMemoryDriver.username" title="username = field(kw_only=True, default='default', metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.RedisConversationMemoryDriver.username)"><code class="language-python">	*str*	The username of the Redis instance.
<a href="./#griptape.drivers.RedisConversationMemoryDriver.password" title="password = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.RedisConversationMemoryDriver.password)"><code class="language-python">	*Optional[str]*	The password of the Redis instance.
<a href="./#griptape.drivers.RedisConversationMemoryDriver.index" title="index = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.RedisConversationMemoryDriver.index)"><code class="language-python">	*str*	The name of the index to use.

continues on next page

Table 121 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.RedisConversationMemoryDriver.conversation_id" title="conversation_id = field(kw_only=True, default=uuid.uuid4().hex) class-attribute instance-attribute (griptape.drivers.RedisConversationMemoryDriver.conversation_id)"><code class="language-python">	*str*	The id of the conversation.

Source Code in griptape/drivers/memory/conversation/redis_conversation_memory_driver.py

```
@define
class RedisConversationMemoryDriver(BaseConversationMemoryDriver):
    """A Conversation Memory Driver for Redis.

    This driver interfaces with a Redis instance and utilizes the Redis hashes and␣
↪RediSearch module to store,
    retrieve, and query conversations in a structured manner.
    Proper setup of the Redis instance and RediSearch is necessary for the driver to␣
↪function correctly.

    Attributes:
        host: The host of the Redis instance.
        port: The port of the Redis instance.
        db: The database of the Redis instance.
        username: The username of the Redis instance.
        password: The password of the Redis instance.
        index: The name of the index to use.
        conversation_id: The id of the conversation.
    """

    host: str = field(kw_only=True, metadata={"serializable": True})
    username: str = field(kw_only=True, default="default", metadata={"serializable":␣
↪False})
    port: int = field(kw_only=True, metadata={"serializable": True})
    db: int = field(kw_only=True, default=0, metadata={"serializable": True})
    password: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": False})
    index: str = field(kw_only=True, metadata={"serializable": True})
    conversation_id: str = field(kw_only=True, default=uuid.uuid4().hex)

    client: Redis = field(
        default=Factory(
```

(continues on next page)

(continued from previous page)

```
            lambda self: import_optional_dependency("redis").Redis(
                host=self.host,
                port=self.port,
                db=self.db,
                username=self.username,
                password=self.password,
                decode_responses=False,
            ),
            takes_self=True,
        ),
    )

    def store(self, runs: list[Run], metadata: dict[str, Any]) -> None:
        self.client.hset(self.index, self.conversation_id, json.dumps(self._to_params_
↪dict(runs, metadata)))

    def load(self) -> tuple[list[Run], dict[str, Any]]:
        memory_json = self.client.hget(self.index, self.conversation_id)
        if memory_json is not None:
            return self._from_params_dict(json.loads(memory_json))  # pyright:␣
↪ignore[reportArgumentType] https://github.com/redis/redis-py/issues/2399
        return [], {}
```

- `client = field(default=Factory(lambda self: import_optional_dependency('redis').Redis(host=self.host, port=self.port, db=self.db, username=self.username, password=self.password, decode_responses=False), takes_self=True))` class-attribute instance-attribute
- `conversation_id = field(kw_only=True, default=uuid.uuid4().hex)` class-attribute instance-attribute
- `db = field(kw_only=True, default=0, metadata={'serializable': True})` class-attribute instance-attribute
- `host = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `index = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `password = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `port = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `username = field(kw_only=True, default='default', metadata={'serializable': False})` class-attribute instance-attribute

load()

Source Code in griptape/drivers/memory/conversation/redis_conversation_memory_driver.py

```
def load(self) -> tuple[list[Run], dict[str, Any]]:
    memory_json = self.client.hget(self.index, self.conversation_id)
    if memory_json is not None:
```

(continues on next page)

(continued from previous page)

```
        return self._from_params_dict(json.loads(memory_json))  # pyright:␣
↪ignore[reportArgumentType] https://github.com/redis/redis-py/issues/2399
    return [], {}
```

store(runs, metadata)

Source Code in griptape/drivers/memory/conversation/redis_conversation_memory_driver.py

```
def store(self, runs: list[Run], metadata: dict[str, Any]) -> None:
    self.client.hset(self.index, self.conversation_id, json.dumps(self._to_params_
↪dict(runs, metadata)))
```

RedisVectorStoreDriver

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.RedisVectorStoreDriver.host" title="host = field(kw_only=True, metadata={'serializable': True})">class-attribute instance-attribute (griptape.drivers.RedisVectorStoreDriver.host)"><code class="language-python">	*str*	The host of the Redis instance.
<a href="./#griptape.drivers.RedisVectorStoreDriver.port" title="port = field(kw_only=True, metadata={'serializable': True})">class-attribute instance-attribute (griptape.drivers.RedisVectorStoreDriver.port)"><code class="language-python">	*int*	The port of the Redis instance.

continues on next page

Table 122 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.RedisVectorStoreDriver.db" title="db = field(kw_only=True, default=0, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.RedisVectorStoreDriver.db)"><code class="language-python">	*int*	The database of the Redis instance.
<a href="./#griptape.drivers.RedisVectorStoreDriver.username" title="username = field(kw_only=True, default='default', metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.RedisVectorStoreDriver.username)"><code class="language-python">	*str*	The username of the Redis instance.
<a href="./#griptape.drivers.RedisVectorStoreDriver.password" title="password = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.RedisVectorStoreDriver.password)"><code class="language-python">	*Optional[str]*	The password of the Redis instance.

continues on next page

Table 122 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.RedisVectorStoreDriver.index" title="index = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.RedisVectorStoreDriver.index)"><code class="language-python">	*str*	The name of the index to use.

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```python
@define
class RedisVectorStoreDriver(BaseVectorStoreDriver):
    """A Vector Store Driver for Redis.

    This driver interfaces with a Redis instance and utilizes the Redis hashes and RediSearch module to store, retrieve, and query vectors in a structured manner.
    Proper setup of the Redis instance and RediSearch is necessary for the driver to function correctly.

    Attributes:
        host: The host of the Redis instance.
        port: The port of the Redis instance.
        db: The database of the Redis instance.
        username: The username of the Redis instance.
        password: The password of the Redis instance.
        index: The name of the index to use.
    """

    host: str = field(kw_only=True, metadata={"serializable": True})
    username: str = field(kw_only=True, default="default", metadata={"serializable": False})
    port: int = field(kw_only=True, metadata={"serializable": True})
    db: int = field(kw_only=True, default=0, metadata={"serializable": True})
    password: Optional[str] = field(default=None, kw_only=True, metadata={"serializable": False})
    index: str = field(kw_only=True, metadata={"serializable": True})
    _client: Optional[Redis] = field(default=None, kw_only=True, alias="client", metadata={"serializable": False})

    @lazy_property()
    def client(self) -> Redis:
        return import_optional_dependency("redis").Redis(
            host=self.host,
            port=self.port,
```

(continues on next page)

(continued from previous page)

```
            db=self.db,
            username=self.username,
            password=self.password,
            decode_responses=False,
        )

    def upsert_vector(
        self,
        vector: list[float],
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        """Inserts or updates a vector in Redis.

        If a vector with the given vector ID already exists, it is updated; otherwise,
↪ a new vector is inserted.
        Metadata associated with the vector can also be provided.
        """
        vector_id = vector_id or str_to_hash(str(vector))
        key = self._generate_key(vector_id, namespace)
        bytes_vector = json.dumps(vector).encode("utf-8")

        mapping = {}
        mapping["vector"] = np.array(vector, dtype=np.float32).tobytes()
        mapping["vec_string"] = bytes_vector

        if namespace:
            mapping["namespace"] = namespace

        if meta:
            mapping["metadata"] = json.dumps(meta)

        self.client.hset(key, mapping=mapping)

        return vector_id

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        """Retrieves a specific vector entry from Redis based on its identifier and␣
↪optional namespace.

        Returns:
            If the entry is found, it returns an instance of BaseVectorStoreDriver.
↪Entry; otherwise, None is returned.
        """
        key = self._generate_key(vector_id, namespace)
        result = self.client.hgetall(key)
        vector = np.frombuffer(result[b"vector"], dtype=np.float32).tolist()  #␣
↪pyright: ignore[reportIndexIssue] https://github.com/redis/redis-py/issues/2399
        meta = json.loads(result[b"metadata"]) if b"metadata" in result else None  #␣
```

(continues on next page)

(continued from previous page)

```
↪pyright: ignore[reportIndexIssue, reportOperatorIssue]

        return BaseVectorStoreDriver.Entry(id=vector_id, meta=meta, vector=vector,␣
↪namespace=namespace)

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Retrieves all vector entries from Redis that match the optional namespace.

        Returns:
            A list of `BaseVectorStoreDriver.Entry` objects.
        """
        pattern = f"{namespace}:*" if namespace else "*"
        keys = self.client.keys(pattern)

        entries = []
        for key in keys:  # pyright: ignore[reportGeneralTypeIssues] https://github.
↪com/redis/redis-py/issues/2399
            entry = self.load_entry(key.decode("utf-8"), namespace=namespace)
            if entry:
                entries.append(entry)

        return entries

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Performs a nearest neighbor search on Redis to find vectors similar to the␣
↪provided input vector.

        Results can be limited using the count parameter and optionally filtered by a␣
↪namespace.

        Returns:
            A list of BaseVectorStoreDriver.Entry objects, each encapsulating the␣
↪retrieved vector, its similarity score, metadata, and namespace.
        """
        search_query = import_optional_dependency("redis.commands.search.query")

        filter_expression = f"(@namespace:{{{namespace}}})" if namespace else "*"
        query_expression = (
            search_query.Query(f"{filter_expression}=>[KNN {count or 10} @vector
↪$vector as score]")
            .sort_by("score")
            .return_fields("id", "score", "metadata", "vec_string")
            .paging(0, count or 10)
```

(continues on next page)

(continued from previous page)

```
            .dialect(2)
        )

        query_params = {"vector": np.array(vector, dtype=np.float32).tobytes()}

        results = self.client.ft(self.index).search(query_expression, query_params).
↪docs  # pyright: ignore[reportArgumentType, reportAttributeAccessIssue]

        query_results = []
        for document in results:
            metadata = json.loads(document.metadata) if hasattr(document, "metadata")␣
↪else None
            namespace = document.id.split(":")[0] if ":" in document.id else None
            vector_id = document.id.split(":")[1] if ":" in document.id else document.
↪id
            vector_float_list = json.loads(document.vec_string) if include_vectors␣
↪else None
            query_results.append(
                BaseVectorStoreDriver.Entry(
                    id=vector_id,
                    vector=vector_float_list,
                    score=float(document.score),
                    meta=metadata,
                    namespace=namespace,
                ),
            )
        return query_results

    def _generate_key(self, vector_id: str, namespace: Optional[str] = None) -> str:
        """Generates a Redis key using the provided vector ID and optionally a␣
↪namespace."""
        return f"{namespace}:{vector_id}" if namespace else vector_id

    def _get_doc_prefix(self, namespace: Optional[str] = None) -> str:
        """Get the document prefix based on the provided namespace."""
        return f"{namespace}:" if namespace else ""

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `db = field(kw_only=True, default=0, metadata={'serializable': True})` class-attribute instance-attribute
- `host = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `index = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `password = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute

- `port = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `username = field(kw_only=True, default='default', metadata={'serializable': False})` class-attribute instance-attribute

_generate_key(vector_id, namespace=None)

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def _generate_key(self, vector_id: str, namespace: Optional[str] = None) -> str:
    """Generates a Redis key using the provided vector ID and optionally a namespace."
↪"""
    return f"{namespace}:{vector_id}" if namespace else vector_id
```

_get_doc_prefix(namespace=None)

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def _get_doc_prefix(self, namespace: Optional[str] = None) -> str:
    """Get the document prefix based on the provided namespace."""
    return f"{namespace}:" if namespace else ""
```

client()

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
@lazy_property()
def client(self) -> Redis:
    return import_optional_dependency("redis").Redis(
        host=self.host,
        port=self.port,
        db=self.db,
        username=self.username,
        password=self.password,
        decode_responses=False,
    )
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

load_entries(*, namespace=None)

Retrieves all vector entries from Redis that match the optional namespace.

Returns

Type	Description
<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	A list of *BaseVectorStoreDriver.Entry* objects.

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Retrieves all vector entries from Redis that match the optional namespace.

    Returns:
        A list of `BaseVectorStoreDriver.Entry` objects.
    """
    pattern = f"{namespace}:*" if namespace else "*"
    keys = self.client.keys(pattern)

    entries = []
    for key in keys:  # pyright: ignore[reportGeneralTypeIssues] https://github.com/
↪redis/redis-py/issues/2399
        entry = self.load_entry(key.decode("utf-8"), namespace=namespace)
        if entry:
            entries.append(entry)

    return entries
```

load_entry(vector_id, *, namespace=None)

Retrieves a specific vector entry from Redis based on its identifier and optional namespace.

Returns

Type	Description
<a href="vector#griptape.drivers.vector.BaseVectorStoreDriver.Entry" title="Entry (griptape.drivers.vector.BaseVectorStoreDriver.Entry)"><code class="language-python">C	If the entry is found, it returns an instance of BaseVectorStoreDriver.Entry; otherwise, None is returned.

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    """Retrieves a specific vector entry from Redis based on its identifier and␣
↪optional namespace.

    Returns:
```

(continues on next page)

(continued from previous page)

```
        If the entry is found, it returns an instance of BaseVectorStoreDriver.Entry;␣
↪otherwise, None is returned.
    """
    key = self._generate_key(vector_id, namespace)
    result = self.client.hgetall(key)
    vector = np.frombuffer(result[b"vector"], dtype=np.float32).tolist()  # pyright:␣
↪ignore[reportIndexIssue] https://github.com/redis/redis-py/issues/2399
    meta = json.loads(result[b"metadata"]) if b"metadata" in result else None  #␣
↪pyright: ignore[reportIndexIssue, reportOperatorIssue]

    return BaseVectorStoreDriver.Entry(id=vector_id, meta=meta, vector=vector,␣
↪namespace=namespace)
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs)

Performs a nearest neighbor search on Redis to find vectors similar to the provided input vector.

Results can be limited using the count parameter and optionally filtered by a namespace.

Returns

Type	Description
<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	A list of BaseVectorStoreDriver.Entry objects, each encapsulating the retrieved vector, its similarity score, metadata, and namespace.

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    """Performs a nearest neighbor search on Redis to find vectors similar to the␣
↪provided input vector.

    Results can be limited using the count parameter and optionally filtered by a␣
↪namespace.

    Returns:
        A list of BaseVectorStoreDriver.Entry objects, each encapsulating the␣
↪retrieved vector, its similarity score, metadata, and namespace.
    """
    search_query = import_optional_dependency("redis.commands.search.query")

    filter_expression = f"(@namespace:{{{namespace}}})" if namespace else "*"
```

(continues on next page)

(continued from previous page)

```
    query_expression = (
        search_query.Query(f"{filter_expression}=>[KNN {count or 10} @vector $vector␣
↪as score]")
        .sort_by("score")
        .return_fields("id", "score", "metadata", "vec_string")
        .paging(0, count or 10)
        .dialect(2)
    )

    query_params = {"vector": np.array(vector, dtype=np.float32).tobytes()}

    results = self.client.ft(self.index).search(query_expression, query_params).docs
↪# pyright: ignore[reportArgumentType, reportAttributeAccessIssue]

    query_results = []
    for document in results:
        metadata = json.loads(document.metadata) if hasattr(document, "metadata")␣
↪else None
        namespace = document.id.split(":")[0] if ":" in document.id else None
        vector_id = document.id.split(":")[1] if ":" in document.id else document.id
        vector_float_list = json.loads(document.vec_string) if include_vectors else␣
↪None
        query_results.append(
            BaseVectorStoreDriver.Entry(
                id=vector_id,
                vector=vector_float_list,
                score=float(document.score),
                meta=metadata,
                namespace=namespace,
            ),
        )
    return query_results
```

upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    """Inserts or updates a vector in Redis.

    If a vector with the given vector ID already exists, it is updated; otherwise, a␣
↪new vector is inserted.
    Metadata associated with the vector can also be provided.
    """
    vector_id = vector_id or str_to_hash(str(vector))
```

(continues on next page)

(continued from previous page)

```
    key = self._generate_key(vector_id, namespace)
    bytes_vector = json.dumps(vector).encode("utf-8")

    mapping = {}
    mapping["vector"] = np.array(vector, dtype=np.float32).tobytes()
    mapping["vec_string"] = bytes_vector

    if namespace:
        mapping["namespace"] = namespace

    if meta:
        mapping["metadata"] = json.dumps(meta)

    self.client.hset(key, mapping=mapping)

    return vector_id
```

SnowflakeSqlDriver

Bases: *BaseSqlDriver* “)

Source Code in griptape/drivers/sql/snowflake_sql_driver.py

```
@define
class SnowflakeSqlDriver(BaseSqlDriver):
    get_connection: Callable[[], SnowflakeConnection] = field(kw_only=True)
    _engine: Optional[Engine] = field(default=None, kw_only=True, alias="engine",␣
↪metadata={"serializable": False})

    @get_connection.validator  # pyright: ignore[reportFunctionMemberAccess]
    def validate_get_connection(self, _: Attribute, get_connection: Callable[[],␣
↪SnowflakeConnection]) -> None:
        snowflake_connection = get_connection()
        snowflake = import_optional_dependency("snowflake")

        if not isinstance(snowflake_connection, snowflake.connector.
↪SnowflakeConnection):
            raise ValueError("The get_connection function must return a␣
↪SnowflakeConnection")
        if not snowflake_connection.schema or not snowflake_connection.database:
            raise ValueError("Provide a schema and database for the Snowflake␣
↪connection")

    @lazy_property()
    def engine(self) -> Engine:
        return import_optional_dependency("sqlalchemy").create_engine(
            "snowflake://not@used/db",
            creator=self.get_connection,
        )

    def execute_query(self, query: str) -> Optional[list[BaseSqlDriver.RowResult]]:
        rows = self.execute_query_raw(query)
```

(continues on next page)

(continued from previous page)

```
        if rows:
            return [BaseSqlDriver.RowResult(row) for row in rows]
        return None

    def execute_query_raw(self, query: str) -> Optional[list[dict[str, Any]]]:
        sqlalchemy = import_optional_dependency("sqlalchemy")

        with self.engine.connect() as con:
            results = con.execute(sqlalchemy.text(query))

            if results is not None:
                if results.returns_rows:
                    return [dict(result._mapping) for result in results]
                return None
            raise ValueError("No results found")

    def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]:
        sqlalchemy = import_optional_dependency("sqlalchemy")

        try:
            metadata_obj = sqlalchemy.MetaData()
            metadata_obj.reflect(bind=self.engine)
            table = sqlalchemy.Table(table_name, metadata_obj, schema=schema,␣
↪autoload=True, autoload_with=self.engine)
            return str([(c.name, c.type) for c in table.columns])
        except sqlalchemy.exc.NoSuchTableError:
            return None
```

- `_engine = field(default=None, kw_only=True, alias='engine', metadata={'serializable': False})` class-attribute instance-attribute
- `get_connection = field(kw_only=True)` class-attribute instance-attribute

engine()

Source Code in griptape/drivers/sql/snowflake_sql_driver.py

```
@lazy_property()
def engine(self) -> Engine:
    return import_optional_dependency("sqlalchemy").create_engine(
        "snowflake://not@used/db",
        creator=self.get_connection,
    )
```

execute_query(query)

Source Code in griptape/drivers/sql/snowflake_sql_driver.py

```
def execute_query(self, query: str) -> Optional[list[BaseSqlDriver.RowResult]]:
    rows = self.execute_query_raw(query)
```

(continues on next page)

(continued from previous page)

```
    if rows:
        return [BaseSqlDriver.RowResult(row) for row in rows]
    return None
```

execute_query_raw(query)

Source Code in griptape/drivers/sql/snowflake_sql_driver.py

```
def execute_query_raw(self, query: str) -> Optional[list[dict[str, Any]]]:
    sqlalchemy = import_optional_dependency("sqlalchemy")

    with self.engine.connect() as con:
        results = con.execute(sqlalchemy.text(query))

        if results is not None:
            if results.returns_rows:
                return [dict(result._mapping) for result in results]
            return None
        raise ValueError("No results found")
```

get_table_schema(table_name, schema=None)

Source Code in griptape/drivers/sql/snowflake_sql_driver.py

```
def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]:
    sqlalchemy = import_optional_dependency("sqlalchemy")

    try:
        metadata_obj = sqlalchemy.MetaData()
        metadata_obj.reflect(bind=self.engine)
        table = sqlalchemy.Table(table_name, metadata_obj, schema=schema,␣
↪autoload=True, autoload_with=self.engine)
        return str([(c.name, c.type) for c in table.columns])
    except sqlalchemy.exc.NoSuchTableError:
        return None
```

validate*get_connection(*, get_connection)

Source Code in griptape/drivers/sql/snowflake_sql_driver.py

```
@get_connection.validator  # pyright: ignore[reportFunctionMemberAccess]
def validate_get_connection(self, _: Attribute, get_connection: Callable[[],␣
↪SnowflakeConnection]) -> None:
    snowflake_connection = get_connection()
    snowflake = import_optional_dependency("snowflake")

    if not isinstance(snowflake_connection, snowflake.connector.SnowflakeConnection):
        raise ValueError("The get_connection function must return a␣
↪SnowflakeConnection")
    if not snowflake_connection.schema or not snowflake_connection.database:
        raise ValueError("Provide a schema and database for the Snowflake connection")
```

SqlDriver

Bases: *BaseSqlDriver* “)

Source Code in griptape/drivers/sql/sql_driver.py

```
@define
class SqlDriver(BaseSqlDriver):
    engine_url: str = field(kw_only=True)
    create_engine_params: dict = field(factory=dict, kw_only=True)
    _engine: Optional[Engine] = field(default=None, kw_only=True, alias="engine",␣
↪metadata={"serializable": False})

    @lazy_property()
    def engine(self) -> Engine:
        return import_optional_dependency("sqlalchemy").create_engine(self.engine_url,
↪ **self.create_engine_params)

    def execute_query(self, query: str) -> Optional[list[BaseSqlDriver.RowResult]]:
        rows = self.execute_query_raw(query)

        if rows:
            return [BaseSqlDriver.RowResult(row) for row in rows]
        return None

    def execute_query_raw(self, query: str) -> Optional[list[dict[str,␣
↪Optional[Any]]]]:
        sqlalchemy = import_optional_dependency("sqlalchemy")

        with self.engine.connect() as con:
            results = con.execute(sqlalchemy.text(query))

            if results is not None:
                if results.returns_rows:
                    return [dict(result._mapping) for result in results]
                con.commit()
                return None
            raise ValueError("No result found")

    def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]:
        sqlalchemy_exc = import_optional_dependency("sqlalchemy.exc")

        try:
            return str(SqlDriver._get_table_schema(self.engine, table_name, schema))
        except sqlalchemy_exc.NoSuchTableError:
            return None

    @staticmethod
    @lru_cache
    def _get_table_schema(
        engine: Engine, table_name: str, schema: Optional[str] = None
    ) -> Optional[list[tuple[str, str]]]:
        sqlalchemy = import_optional_dependency("sqlalchemy")
```

(continues on next page)

(continued from previous page)

```
        return [(col["name"], col["type"]) for col in sqlalchemy.inspect(engine).get_
↪columns(table_name, schema=schema)]
```

- `_engine = field(default=None, kw_only=True, alias='engine', metadata={'serializable': False})` class-attribute instance-attribute
- `create_engine_params = field(factory=dict, kw_only=True)` class-attribute instance-attribute
- `engine_url = field(kw_only=True)` class-attribute instance-attribute

_get_table_schema(engine, table_name, schema=None)cachedstaticmethod

Source Code in griptape/drivers/sql/sql_driver.py

```
@staticmethod
@lru_cache
def _get_table_schema(
    engine: Engine, table_name: str, schema: Optional[str] = None
) -> Optional[list[tuple[str, str]]]:
    sqlalchemy = import_optional_dependency("sqlalchemy")

    return [(col["name"], col["type"]) for col in sqlalchemy.inspect(engine).get_
↪columns(table_name, schema=schema)]
```

engine()

Source Code in griptape/drivers/sql/sql_driver.py

```
@lazy_property()
def engine(self) -> Engine:
    return import_optional_dependency("sqlalchemy").create_engine(self.engine_url,␣
↪**self.create_engine_params)
```

execute_query(query)

Source Code in griptape/drivers/sql/sql_driver.py

```
def execute_query(self, query: str) -> Optional[list[BaseSqlDriver.RowResult]]:
    rows = self.execute_query_raw(query)

    if rows:
        return [BaseSqlDriver.RowResult(row) for row in rows]
    return None
```

execute_query_raw(query)

Source Code in griptape/drivers/sql/sql_driver.py

```
def execute_query_raw(self, query: str) -> Optional[list[dict[str, Optional[Any]]]]:
    sqlalchemy = import_optional_dependency("sqlalchemy")

    with self.engine.connect() as con:
```

(continues on next page)

(continued from previous page)

```
        results = con.execute(sqlalchemy.text(query))

        if results is not None:
            if results.returns_rows:
                return [dict(result._mapping) for result in results]
            con.commit()
            return None
        raise ValueError("No result found")
```

get_table_schema(table_name, schema=None)

Source Code in griptape/drivers/sql/sql_driver.py

```
def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]:
    sqlalchemy_exc = import_optional_dependency("sqlalchemy.exc")

    try:
        return str(SqlDriver._get_table_schema(self.engine, table_name, schema))
    except sqlalchemy_exc.NoSuchTableError:
        return None
```

StableDiffusion3ControlNetImageGenerationPipelineDriver

Bases: *StableDiffusion3ImageGenerationPipelineDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.StableDiffusion3ControlNetImageGenerationPipelineDriver.controlnet_model" title="controlnet_model = field(kw_only=True) class-attribute instance-attribute (griptape.drivers.StableDiffusion3ControlNetImageGenerationPipelineDriver.controlnet_model)"><code class="language-python">	*str*	The ControlNet model to use for image generation.

continues on next page

Table 126 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.StableDiffusion3ControlNetImageGenerationPipelineDriver.controlnet_conditioning_scale" title="controlnet_conditioning_scale = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.StableDiffusion3ControlNetImageGenerationPipelineDriver.controlnet_conditioning_scale)"><code class="language-python">	*Optional[float]*	The conditioning scale for the ControlNet model. Defaults to None.

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_controlnet_image_generation_pipeline_driver.py

```
@define
class StableDiffusion3ControlNetImageGenerationPipelineDriver(StableDiffusion3ImageGenerationPipelineDriv
    """Image generation model driver for Stable Diffusion 3 models with ControlNet.

    For more information, see the HuggingFace documentation for the StableDiffusion3ControlNetPipeline:
        https://huggingface.co/docs/diffusers/en/api/pipelines/controlnet_sd3

    Attributes:
        controlnet_model: The ControlNet model to use for image generation.
        controlnet_conditioning_scale: The conditioning scale for the ControlNet model. Defaults to None.
    """

    controlnet_model: str = field(kw_only=True)
    controlnet_conditioning_scale: Optional[float] = field(default=None, kw_only=True, metadata={"serializable": True})

    def prepare_pipeline(self, model: str, device: Optional[str]) -> Any:
        sd3_controlnet_model = import_optional_dependency("diffusers.models.controlnet_sd3").SD3ControlNetModel
        sd3_controlnet_pipeline = import_optional_dependency(
            "diffusers.pipelines.controlnet_sd3.pipeline_stable_diffusion_3_controlnet"
```

(continues on next page)

(continued from previous page)

```
        ).StableDiffusion3ControlNetPipeline

        pipeline_params = {}
        controlnet_pipeline_params = {}
        if self.torch_dtype is not None:
            pipeline_params["torch_dtype"] = self.torch_dtype
            controlnet_pipeline_params["torch_dtype"] = self.torch_dtype

        if self.drop_t5_encoder:
            pipeline_params["text_encoder_3"] = None
            pipeline_params["tokenizer_3"] = None

        # For both Stable Diffusion and ControlNet, models can be provided either
        # as a path to a local file or as a HuggingFace model repo name.
        # We use the from_single_file method if the model is a local file and the
        # from_pretrained method if the model is a local directory or hosted on␣
↪HuggingFace.
        if os.path.isfile(self.controlnet_model):
            pipeline_params["controlnet"] = sd3_controlnet_model.from_single_file(
                self.controlnet_model, **controlnet_pipeline_params
            )
        else:
            pipeline_params["controlnet"] = sd3_controlnet_model.from_pretrained(
                self.controlnet_model, **controlnet_pipeline_params
            )

        if os.path.isfile(model):
            pipeline = sd3_controlnet_pipeline.from_single_file(model, **pipeline_
↪params)
        else:
            pipeline = sd3_controlnet_pipeline.from_pretrained(model, **pipeline_
↪params)

        if self.enable_model_cpu_offload:
            pipeline.enable_model_cpu_offload()

        if device is not None:
            pipeline.to(device)

        return pipeline

    def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:
        if image is None:
            raise ValueError("Input image is required for ControlNet pipelines.")

        return {"control_image": image}

    def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict[str, Any]:
        additional_params = super().make_additional_params(negative_prompts, device)

        del additional_params["height"]
```

(continues on next page)

(continued from previous page)

```
        del additional_params["width"]

        if self.controlnet_conditioning_scale is not None:
            additional_params["controlnet_conditioning_scale"] = self.controlnet_
↪conditioning_scale

        return additional_params
```

- `controlnet_conditioning_scale = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `controlnet_model = field(kw_only=True)` class-attribute instance-attribute

make_additional_params(negative_prompts, device)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_controlnet_image_generation_pipeline_driver.py

```
def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict[str, Any]:
    additional_params = super().make_additional_params(negative_prompts, device)

    del additional_params["height"]
    del additional_params["width"]

    if self.controlnet_conditioning_scale is not None:
        additional_params["controlnet_conditioning_scale"] = self.controlnet_
↪conditioning_scale

    return additional_params
```

make_image_param(image)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_controlnet_image_generation_pipeline_driver.py

```
def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:
    if image is None:
        raise ValueError("Input image is required for ControlNet pipelines.")

    return {"control_image": image}
```

prepare_pipeline(model, device)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_controlnet_image_generation_pipeline_driver.py

```
def prepare_pipeline(self, model: str, device: Optional[str]) -> Any:
    sd3_controlnet_model = import_optional_dependency("diffusers.models.controlnet_sd3
↪").SD3ControlNetModel
    sd3_controlnet_pipeline = import_optional_dependency(
        "diffusers.pipelines.controlnet_sd3.pipeline_stable_diffusion_3_controlnet"
    ).StableDiffusion3ControlNetPipeline
```

(continues on next page)

(continued from previous page)

```
    pipeline_params = {}
    controlnet_pipeline_params = {}
    if self.torch_dtype is not None:
        pipeline_params["torch_dtype"] = self.torch_dtype
        controlnet_pipeline_params["torch_dtype"] = self.torch_dtype

    if self.drop_t5_encoder:
        pipeline_params["text_encoder_3"] = None
        pipeline_params["tokenizer_3"] = None

    # For both Stable Diffusion and ControlNet, models can be provided either
    # as a path to a local file or as a HuggingFace model repo name.
    # We use the from_single_file method if the model is a local file and the
    # from_pretrained method if the model is a local directory or hosted on␣
↪HuggingFace.
    if os.path.isfile(self.controlnet_model):
        pipeline_params["controlnet"] = sd3_controlnet_model.from_single_file(
            self.controlnet_model, **controlnet_pipeline_params
        )
    else:
        pipeline_params["controlnet"] = sd3_controlnet_model.from_pretrained(
            self.controlnet_model, **controlnet_pipeline_params
        )

    if os.path.isfile(model):
        pipeline = sd3_controlnet_pipeline.from_single_file(model, **pipeline_params)
    else:
        pipeline = sd3_controlnet_pipeline.from_pretrained(model, **pipeline_params)

    if self.enable_model_cpu_offload:
        pipeline.enable_model_cpu_offload()

    if device is not None:
        pipeline.to(device)

    return pipeline
```

StableDiffusion3ImageGenerationPipelineDriver

Bases: *BaseDiffusionImageGenerationPipelineDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.StableDiffusion3ImageGenerationPipelineDriver.width" title="width = field(default=1024, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.StableDiffusion3ImageGenerationPipelineDriver.width) class="language-python">	*int*	The width of the generated image. Defaults to 1024. Must be a multiple of 64.
<a href="./#griptape.drivers.StableDiffusion3ImageGenerationPipelineDriver.height" title="height = field(default=1024, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.StableDiffusion3ImageGenerationPipelineDriver.height) class="language-python">	*int*	The height of the generated image. Defaults to 1024. Must be a multiple of 64.
<a href="./#griptape.drivers.StableDiffusion3ImageGenerationPipelineDriver.seed" title="seed = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.StableDiffusion3ImageGenerationPipelineDriver.seed)" class="language-python">	*Optional[int]*	The random seed to use for image generation. If not provided, a random seed will be used.

continues on next page

Table 127 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.StableDiffusion3ImageGenerationPipelineDriver.guidance_scale" title="guidance_scale = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.StableDiffusion3ImageGenerationPipelineDriver.guidance_scale)"><code class="language-python">	*Optional[float]*	The strength of the guidance loss. If not provided, the default value will be used.
<a href="./#griptape.drivers.StableDiffusion3ImageGenerationPipelineDriver.steps" title="steps = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.StableDiffusion3ImageGenerationPipelineDriver.steps)" class="language-python">	*Optional[int]*	The number of inference steps to use in image generation. If not provided, the default value will be used.
<a href="./#griptape.drivers.StableDiffusion3ImageGenerationPipelineDriver.torch_ title="torch_dtype = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.StableDiffusion3ImageGenerationPipelineDriver.torch_ class="language-python">	*Optional[dtype]*	The torch data type to use for image generation. If not provided, the default value will be used.

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_image_generation_pipeline_driver.py

```
@define
class␣
↪StableDiffusion3ImageGenerationPipelineDriver(BaseDiffusionImageGenerationPipelineDriver):
```

(continues on next page)

(continued from previous page)

```
    """Image generation model driver for Stable Diffusion 3 models.

    For more information, see the HuggingFace documentation for the␣
↪StableDiffusion3Pipeline:
        https://huggingface.co/docs/diffusers/main/en/api/pipelines/stable_diffusion/
↪stable_diffusion_3

    Attributes:
        width: The width of the generated image. Defaults to 1024. Must be a multiple␣
↪of 64.
        height: The height of the generated image. Defaults to 1024. Must be a␣
↪multiple of 64.
        seed: The random seed to use for image generation. If not provided, a random␣
↪seed will be used.
        guidance_scale: The strength of the guidance loss. If not provided, the␣
↪default value will be used.
        steps: The number of inference steps to use in image generation. If not␣
↪provided, the default value will be used.
        torch_dtype: The torch data type to use for image generation. If not provided,
↪ the default value will be used.
    """

    width: int = field(default=1024, kw_only=True, metadata={"serializable": True})
    height: int = field(default=1024, kw_only=True, metadata={"serializable": True})
    seed: Optional[int] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    guidance_scale: Optional[float] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    steps: Optional[int] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    torch_dtype: Optional[torch.dtype] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    enable_model_cpu_offload: bool = field(default=False, kw_only=True, metadata={
↪"serializable": True})
    drop_t5_encoder: bool = field(default=False, kw_only=True, metadata={"serializable
↪": True})

    def prepare_pipeline(self, model: str, device: Optional[str]) -> Any:
        sd3_pipeline = import_optional_dependency(
            "diffusers.pipelines.stable_diffusion_3.pipeline_stable_diffusion_3"
        ).StableDiffusion3Pipeline

        pipeline_params = {}
        if self.torch_dtype is not None:
            pipeline_params["torch_dtype"] = self.torch_dtype

        if self.drop_t5_encoder:
            pipeline_params["text_encoder_3"] = None
            pipeline_params["tokenizer_3"] = None

        # A model can be provided either as a path to a local file
        # or as a HuggingFace model repo name.
```

(continues on next page)

(continued from previous page)

```
        if os.path.isfile(model):
            # If the model provided is a local file (not a directory),
            # we load it using the from_single_file method.
            pipeline = sd3_pipeline.from_single_file(model, **pipeline_params)
        else:
            # If the model is a local directory or hosted on HuggingFace,
            # we load it using the from_pretrained method.
            pipeline = sd3_pipeline.from_pretrained(model, **pipeline_params)

        if self.enable_model_cpu_offload:
            pipeline.enable_model_cpu_offload()

        # Move inference to particular device if requested.
        if device is not None:
            pipeline.to(device)

        return pipeline

    def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:
        return None

    def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict[str, Any]:
        torch_generator = import_optional_dependency("torch").Generator

        additional_params = {}
        if negative_prompts:
            additional_params["negative_prompt"] = ", ".join(negative_prompts)

        if self.width is not None:
            additional_params["width"] = self.width

        if self.height is not None:
            additional_params["height"] = self.height

        if self.seed is not None:
            additional_params["generator"] = [torch_generator(device=device).manual_
↪seed(self.seed)]

        if self.guidance_scale is not None:
            additional_params["guidance_scale"] = self.guidance_scale

        if self.steps is not None:
            additional_params["num_inference_steps"] = self.steps

        return additional_params

    @property
    def output_image_dimensions(self) -> tuple[int, int]:
        return self.width, self.height
```

- `drop_t5_encoder = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

- `enable_model_cpu_offload = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `guidance_scale = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `height = field(default=1024, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output_image_dimensions` property
- `seed = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `steps = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `torch_dtype = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `width = field(default=1024, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

make_additional_params(negative_prompts, device)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_image_generation_pipeline_driver.py

```
def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict[str, Any]:
    torch_generator = import_optional_dependency("torch").Generator

    additional_params = {}
    if negative_prompts:
        additional_params["negative_prompt"] = ", ".join(negative_prompts)

    if self.width is not None:
        additional_params["width"] = self.width

    if self.height is not None:
        additional_params["height"] = self.height

    if self.seed is not None:
        additional_params["generator"] = [torch_generator(device=device).manual_
↪seed(self.seed)]

    if self.guidance_scale is not None:
        additional_params["guidance_scale"] = self.guidance_scale

    if self.steps is not None:
        additional_params["num_inference_steps"] = self.steps

    return additional_params
```

make_image_param(image)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_image_generation_pipeline_driver.py

```
def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:
    return None
```

prepare_pipeline(model, device)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_image_generation_pipeline_driver.py

```
def prepare_pipeline(self, model: str, device: Optional[str]) -> Any:
    sd3_pipeline = import_optional_dependency(
        "diffusers.pipelines.stable_diffusion_3.pipeline_stable_diffusion_3"
    ).StableDiffusion3Pipeline

    pipeline_params = {}
    if self.torch_dtype is not None:
        pipeline_params["torch_dtype"] = self.torch_dtype

    if self.drop_t5_encoder:
        pipeline_params["text_encoder_3"] = None
        pipeline_params["tokenizer_3"] = None

    # A model can be provided either as a path to a local file
    # or as a HuggingFace model repo name.
    if os.path.isfile(model):
        # If the model provided is a local file (not a directory),
        # we load it using the from_single_file method.
        pipeline = sd3_pipeline.from_single_file(model, **pipeline_params)
    else:
        # If the model is a local directory or hosted on HuggingFace,
        # we load it using the from_pretrained method.
        pipeline = sd3_pipeline.from_pretrained(model, **pipeline_params)

    if self.enable_model_cpu_offload:
        pipeline.enable_model_cpu_offload()

    # Move inference to particular device if requested.
    if device is not None:
        pipeline.to(device)

    return pipeline
```

StableDiffusion3Img2ImgImageGenerationPipelineDriver

Bases: *StableDiffusion3ImageGenerationPipelineDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.StableDiffusion3Img2ImgImageGenerationPipelineDriver.strengt title="strength = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.StableDiffusion3Img2ImgImageGenerationPipelineDriver.strengt class="language-python">	*Optional[float]*	A value [0.0, 1.0] that determines the strength of the initial image in the output.

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_img_2_img_image_generation_pipeline_driver.py

```
@define
class␣
↪StableDiffusion3Img2ImgImageGenerationPipelineDriver(StableDiffusion3ImageGenerationPipelineDriver)
    """Image generation model driver for Stable Diffusion 3 model image to image␣
↪pipelines.

    For more information, see the HuggingFace documentation for the␣
↪StableDiffusion3Img2ImgPipeline:
        https://github.com/huggingface/diffusers/blob/main/src/diffusers/pipelines/
↪stable_diffusion_3/pipeline_stable_diffusion_3_img2img.py

    Attributes:
        strength: A value [0.0, 1.0] that determines the strength of the initial␣
↪image in the output.
    """

    strength: Optional[float] = field(default=None, kw_only=True, metadata={
↪"serializable": True})

    def prepare_pipeline(self, model: str, device: Optional[str]) -> Any:
        sd3_img2img_pipeline = import_optional_dependency(
            "diffusers.pipelines.stable_diffusion_3.pipeline_stable_diffusion_3_
↪img2img"
        ).StableDiffusion3Img2ImgPipeline

        pipeline_params = {}
        if self.torch_dtype is not None:
            pipeline_params["torch_dtype"] = self.torch_dtype

        if self.drop_t5_encoder:
            pipeline_params["text_encoder_3"] = None
```

(continues on next page)

(continued from previous page)

```
            pipeline_params["tokenizer_3"] = None

        # A model can be provided either as a path to a local file
        # or as a HuggingFace model repo name.
        if os.path.isfile(model):
            # If the model provided is a local file (not a directory),
            # we load it using the from_single_file method.
            pipeline = sd3_img2img_pipeline.from_single_file(model, **pipeline_params)
        else:
            # If the model is a local directory or hosted on HuggingFace,
            # we load it using the from_pretrained method.
            pipeline = sd3_img2img_pipeline.from_pretrained(model, **pipeline_params)

        if self.enable_model_cpu_offload:
            pipeline.enable_model_cpu_offload()

        # Move inference to particular device if requested.
        if device is not None:
            pipeline.to(device)

        return pipeline

    def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:
        if image is None:
            raise ValueError("Input image is required for image to image pipelines.")

        return {"image": image}

    def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict[str, Any]:
        additional_params = super().make_additional_params(negative_prompts, device)

        # Explicit height and width params are not supported, but
        # are instead inferred from input image.
        del additional_params["height"]
        del additional_params["width"]

        if self.strength is not None:
            additional_params["strength"] = self.strength

        return additional_params
```

- `strength = field(default=None, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute

make_additional_params(negative_prompts, device)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_img_2_img_image_generation_pipeline_driver.py

```
def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict[str, Any]:
```

(continues on next page)

(continued from previous page)

```
    additional_params = super().make_additional_params(negative_prompts, device)

    # Explicit height and width params are not supported, but
    # are instead inferred from input image.
    del additional_params["height"]
    del additional_params["width"]

    if self.strength is not None:
        additional_params["strength"] = self.strength

    return additional_params
```

make_image_param(image)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_img_2_img_image_generation_pipeline_driver.py

```
def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:
    if image is None:
        raise ValueError("Input image is required for image to image pipelines.")

    return {"image": image}
```

prepare_pipeline(model, device)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_img_2_img_image_generation_pipeline_driver.py

```
def prepare_pipeline(self, model: str, device: Optional[str]) -> Any:
    sd3_img2img_pipeline = import_optional_dependency(
        "diffusers.pipelines.stable_diffusion_3.pipeline_stable_diffusion_3_img2img"
    ).StableDiffusion3Img2ImgPipeline

    pipeline_params = {}
    if self.torch_dtype is not None:
        pipeline_params["torch_dtype"] = self.torch_dtype

    if self.drop_t5_encoder:
        pipeline_params["text_encoder_3"] = None
        pipeline_params["tokenizer_3"] = None

    # A model can be provided either as a path to a local file
    # or as a HuggingFace model repo name.
    if os.path.isfile(model):
        # If the model provided is a local file (not a directory),
        # we load it using the from_single_file method.
        pipeline = sd3_img2img_pipeline.from_single_file(model, **pipeline_params)
    else:
        # If the model is a local directory or hosted on HuggingFace,
        # we load it using the from_pretrained method.
        pipeline = sd3_img2img_pipeline.from_pretrained(model, **pipeline_params)
```

(continues on next page)

(continued from previous page)

```
    if self.enable_model_cpu_offload:
        pipeline.enable_model_cpu_offload()

    # Move inference to particular device if requested.
    if device is not None:
        pipeline.to(device)

    return pipeline
```

TavilyWebSearchDriver

Bases: *BaseWebSearchDriver* “)

Source Code in griptape/drivers/web_search/tavily_web_search_driver.py

```
@define
class TavilyWebSearchDriver(BaseWebSearchDriver):
    api_key: str = field(kw_only=True)
    params: dict[str, Any] = field(factory=dict, kw_only=True, metadata={"serializable
↪": True})
    _client: Optional[TavilyClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> TavilyClient:
        return import_optional_dependency("tavily").TavilyClient(self.api_key)

    def search(self, query: str, **kwargs) -> ListArtifact:
        response = self.client.search(query, max_results=self.results_count, **self.
↪params, **kwargs)
        results = response.get("results", [])
        return ListArtifact([(JsonArtifact(result)) for result in results])
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(kw_only=True)` class-attribute instance-attribute
- `params = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/web_search/tavily_web_search_driver.py

```
@lazy_property()
def client(self) -> TavilyClient:
    return import_optional_dependency("tavily").TavilyClient(self.api_key)
```

search(query, **kwargs)

Source Code in griptape/drivers/web_search/tavily_web_search_driver.py

```
def search(self, query: str, **kwargs) -> ListArtifact:
    response = self.client.search(query, max_results=self.results_count, **self.
↪params, **kwargs)
    results = response.get("results", [])
    return ListArtifact([(JsonArtifact(result)) for result in results])
```

TrafilaturaWebScraperDriver

Bases: *BaseWebScraperDriver* “)

Source Code in griptape/drivers/web_scraper/trafilatura_web_scraper_driver.py

```
@define
class TrafilaturaWebScraperDriver(BaseWebScraperDriver):
    include_links: bool = field(default=True, kw_only=True)
    no_ssl: bool = field(default=False, kw_only=True)

    def fetch_url(self, url: str) -> str:
        trafilatura = import_optional_dependency("trafilatura")
        use_config = trafilatura.settings.use_config

        config = use_config()
        page = trafilatura.fetch_url(url, no_ssl=self.no_ssl)

        # This disables signal, so that trafilatura can work on any thread:
        # More info: https://trafilatura.readthedocs.io/usage-python.html#disabling-
↪signal
        config.set("DEFAULT", "EXTRACTION_TIMEOUT", "0")

        # Disable error logging in trafilatura as it sometimes logs errors from lxml,␣
↪even though
        # the end result of page parsing is successful.
        logging.getLogger("trafilatura").setLevel(logging.FATAL)

        if page is None:
            raise Exception("cant access URL")

        return page

    def extract_page(self, page: str) -> TextArtifact:
        trafilatura = import_optional_dependency("trafilatura")
        use_config = trafilatura.settings.use_config

        config = use_config()

        extracted_page = trafilatura.extract(
            page,
            include_links=self.include_links,
            output_format="json",
            config=config,
```

(continues on next page)

(continued from previous page)

```
        )

        if not extracted_page:
            raise Exception("cant extract page")

        text = json.loads(extracted_page).get("text")

        return TextArtifact(text)
```

- `include_links = field(default=True, kw_only=True)` class-attribute instance-attribute
- `no_ssl = field(default=False, kw_only=True)` class-attribute instance-attribute

extract_page(page)

Source Code in griptape/drivers/web_scraper/trafilatura_web_scraper_driver.py

```
def extract_page(self, page: str) -> TextArtifact:
    trafilatura = import_optional_dependency("trafilatura")
    use_config = trafilatura.settings.use_config

    config = use_config()

    extracted_page = trafilatura.extract(
        page,
        include_links=self.include_links,
        output_format="json",
        config=config,
    )

    if not extracted_page:
        raise Exception("cant extract page")

    text = json.loads(extracted_page).get("text")

    return TextArtifact(text)
```

fetch_url(url)

Source Code in griptape/drivers/web_scraper/trafilatura_web_scraper_driver.py

```
def fetch_url(self, url: str) -> str:
    trafilatura = import_optional_dependency("trafilatura")
    use_config = trafilatura.settings.use_config

    config = use_config()
    page = trafilatura.fetch_url(url, no_ssl=self.no_ssl)

    # This disables signal, so that trafilatura can work on any thread:
    # More info: https://trafilatura.readthedocs.io/usage-python.html#disabling-signal
    config.set("DEFAULT", "EXTRACTION_TIMEOUT", "0")

    # Disable error logging in trafilatura as it sometimes logs errors from lxml,␣
```

(continues on next page)

(continued from previous page)

```
↪even though
    # the end result of page parsing is successful.
    logging.getLogger("trafilatura").setLevel(logging.FATAL)

    if page is None:
        raise Exception("cant access URL")

    return page
```

VoyageAiEmbeddingDriver

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.VoyageAiEmbeddingDriver.model" title="model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.VoyageAiEmbeddingDriver.model)"><code class="language-python">	*str*	VoyageAI embedding model name. Defaults to *voyage-large-2*.
<a href="./#griptape.drivers.VoyageAiEmbeddingDriver.api_key" title="api_key = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.VoyageAiEmbeddingDriver.api_key)"><code class="language-python">	*Optional[str]*	API key to pass directly. Defaults to *VOYAGE_API_KEY* environment variable.

continues on next page

Table 129 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.VoyageAiEmbeddingDriver.tokenizer" title="tokenizer = field(default=Factory(lambda self: VoyageAiTokenizer(model=self.model, api_key=self.api_key), takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.drivers.VoyageAiEmbeddingDriver.tokenizer)"><code class="language-python">	<a href="../tokenizers#griptape.tokenizers.VoyageAiTokenizer" title="VoyageAiTokenizer (griptape.tokenizers.VoyageAiTokenizer)"><code class="language-python">VoyageAiTokenizer</code>	Optionally provide custom *VoyageAiTokenizer*.
<a href="./#griptape.drivers.VoyageAiEmbeddingDriver.client" title="client() (griptape.drivers.VoyageAiEmbeddingDriver.client)"><code class="language-python">	*Any*	Optionally provide custom VoyageAI *Client*.
<a href="./#griptape.drivers.VoyageAiEmbeddingDriver.input_type" title="input_type = field(default='document', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.VoyageAiEmbeddingDriver.input_type)"><code class="language-python">	*str*	VoyageAI input type. Defaults to *document*.

Source Code in griptape/drivers/embedding/voyageai_embedding_driver.py

```
@define
class VoyageAiEmbeddingDriver(BaseEmbeddingDriver):
    """VoyageAI Embedding Driver.
```

(continues on next page)

(continued from previous page)

```
    Attributes:
        model: VoyageAI embedding model name. Defaults to `voyage-large-2`.
        api_key: API key to pass directly. Defaults to `VOYAGE_API_KEY` environment␣
↪variable.
        tokenizer: Optionally provide custom `VoyageAiTokenizer`.
        client: Optionally provide custom VoyageAI `Client`.
        input_type: VoyageAI input type. Defaults to `document`.
    """

    DEFAULT_MODEL = "voyage-large-2"

    model: str = field(default=DEFAULT_MODEL, kw_only=True, metadata={"serializable":␣
↪True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    tokenizer: VoyageAiTokenizer = field(
        default=Factory(lambda self: VoyageAiTokenizer(model=self.model, api_key=self.
↪api_key), takes_self=True),
        kw_only=True,
    )
    input_type: str = field(default="document", kw_only=True, metadata={"serializable
↪": True})
    _client: Optional[Client] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> Any:
        return import_optional_dependency("voyageai").Client(api_key=self.api_key)

    def try_embed_artifact(self, artifact: TextArtifact | ImageArtifact, **kwargs) ->␣
↪list[float]:
        if isinstance(artifact, TextArtifact):
            return self.try_embed_chunk(artifact.value, **kwargs)
        pil_image = import_optional_dependency("PIL.Image")

        return self.client.multimodal_embed([[pil_image.open(BytesIO(artifact.
↪value))]], model=self.model).embeddings[0]

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        return self.client.embed([chunk], model=self.model, input_type=self.input_
↪type).embeddings[0]
```

- `DEFAULT_MODEL = 'voyage-large-2'` class-attribute instance-attribute
- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `input_type = field(default='document', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable':`

`True})` class-attribute instance-attribute

- `tokenizer = field(default=Factory(lambda self: VoyageAiTokenizer(model=self.model, api_key=self.api_key), takes_self=True), kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/voyageai_embedding_driver.py

```
@lazy_property()
def client(self) -> Any:
    return import_optional_dependency("voyageai").Client(api_key=self.api_key)
```

try_embed_artifact(artifact, **kwargs)

Source Code in griptape/drivers/embedding/voyageai_embedding_driver.py

```
def try_embed_artifact(self, artifact: TextArtifact | ImageArtifact, **kwargs) ->␣
↪list[float]:
    if isinstance(artifact, TextArtifact):
        return self.try_embed_chunk(artifact.value, **kwargs)
    pil_image = import_optional_dependency("PIL.Image")

    return self.client.multimodal_embed([[pil_image.open(BytesIO(artifact.value))]],␣
↪model=self.model).embeddings[0]
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/voyageai_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    return self.client.embed([chunk], model=self.model, input_type=self.input_type).
↪embeddings[0]
```

WebhookEventListenerDriver

Bases: *BaseEventListenerDriver* “)

Source Code in griptape/drivers/event_listener/webhook_event_listener_driver.py

```
@define
class WebhookEventListenerDriver(BaseEventListenerDriver):
    webhook_url: str = field(kw_only=True)
    headers: Optional[dict] = field(default=None, kw_only=True)

    def try_publish_event_payload(self, event_payload: dict) -> None:
        response = requests.post(url=self.webhook_url, json=event_payload,␣
↪headers=self.headers)
        response.raise_for_status()

    def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None:
        response = requests.post(url=self.webhook_url, json=event_payload_batch,␣
↪headers=self.headers)
        response.raise_for_status()
```

- `headers = field(default=None, kw_only=True)` class-attribute instance-attribute
- `webhook_url = field(kw_only=True)` class-attribute instance-attribute

try_publish_event_payload(event_payload)

Source Code in griptape/drivers/event_listener/webhook_event_listener_driver.py

```
def try_publish_event_payload(self, event_payload: dict) -> None:
    response = requests.post(url=self.webhook_url, json=event_payload, headers=self.
↪headers)
    response.raise_for_status()
```

try_publish_event_payload_batch(event_payload_batch)

Source Code in griptape/drivers/event_listener/webhook_event_listener_driver.py

```
def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None:
    response = requests.post(url=self.webhook_url, json=event_payload_batch,␣
↪headers=self.headers)
    response.raise_for_status()
```

assistant

- `__all__ = ['BaseAssistantDriver']` module-attribute

Bases:

ABC

Source Code in griptape/drivers/assistant/base_assistant_driver.py

```
@define
class BaseAssistantDriver(ABC):
    """Base class for AssistantDrivers."""

    def run(self, *args: BaseArtifact) -> TextArtifact:
        return self.try_run(*args)

    @abstractmethod
    def try_run(self, *args: BaseArtifact) -> TextArtifact: ...
```

run(*args)

Source Code in griptape/drivers/assistant/base_assistant_driver.py

```
def run(self, *args: BaseArtifact) -> TextArtifact:
    return self.try_run(*args)
```

try_run(*args)abstractmethod

Source Code in griptape/drivers/assistant/base_assistant_driver.py

```
@abstractmethod
def try_run(self, *args: BaseArtifact) -> TextArtifact: ...
```

base_assistant_driver

Bases:

ABC

Source Code in griptape/drivers/assistant/base_assistant_driver.py

```
@define
class BaseAssistantDriver(ABC):
    """Base class for AssistantDrivers."""

    def run(self, *args: BaseArtifact) -> TextArtifact:
        return self.try_run(*args)

    @abstractmethod
    def try_run(self, *args: BaseArtifact) -> TextArtifact: ...
```

run(*args)

Source Code in griptape/drivers/assistant/base_assistant_driver.py

```
def run(self, *args: BaseArtifact) -> TextArtifact:
    return self.try_run(*args)
```

try_run(*args)abstractmethod

Source Code in griptape/drivers/assistant/base_assistant_driver.py

```
@abstractmethod
def try_run(self, *args: BaseArtifact) -> TextArtifact: ...
```

griptape_cloud

- `__all__ = ['GriptapeCloudAssistantDriver']` module-attribute

Bases: *BaseAssistantDriver* “)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
@define
class GriptapeCloudAssistantDriver(BaseAssistantDriver):
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]))
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        kw_only=True,
    )
    input: Optional[str] = field(default=None, kw_only=True)
    assistant_id: str = field(kw_only=True)
    thread_id: Optional[str] = field(default=None, kw_only=True)
```

(continues on next page)

(continued from previous page)

```
    thread_alias: Optional[str] = field(default=None, kw_only=True)
    ruleset_ids: Optional[list[str]] = field(default=None, kw_only=True)
    additional_ruleset_ids: list[str] = field(factory=list, kw_only=True)
    knowledge_base_ids: Optional[list[str]] = field(default=None, kw_only=True)
    additional_knowledge_base_ids: list[str] = field(factory=list, kw_only=True)
    structure_ids: Optional[list[str]] = field(default=None, kw_only=True)
    additional_structure_ids: list[str] = field(factory=list, kw_only=True)
    tool_ids: Optional[list[str]] = field(default=None, kw_only=True)
    additional_tool_ids: list[str] = field(factory=list, kw_only=True)
    stream: bool = field(default=False, kw_only=True)
    poll_interval: int = field(default=1, kw_only=True)
    max_attempts: int = field(default=20, kw_only=True)
    auto_create_thread: bool = field(default=True, kw_only=True)

    def try_run(self, *args: BaseArtifact) -> TextArtifact:
        if self.thread_id is None and self.auto_create_thread:
            self._create_or_find_thread(self.thread_alias)
        assistant_run_id = self._create_run(*args)
        run_result = self._get_run_result(assistant_run_id)

        run_result.meta.update(
            {"assistant_id": self.assistant_id, "assistant_run_id": assistant_run_id,
↪"thread_id": self.thread_id}
        )

        return run_result

    def _create_or_find_thread(self, thread_alias: Optional[str] = None) -> None:
        if thread_alias is None:
            self.thread_id = self._create_thread()
        else:
            thread = self._find_thread_by_alias(thread_alias)

            if thread is None:
                self.thread_id = self._create_thread(thread_alias)
            else:
                self.thread_id = thread["thread_id"]

    def _create_thread(self, thread_alias: Optional[str] = None) -> str:
        url = griptape_cloud_url(self.base_url, "api/threads")

        body = {"name": uuid.uuid4().hex}
        if thread_alias is not None:
            body["alias"] = thread_alias

        response = requests.post(url, json=body, headers=self.headers)
        response.raise_for_status()
        return response.json()["thread_id"]

    def _create_run(self, *args: BaseArtifact) -> str:
        url = griptape_cloud_url(self.base_url, f"api/assistants/{self.assistant_id}/
↪runs")
```

(continues on next page)

(continued from previous page)

```
        response = requests.post(
            url,
            json={
                "args": [arg.value for arg in args],
                "stream": self.stream,
                "thread_id": self.thread_id,
                "input": self.input,
                **({"ruleset_ids": self.ruleset_ids} if self.ruleset_ids is not None␣
↪else {}),
                "additional_ruleset_ids": self.additional_ruleset_ids,
                **({"knowledge_base_ids": self.knowledge_base_ids} if self.knowledge_
↪base_ids is not None else {}),
                "additional_knowledge_base_ids": self.additional_knowledge_base_ids,
                **({"structure_ids": self.structure_ids} if self.structure_ids is not␣
↪None else {}),
                "additional_structure_ids": self.additional_structure_ids,
                **({"tool_ids": self.tool_ids} if self.tool_ids is not None else {}),
                "additional_tool_ids": self.additional_tool_ids,
            },
            headers=self.headers,
        )
        response.raise_for_status()
        return response.json()["assistant_run_id"]

    def _get_run_result(self, assistant_run_id: str) -> TextArtifact:
        events = self._get_run_events(assistant_run_id)
        output = None

        for event in events:
            if event["origin"] == "ASSISTANT":
                event_payload = event["payload"]
                try:
                    EventBus.publish_event(BaseEvent.from_dict(event_payload))
                except ValueError as e:
                    logger.warning("Failed to deserialize event: %s", e)
                if event["type"] == "FinishStructureRunEvent":
                    output = TextArtifact.from_dict(event_payload["output_task_output
↪"])

        if output is None:
            raise ValueError("Output not found.")

        return output

    def _get_run_events(self, assistant_run_id: str) -> Iterator[dict]:
        url = griptape_cloud_url(self.base_url, f"api/assistant-runs/{assistant_run_
↪id}/events/stream")
        with requests.get(url, headers=self.headers, stream=True) as response:
            response.raise_for_status()
            for line in response.iter_lines():
                if line:
```

(continues on next page)

(continued from previous page)

```
                    decoded_line = line.decode("utf-8")
                    if decoded_line.startswith("data:"):
                        yield json.loads(decoded_line.removeprefix("data:").strip())

    def _find_thread_by_alias(self, thread_alias: str) -> Optional[dict]:
        url = griptape_cloud_url(self.base_url, "api/threads")
        response = requests.get(url, params={"alias": thread_alias}, headers=self.
↪headers)
        response.raise_for_status()

        threads = response.json()["threads"]

        return next((thread for thread in threads if thread["alias"] == thread_alias),
↪ None)
```

- `additional_knowledge_base_ids = field(factory=list, kw_only=True)` class-attribute instance-attribute
- `additional_ruleset_ids = field(factory=list, kw_only=True)` class-attribute instance-attribute
- `additional_structure_ids = field(factory=list, kw_only=True)` class-attribute instance-attribute
- `additional_tool_ids = field(factory=list, kw_only=True)` class-attribute instance-attribute
- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']))` class-attribute instance-attribute
- `assistant_id = field(kw_only=True)` class-attribute instance-attribute
- `auto_create_thread = field(default=True, kw_only=True)` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')))` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `input = field(default=None, kw_only=True)` class-attribute instance-attribute
- `knowledge_base_ids = field(default=None, kw_only=True)` class-attribute instance-attribute
- `max_attempts = field(default=20, kw_only=True)` class-attribute instance-attribute
- `poll_interval = field(default=1, kw_only=True)` class-attribute instance-attribute
- `ruleset_ids = field(default=None, kw_only=True)` class-attribute instance-attribute
- `stream = field(default=False, kw_only=True)` class-attribute instance-attribute
- `structure_ids = field(default=None, kw_only=True)` class-attribute instance-attribute
- `thread_alias = field(default=None, kw_only=True)` class-attribute instance-attribute
- `thread_id = field(default=None, kw_only=True)` class-attribute instance-attribute
- `tool_ids = field(default=None, kw_only=True)` class-attribute instance-attribute

_create_or_find_thread(thread_alias=None)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def _create_or_find_thread(self, thread_alias: Optional[str] = None) -> None:
    if thread_alias is None:
        self.thread_id = self._create_thread()
    else:
        thread = self._find_thread_by_alias(thread_alias)

        if thread is None:
            self.thread_id = self._create_thread(thread_alias)
        else:
            self.thread_id = thread["thread_id"]
```

_create_run(*args)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def _create_run(self, *args: BaseArtifact) -> str:
    url = griptape_cloud_url(self.base_url, f"api/assistants/{self.assistant_id}/runs
↪")

    response = requests.post(
        url,
        json={
            "args": [arg.value for arg in args],
            "stream": self.stream,
            "thread_id": self.thread_id,
            "input": self.input,
            **({"ruleset_ids": self.ruleset_ids} if self.ruleset_ids is not None else
↪{}),
            "additional_ruleset_ids": self.additional_ruleset_ids,
            **({"knowledge_base_ids": self.knowledge_base_ids} if self.knowledge_base_
↪ids is not None else {}),
            "additional_knowledge_base_ids": self.additional_knowledge_base_ids,
            **({"structure_ids": self.structure_ids} if self.structure_ids is not␣
↪None else {}),
            "additional_structure_ids": self.additional_structure_ids,
            **({"tool_ids": self.tool_ids} if self.tool_ids is not None else {}),
            "additional_tool_ids": self.additional_tool_ids,
        },
        headers=self.headers,
    )
    response.raise_for_status()
    return response.json()["assistant_run_id"]
```

_create_thread(thread_alias=None)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def _create_thread(self, thread_alias: Optional[str] = None) -> str:
    url = griptape_cloud_url(self.base_url, "api/threads")
```

(continues on next page)

(continued from previous page)

```
    body = {"name": uuid.uuid4().hex}
    if thread_alias is not None:
        body["alias"] = thread_alias

    response = requests.post(url, json=body, headers=self.headers)
    response.raise_for_status()
    return response.json()["thread_id"]
```

_find_thread_by_alias(thread_alias)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def _find_thread_by_alias(self, thread_alias: str) -> Optional[dict]:
    url = griptape_cloud_url(self.base_url, "api/threads")
    response = requests.get(url, params={"alias": thread_alias}, headers=self.headers)
    response.raise_for_status()

    threads = response.json()["threads"]

    return next((thread for thread in threads if thread["alias"] == thread_alias),␣
↪None)
```

_get_run_events(assistant_run_id)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def _get_run_events(self, assistant_run_id: str) -> Iterator[dict]:
    url = griptape_cloud_url(self.base_url, f"api/assistant-runs/{assistant_run_id}/
↪events/stream")
    with requests.get(url, headers=self.headers, stream=True) as response:
        response.raise_for_status()
        for line in response.iter_lines():
            if line:
                decoded_line = line.decode("utf-8")
                if decoded_line.startswith("data:"):
                    yield json.loads(decoded_line.removeprefix("data:").strip())
```

_get_run_result(assistant_run_id)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def _get_run_result(self, assistant_run_id: str) -> TextArtifact:
    events = self._get_run_events(assistant_run_id)
    output = None

    for event in events:
        if event["origin"] == "ASSISTANT":
            event_payload = event["payload"]
            try:
                EventBus.publish_event(BaseEvent.from_dict(event_payload))
```

(continues on next page)

(continued from previous page)

```
            except ValueError as e:
                logger.warning("Failed to deserialize event: %s", e)
            if event["type"] == "FinishStructureRunEvent":
                output = TextArtifact.from_dict(event_payload["output_task_output"])

    if output is None:
        raise ValueError("Output not found.")

    return output
```

try_run(*args)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def try_run(self, *args: BaseArtifact) -> TextArtifact:
    if self.thread_id is None and self.auto_create_thread:
        self._create_or_find_thread(self.thread_alias)
    assistant_run_id = self._create_run(*args)
    run_result = self._get_run_result(assistant_run_id)

    run_result.meta.update(
        {"assistant_id": self.assistant_id, "assistant_run_id": assistant_run_id,
↪"thread_id": self.thread_id}
    )

    return run_result
```

griptape_cloud_assistant_driver

- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *BaseAssistantDriver* “)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
@define
class GriptapeCloudAssistantDriver(BaseAssistantDriver):
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]))
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        kw_only=True,
    )
    input: Optional[str] = field(default=None, kw_only=True)
    assistant_id: str = field(kw_only=True)
    thread_id: Optional[str] = field(default=None, kw_only=True)
    thread_alias: Optional[str] = field(default=None, kw_only=True)
    ruleset_ids: Optional[list[str]] = field(default=None, kw_only=True)
```

(continues on next page)

(continued from previous page)

```
    additional_ruleset_ids: list[str] = field(factory=list, kw_only=True)
    knowledge_base_ids: Optional[list[str]] = field(default=None, kw_only=True)
    additional_knowledge_base_ids: list[str] = field(factory=list, kw_only=True)
    structure_ids: Optional[list[str]] = field(default=None, kw_only=True)
    additional_structure_ids: list[str] = field(factory=list, kw_only=True)
    tool_ids: Optional[list[str]] = field(default=None, kw_only=True)
    additional_tool_ids: list[str] = field(factory=list, kw_only=True)
    stream: bool = field(default=False, kw_only=True)
    poll_interval: int = field(default=1, kw_only=True)
    max_attempts: int = field(default=20, kw_only=True)
    auto_create_thread: bool = field(default=True, kw_only=True)

    def try_run(self, *args: BaseArtifact) -> TextArtifact:
        if self.thread_id is None and self.auto_create_thread:
            self._create_or_find_thread(self.thread_alias)
        assistant_run_id = self._create_run(*args)
        run_result = self._get_run_result(assistant_run_id)

        run_result.meta.update(
            {"assistant_id": self.assistant_id, "assistant_run_id": assistant_run_id,
↪"thread_id": self.thread_id}
        )

        return run_result

    def _create_or_find_thread(self, thread_alias: Optional[str] = None) -> None:
        if thread_alias is None:
            self.thread_id = self._create_thread()
        else:
            thread = self._find_thread_by_alias(thread_alias)

            if thread is None:
                self.thread_id = self._create_thread(thread_alias)
            else:
                self.thread_id = thread["thread_id"]

    def _create_thread(self, thread_alias: Optional[str] = None) -> str:
        url = griptape_cloud_url(self.base_url, "api/threads")

        body = {"name": uuid.uuid4().hex}
        if thread_alias is not None:
            body["alias"] = thread_alias

        response = requests.post(url, json=body, headers=self.headers)
        response.raise_for_status()
        return response.json()["thread_id"]

    def _create_run(self, *args: BaseArtifact) -> str:
        url = griptape_cloud_url(self.base_url, f"api/assistants/{self.assistant_id}/
↪runs")

        response = requests.post(
```

(continues on next page)

(continued from previous page)

```
            url,
            json={
                "args": [arg.value for arg in args],
                "stream": self.stream,
                "thread_id": self.thread_id,
                "input": self.input,
                **({"ruleset_ids": self.ruleset_ids} if self.ruleset_ids is not None␣
↪else {}),
                "additional_ruleset_ids": self.additional_ruleset_ids,
                **({"knowledge_base_ids": self.knowledge_base_ids} if self.knowledge_
↪base_ids is not None else {}),
                "additional_knowledge_base_ids": self.additional_knowledge_base_ids,
                **({"structure_ids": self.structure_ids} if self.structure_ids is not␣
↪None else {}),
                "additional_structure_ids": self.additional_structure_ids,
                **({"tool_ids": self.tool_ids} if self.tool_ids is not None else {}),
                "additional_tool_ids": self.additional_tool_ids,
            },
            headers=self.headers,
        )
        response.raise_for_status()
        return response.json()["assistant_run_id"]

    def _get_run_result(self, assistant_run_id: str) -> TextArtifact:
        events = self._get_run_events(assistant_run_id)
        output = None

        for event in events:
            if event["origin"] == "ASSISTANT":
                event_payload = event["payload"]
                try:
                    EventBus.publish_event(BaseEvent.from_dict(event_payload))
                except ValueError as e:
                    logger.warning("Failed to deserialize event: %s", e)
                if event["type"] == "FinishStructureRunEvent":
                    output = TextArtifact.from_dict(event_payload["output_task_output
↪"])

        if output is None:
            raise ValueError("Output not found.")

        return output

    def _get_run_events(self, assistant_run_id: str) -> Iterator[dict]:
        url = griptape_cloud_url(self.base_url, f"api/assistant-runs/{assistant_run_
↪id}/events/stream")
        with requests.get(url, headers=self.headers, stream=True) as response:
            response.raise_for_status()
            for line in response.iter_lines():
                if line:
                    decoded_line = line.decode("utf-8")
                    if decoded_line.startswith("data:"):
```

(continues on next page)

(continued from previous page)

```
                    yield json.loads(decoded_line.removeprefix("data:").strip())

    def _find_thread_by_alias(self, thread_alias: str) -> Optional[dict]:
        url = griptape_cloud_url(self.base_url, "api/threads")
        response = requests.get(url, params={"alias": thread_alias}, headers=self.
↪headers)
        response.raise_for_status()

        threads = response.json()["threads"]

        return next((thread for thread in threads if thread["alias"] == thread_alias),
↪ None)
```

- `additional_knowledge_base_ids = field(factory=list, kw_only=True)` class-attribute instance-attribute
- `additional_ruleset_ids = field(factory=list, kw_only=True)` class-attribute instance-attribute
- `additional_structure_ids = field(factory=list, kw_only=True)` class-attribute instance-attribute
- `additional_tool_ids = field(factory=list, kw_only=True)` class-attribute instance-attribute
- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']))` class-attribute instance-attribute
- `assistant_id = field(kw_only=True)` class-attribute instance-attribute
- `auto_create_thread = field(default=True, kw_only=True)` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')))` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `input = field(default=None, kw_only=True)` class-attribute instance-attribute
- `knowledge_base_ids = field(default=None, kw_only=True)` class-attribute instance-attribute
- `max_attempts = field(default=20, kw_only=True)` class-attribute instance-attribute
- `poll_interval = field(default=1, kw_only=True)` class-attribute instance-attribute
- `ruleset_ids = field(default=None, kw_only=True)` class-attribute instance-attribute
- `stream = field(default=False, kw_only=True)` class-attribute instance-attribute
- `structure_ids = field(default=None, kw_only=True)` class-attribute instance-attribute
- `thread_alias = field(default=None, kw_only=True)` class-attribute instance-attribute
- `thread_id = field(default=None, kw_only=True)` class-attribute instance-attribute
- `tool_ids = field(default=None, kw_only=True)` class-attribute instance-attribute

_create_or_find_thread(thread_alias=None)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def _create_or_find_thread(self, thread_alias: Optional[str] = None) -> None:
    if thread_alias is None:
        self.thread_id = self._create_thread()
    else:
        thread = self._find_thread_by_alias(thread_alias)

        if thread is None:
            self.thread_id = self._create_thread(thread_alias)
        else:
            self.thread_id = thread["thread_id"]
```

_create_run(*args)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def _create_run(self, *args: BaseArtifact) -> str:
    url = griptape_cloud_url(self.base_url, f"api/assistants/{self.assistant_id}/runs
↪")

    response = requests.post(
        url,
        json={
            "args": [arg.value for arg in args],
            "stream": self.stream,
            "thread_id": self.thread_id,
            "input": self.input,
            **({"ruleset_ids": self.ruleset_ids} if self.ruleset_ids is not None else
↪{}),
            "additional_ruleset_ids": self.additional_ruleset_ids,
            **({"knowledge_base_ids": self.knowledge_base_ids} if self.knowledge_base_
↪ids is not None else {}),
            "additional_knowledge_base_ids": self.additional_knowledge_base_ids,
            **({"structure_ids": self.structure_ids} if self.structure_ids is not␣
↪None else {}),
            "additional_structure_ids": self.additional_structure_ids,
            **({"tool_ids": self.tool_ids} if self.tool_ids is not None else {}),
            "additional_tool_ids": self.additional_tool_ids,
        },
        headers=self.headers,
    )
    response.raise_for_status()
    return response.json()["assistant_run_id"]
```

_create_thread(thread_alias=None)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def _create_thread(self, thread_alias: Optional[str] = None) -> str:
    url = griptape_cloud_url(self.base_url, "api/threads")

    body = {"name": uuid.uuid4().hex}
    if thread_alias is not None:
```

(continues on next page)

(continued from previous page)

```
        body["alias"] = thread_alias

    response = requests.post(url, json=body, headers=self.headers)
    response.raise_for_status()
    return response.json()["thread_id"]
```

_find_thread_by_alias(thread_alias)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def _find_thread_by_alias(self, thread_alias: str) -> Optional[dict]:
    url = griptape_cloud_url(self.base_url, "api/threads")
    response = requests.get(url, params={"alias": thread_alias}, headers=self.headers)
    response.raise_for_status()

    threads = response.json()["threads"]

    return next((thread for thread in threads if thread["alias"] == thread_alias),␣
↪None)
```

_get_run_events(assistant_run_id)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def _get_run_events(self, assistant_run_id: str) -> Iterator[dict]:
    url = griptape_cloud_url(self.base_url, f"api/assistant-runs/{assistant_run_id}/
↪events/stream")
    with requests.get(url, headers=self.headers, stream=True) as response:
        response.raise_for_status()
        for line in response.iter_lines():
            if line:
                decoded_line = line.decode("utf-8")
                if decoded_line.startswith("data:"):
                    yield json.loads(decoded_line.removeprefix("data:").strip())
```

_get_run_result(assistant_run_id)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def _get_run_result(self, assistant_run_id: str) -> TextArtifact:
    events = self._get_run_events(assistant_run_id)
    output = None

    for event in events:
        if event["origin"] == "ASSISTANT":
            event_payload = event["payload"]
            try:
                EventBus.publish_event(BaseEvent.from_dict(event_payload))
            except ValueError as e:
                logger.warning("Failed to deserialize event: %s", e)
            if event["type"] == "FinishStructureRunEvent":
```

(continues on next page)

(continued from previous page)

```
                output = TextArtifact.from_dict(event_payload["output_task_output"])

    if output is None:
        raise ValueError("Output not found.")

    return output
```

try_run(*args)

Source Code in griptape/drivers/assistant/griptape_cloud_assistant_driver.py

```
def try_run(self, *args: BaseArtifact) -> TextArtifact:
    if self.thread_id is None and self.auto_create_thread:
        self._create_or_find_thread(self.thread_alias)
    assistant_run_id = self._create_run(*args)
    run_result = self._get_run_result(assistant_run_id)

    run_result.meta.update(
        {"assistant_id": self.assistant_id, "assistant_run_id": assistant_run_id,
↪"thread_id": self.thread_id}
    )

    return run_result
```

openai

- `__all__ = ['OpenAiAssistantDriver']` module-attribute

Bases: *BaseAssistantDriver* “)

Source Code in griptape/drivers/assistant/openai_assistant_driver.py

```
@define
class OpenAiAssistantDriver(BaseAssistantDriver):
    class EventHandler(AssistantEventHandler):
        @override
        def on_text_delta(self, delta: TextDelta, snapshot: Text) -> None:
            if delta.value is not None:
                EventBus.publish_event(TextChunkEvent(token=delta.value))

        @override
        def on_tool_call_delta(self, delta: ToolCallDelta, snapshot: ToolCall) ->␣
↪None:
            if delta.type == "code_interpreter" and delta.code_interpreter is not␣
↪None:
                if delta.code_interpreter.input:
                    EventBus.publish_event(TextChunkEvent(token=delta.code_
↪interpreter.input))
                if delta.code_interpreter.outputs:
                    EventBus.publish_event(TextChunkEvent(token="\n\noutput >"))
                    for output in delta.code_interpreter.outputs:
                        if output.type == "logs" and output.logs:
```

(continues on next page)

(continued from previous page)

```
                            EventBus.publish_event(TextChunkEvent(token=output.logs))

    base_url: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    organization: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    thread_id: Optional[str] = field(default=None, kw_only=True)
    assistant_id: str = field(kw_only=True)
    event_handler: AssistantEventHandler = field(
        default=Factory(lambda: OpenAiAssistantDriver.EventHandler()), kw_only=True,␣
↪metadata={"serializable": False}
    )
    auto_create_thread: bool = field(default=True, kw_only=True)

    _client: Optional[openai.OpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.OpenAI:
        return openai.OpenAI(
            base_url=self.base_url,
            api_key=self.api_key,
            organization=self.organization,
        )

    def try_run(self, *args: BaseArtifact) -> TextArtifact:
        if self.thread_id is None:
            if self.auto_create_thread:
                thread_id = self.client.beta.threads.create().id
                self.thread_id = thread_id
            else:
                raise ValueError("Thread ID is required but not provided and auto_
↪create_thread is disabled.")
        else:
            thread_id = self.thread_id

        response = self._create_run(thread_id, *args)

        response.meta.update({"assistant_id": self.assistant_id, "thread_id": self.
↪thread_id})

        return response

    def _create_run(self, thread_id: str, *args: BaseArtifact) -> TextArtifact:
        content = "\n".join(arg.value for arg in args)
        message_id = self.client.beta.threads.messages.create(thread_id=thread_id,␣
↪role="user", content=content)
        with self.client.beta.threads.runs.stream(
            thread_id=thread_id,
```

(continues on next page)

(continued from previous page)

```
            assistant_id=self.assistant_id,
            event_handler=self.event_handler,
        ) as stream:
            stream.until_done()
            last_messages = stream.get_final_messages()

            message_contents = []
            for message in last_messages:
                message_contents.append(
                    "".join(content.text.value for content in message.content if␣
↪content.type == "TextContentBlock")
                )
            message_text = "\n".join(message_contents)

            response = TextArtifact(message_text)

            response.meta.update(
                {"assistant_id": self.assistant_id, "thread_id": self.thread_id,
↪"message_id": message_id}
            )
            return response
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `assistant_id = field(kw_only=True)` class-attribute instance-attribute
- `auto_create_thread = field(default=True, kw_only=True)` class-attribute instance-attribute
- `base_url = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `event_handler = field(default=Factory(lambda: OpenAiAssistantDriver.EventHandler()), kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `organization = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `thread_id = field(default=None, kw_only=True)` class-attribute instance-attribute

EventHandler

Bases:

AssistantEventHandler

Source Code in griptape/drivers/assistant/openai_assistant_driver.py

```
class EventHandler(AssistantEventHandler):
    @override
    def on_text_delta(self, delta: TextDelta, snapshot: Text) -> None:
        if delta.value is not None:
            EventBus.publish_event(TextChunkEvent(token=delta.value))
```

(continues on next page)

(continued from previous page)

```
    @override
    def on_tool_call_delta(self, delta: ToolCallDelta, snapshot: ToolCall) -> None:
        if delta.type == "code_interpreter" and delta.code_interpreter is not None:
            if delta.code_interpreter.input:
                EventBus.publish_event(TextChunkEvent(token=delta.code_interpreter.
↪input))
            if delta.code_interpreter.outputs:
                EventBus.publish_event(TextChunkEvent(token="\n\noutput >"))
                for output in delta.code_interpreter.outputs:
                    if output.type == "logs" and output.logs:
                        EventBus.publish_event(TextChunkEvent(token=output.logs))
```

_create_run(thread_id, *args)

Source Code in griptape/drivers/assistant/openai_assistant_driver.py

```
def _create_run(self, thread_id: str, *args: BaseArtifact) -> TextArtifact:
    content = "\n".join(arg.value for arg in args)
    message_id = self.client.beta.threads.messages.create(thread_id=thread_id, role=
↪"user", content=content)
    with self.client.beta.threads.runs.stream(
        thread_id=thread_id,
        assistant_id=self.assistant_id,
        event_handler=self.event_handler,
    ) as stream:
        stream.until_done()
        last_messages = stream.get_final_messages()

        message_contents = []
        for message in last_messages:
            message_contents.append(
                "".join(content.text.value for content in message.content if content.
↪type == "TextContentBlock")
            )
        message_text = "\n".join(message_contents)

        response = TextArtifact(message_text)

        response.meta.update(
            {"assistant_id": self.assistant_id, "thread_id": self.thread_id, "message_
↪id": message_id}
        )
        return response
```

client()

Source Code in griptape/drivers/assistant/openai_assistant_driver.py

```
@lazy_property()
def client(self) -> openai.OpenAI:
```

(continues on next page)

(continued from previous page)

```
    return openai.OpenAI(
        base_url=self.base_url,
        api_key=self.api_key,
        organization=self.organization,
    )
```

try_run(*args)

Source Code in griptape/drivers/assistant/openai_assistant_driver.py

```
def try_run(self, *args: BaseArtifact) -> TextArtifact:
    if self.thread_id is None:
        if self.auto_create_thread:
            thread_id = self.client.beta.threads.create().id
            self.thread_id = thread_id
        else:
            raise ValueError("Thread ID is required but not provided and auto_create_
↪thread is disabled.")
    else:
        thread_id = self.thread_id

    response = self._create_run(thread_id, *args)

    response.meta.update({"assistant_id": self.assistant_id, "thread_id": self.thread_
↪id})

    return response
```

openai_assistant_driver

Bases: *BaseAssistantDriver* “)

Source Code in griptape/drivers/assistant/openai_assistant_driver.py

```
@define
class OpenAiAssistantDriver(BaseAssistantDriver):
    class EventHandler(AssistantEventHandler):
        @override
        def on_text_delta(self, delta: TextDelta, snapshot: Text) -> None:
            if delta.value is not None:
                EventBus.publish_event(TextChunkEvent(token=delta.value))

        @override
        def on_tool_call_delta(self, delta: ToolCallDelta, snapshot: ToolCall) ->␣
↪None:
            if delta.type == "code_interpreter" and delta.code_interpreter is not␣
↪None:
                if delta.code_interpreter.input:
                    EventBus.publish_event(TextChunkEvent(token=delta.code_
↪interpreter.input))
                if delta.code_interpreter.outputs:
                    EventBus.publish_event(TextChunkEvent(token="\n\noutput >"))
```

(continues on next page)

(continued from previous page)

```
                    for output in delta.code_interpreter.outputs:
                        if output.type == "logs" and output.logs:
                            EventBus.publish_event(TextChunkEvent(token=output.logs))

    base_url: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    organization: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    thread_id: Optional[str] = field(default=None, kw_only=True)
    assistant_id: str = field(kw_only=True)
    event_handler: AssistantEventHandler = field(
        default=Factory(lambda: OpenAiAssistantDriver.EventHandler()), kw_only=True,␣
↪metadata={"serializable": False}
    )
    auto_create_thread: bool = field(default=True, kw_only=True)

    _client: Optional[openai.OpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.OpenAI:
        return openai.OpenAI(
            base_url=self.base_url,
            api_key=self.api_key,
            organization=self.organization,
        )

    def try_run(self, *args: BaseArtifact) -> TextArtifact:
        if self.thread_id is None:
            if self.auto_create_thread:
                thread_id = self.client.beta.threads.create().id
                self.thread_id = thread_id
            else:
                raise ValueError("Thread ID is required but not provided and auto_
↪create_thread is disabled.")
        else:
            thread_id = self.thread_id

        response = self._create_run(thread_id, *args)

        response.meta.update({"assistant_id": self.assistant_id, "thread_id": self.
↪thread_id})

        return response

    def _create_run(self, thread_id: str, *args: BaseArtifact) -> TextArtifact:
        content = "\n".join(arg.value for arg in args)
        message_id = self.client.beta.threads.messages.create(thread_id=thread_id,␣
↪role="user", content=content)
```

(continues on next page)

(continued from previous page)

```
        with self.client.beta.threads.runs.stream(
            thread_id=thread_id,
            assistant_id=self.assistant_id,
            event_handler=self.event_handler,
        ) as stream:
            stream.until_done()
            last_messages = stream.get_final_messages()

            message_contents = []
            for message in last_messages:
                message_contents.append(
                    "".join(content.text.value for content in message.content if␣
↪content.type == "TextContentBlock")
                )
            message_text = "\n".join(message_contents)

            response = TextArtifact(message_text)

            response.meta.update(
                {"assistant_id": self.assistant_id, "thread_id": self.thread_id,
↪"message_id": message_id}
            )
            return response
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `assistant_id = field(kw_only=True)` class-attribute instance-attribute
- `auto_create_thread = field(default=True, kw_only=True)` class-attribute instance-attribute
- `base_url = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `event_handler = field(default=Factory(lambda: OpenAiAssistantDriver.EventHandler()), kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `organization = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `thread_id = field(default=None, kw_only=True)` class-attribute instance-attribute

EventHandler

Bases:

AssistantEventHandler

Source Code in griptape/drivers/assistant/openai_assistant_driver.py

```
class EventHandler(AssistantEventHandler):
    @override
    def on_text_delta(self, delta: TextDelta, snapshot: Text) -> None:
```

(continues on next page)

(continued from previous page)

```
        if delta.value is not None:
            EventBus.publish_event(TextChunkEvent(token=delta.value))

    @override
    def on_tool_call_delta(self, delta: ToolCallDelta, snapshot: ToolCall) -> None:
        if delta.type == "code_interpreter" and delta.code_interpreter is not None:
            if delta.code_interpreter.input:
                EventBus.publish_event(TextChunkEvent(token=delta.code_interpreter.
↪input))
            if delta.code_interpreter.outputs:
                EventBus.publish_event(TextChunkEvent(token="\n\noutput >"))
                for output in delta.code_interpreter.outputs:
                    if output.type == "logs" and output.logs:
                        EventBus.publish_event(TextChunkEvent(token=output.logs))
```

_create_run(thread_id, *args)

Source Code in griptape/drivers/assistant/openai_assistant_driver.py

```
def _create_run(self, thread_id: str, *args: BaseArtifact) -> TextArtifact:
    content = "\n".join(arg.value for arg in args)
    message_id = self.client.beta.threads.messages.create(thread_id=thread_id, role=
↪"user", content=content)
    with self.client.beta.threads.runs.stream(
        thread_id=thread_id,
        assistant_id=self.assistant_id,
        event_handler=self.event_handler,
    ) as stream:
        stream.until_done()
        last_messages = stream.get_final_messages()

        message_contents = []
        for message in last_messages:
            message_contents.append(
                "".join(content.text.value for content in message.content if content.
↪type == "TextContentBlock")
            )
        message_text = "\n".join(message_contents)

        response = TextArtifact(message_text)

        response.meta.update(
            {"assistant_id": self.assistant_id, "thread_id": self.thread_id, "message_
↪id": message_id}
        )
        return response
```

client()

Source Code in griptape/drivers/assistant/openai_assistant_driver.py

```
@lazy_property()
def client(self) -> openai.OpenAI:
    return openai.OpenAI(
        base_url=self.base_url,
        api_key=self.api_key,
        organization=self.organization,
    )
```

try_run(*args)

Source Code in griptape/drivers/assistant/openai_assistant_driver.py

```
def try_run(self, *args: BaseArtifact) -> TextArtifact:
    if self.thread_id is None:
        if self.auto_create_thread:
            thread_id = self.client.beta.threads.create().id
            self.thread_id = thread_id
        else:
            raise ValueError("Thread ID is required but not provided and auto_create_
↪thread is disabled.")
    else:
        thread_id = self.thread_id

    response = self._create_run(thread_id, *args)

    response.meta.update({"assistant_id": self.assistant_id, "thread_id": self.thread_
↪id})

    return response
```

audio_transcription

- `__all__ = ['BaseAudioTranscriptionDriver']` module-attribute

Bases: *SerializableMixin* “) , *ExponentialBackoffMixin* ”) , ABC

Source Code in griptape/drivers/audio_transcription/base_audio_transcription_driver.py

```
@define
class BaseAudioTranscriptionDriver(SerializableMixin, ExponentialBackoffMixin, ABC):
    model: str = field(kw_only=True, metadata={"serializable": True})

    def before_run(self) -> None:
        EventBus.publish_event(StartAudioTranscriptionEvent())

    def after_run(self) -> None:
        EventBus.publish_event(FinishAudioTranscriptionEvent())

    def run(self, audio: AudioArtifact, prompts: Optional[list[str]] = None) ->␣
↪TextArtifact:
```

(continues on next page)

(continued from previous page)

```
        for attempt in self.retrying():
            with attempt:
                self.before_run()
                result = self.try_run(audio, prompts)
                self.after_run()

                return result

        raise Exception("Failed to run audio transcription")

    @abstractmethod
    def try_run(self, audio: AudioArtifact, prompts: Optional[list[str]] = None) ->␣
↪TextArtifact: ...
```

- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

after_run()

Source Code in griptape/drivers/audio_transcription/base_audio_transcription_driver.py

```
def after_run(self) -> None:
    EventBus.publish_event(FinishAudioTranscriptionEvent())
```

before_run()

Source Code in griptape/drivers/audio_transcription/base_audio_transcription_driver.py

```
def before_run(self) -> None:
    EventBus.publish_event(StartAudioTranscriptionEvent())
```

run(audio, prompts=None)

Source Code in griptape/drivers/audio_transcription/base_audio_transcription_driver.py

```
def run(self, audio: AudioArtifact, prompts: Optional[list[str]] = None) ->␣
↪TextArtifact:
    for attempt in self.retrying():
        with attempt:
            self.before_run()
            result = self.try_run(audio, prompts)
            self.after_run()

            return result

    raise Exception("Failed to run audio transcription")
```

try_run(audio, prompts=None)abstractmethod

Source Code in griptape/drivers/audio_transcription/base_audio_transcription_driver.py

```
@abstractmethod
def try_run(self, audio: AudioArtifact, prompts: Optional[list[str]] = None) ->␣
↪TextArtifact: ...
```

base_audio_transcription_driver

Bases: *SerializableMixin* “) , *ExponentialBackoffMixin* ”) , ABC

Source Code in griptape/drivers/audio_transcription/base_audio_transcription_driver.py

```
@define
class BaseAudioTranscriptionDriver(SerializableMixin, ExponentialBackoffMixin, ABC):
    model: str = field(kw_only=True, metadata={"serializable": True})

    def before_run(self) -> None:
        EventBus.publish_event(StartAudioTranscriptionEvent())

    def after_run(self) -> None:
        EventBus.publish_event(FinishAudioTranscriptionEvent())

    def run(self, audio: AudioArtifact, prompts: Optional[list[str]] = None) ->␣
↪TextArtifact:
        for attempt in self.retrying():
            with attempt:
                self.before_run()
                result = self.try_run(audio, prompts)
                self.after_run()

                return result

        raise Exception("Failed to run audio transcription")

    @abstractmethod
    def try_run(self, audio: AudioArtifact, prompts: Optional[list[str]] = None) ->␣
↪TextArtifact: ...
```

- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

after_run()

Source Code in griptape/drivers/audio_transcription/base_audio_transcription_driver.py

```
def after_run(self) -> None:
    EventBus.publish_event(FinishAudioTranscriptionEvent())
```

before_run()

Source Code in griptape/drivers/audio_transcription/base_audio_transcription_driver.py

```
def before_run(self) -> None:
    EventBus.publish_event(StartAudioTranscriptionEvent())
```

run(audio, prompts=None)

Source Code in griptape/drivers/audio_transcription/base_audio_transcription_driver.py

```
def run(self, audio: AudioArtifact, prompts: Optional[list[str]] = None) ->␣
↪TextArtifact:
    for attempt in self.retrying():
        with attempt:
            self.before_run()
            result = self.try_run(audio, prompts)
            self.after_run()

            return result

    raise Exception("Failed to run audio transcription")
```

try_run(audio, prompts=None)abstractmethod

Source Code in griptape/drivers/audio_transcription/base_audio_transcription_driver.py

```
@abstractmethod
def try_run(self, audio: AudioArtifact, prompts: Optional[list[str]] = None) ->␣
↪TextArtifact: ...
```

dummy

- `__all__ = ['DummyAudioTranscriptionDriver']` module-attribute

Bases: *BaseAudioTranscriptionDriver* “)

Source Code in griptape/drivers/audio_transcription/dummy_audio_transcription_driver.py

```
@define
class DummyAudioTranscriptionDriver(BaseAudioTranscriptionDriver):
    model: str = field(init=False)

    def try_run(self, audio: AudioArtifact, prompts: Optional[list] = None) ->␣
↪TextArtifact:
        raise DummyError(__class__.__name__, "try_transcription")
```

- `model = field(init=False)` class-attribute instance-attribute

try_run(audio, prompts=None)

Source Code in griptape/drivers/audio_transcription/dummy_audio_transcription_driver.py

```
def try_run(self, audio: AudioArtifact, prompts: Optional[list] = None) ->␣
↪TextArtifact:
    raise DummyError(__class__.__name__, "try_transcription")
```

dummy_audio_transcription_driver

Bases: *BaseAudioTranscriptionDriver* “)

Source Code in griptape/drivers/audio_transcription/dummy_audio_transcription_driver.py

```
@define
class DummyAudioTranscriptionDriver(BaseAudioTranscriptionDriver):
    model: str = field(init=False)

    def try_run(self, audio: AudioArtifact, prompts: Optional[list] = None) ->␣
↪TextArtifact:
        raise DummyError(__class__.__name__, "try_transcription")
```

- `model = field(init=False)` class-attribute instance-attribute

try_run(audio, prompts=None)

Source Code in griptape/drivers/audio_transcription/dummy_audio_transcription_driver.py

```
def try_run(self, audio: AudioArtifact, prompts: Optional[list] = None) ->␣
↪TextArtifact:
    raise DummyError(__class__.__name__, "try_transcription")
```

openai

- `__all__ = ['OpenAiAudioTranscriptionDriver']` module-attribute

Bases: *BaseAudioTranscriptionDriver* “)

Source Code in griptape/drivers/audio_transcription/openai_audio_transcription_driver.py

```
@define
class OpenAiAudioTranscriptionDriver(BaseAudioTranscriptionDriver):
    api_type: Optional[str] = field(default=openai.api_type, kw_only=True)
    api_version: Optional[str] = field(default=openai.api_version, kw_only=True,␣
↪metadata={"serializable": True})
    base_url: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    organization: Optional[str] = field(default=openai.organization, kw_only=True,␣
↪metadata={"serializable": True})
    _client: Optional[openai.OpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.OpenAI:
        return openai.OpenAI(api_key=self.api_key, base_url=self.base_url,␣
```

(continues on next page)

(continued from previous page)

```
↪organization=self.organization)

    def try_run(self, audio: AudioArtifact, prompts: Optional[list[str]] = None) ->␣
↪TextArtifact:
        additional_params = {}

        if prompts is not None:
            additional_params["prompt"] = ", ".join(prompts)

        transcription = self.client.audio.transcriptions.create(
            # Even though were not actually providing a file to the client, the API␣
↪still requires that we send a file
            # name. We set the file name to use the same format as the audio file so␣
↪that the API can reject
            # it if the format is unsupported.
            model=self.model,
            file=(f"a.{audio.format}", io.BytesIO(audio.value)),
            response_format="json",
            **additional_params,
        )

        return TextArtifact(value=transcription.text)
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `api_type = field(default=openai.api_type, kw_only=True)` class-attribute instance-attribute
- `api_version = field(default=openai.api_version, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `base_url = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `organization = field(default=openai.organization, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/audio_transcription/openai_audio_transcription_driver.py

```
@lazy_property()
def client(self) -> openai.OpenAI:
    return openai.OpenAI(api_key=self.api_key, base_url=self.base_url,␣
↪organization=self.organization)
```

try_run(audio, prompts=None)

Source Code in griptape/drivers/audio_transcription/openai_audio_transcription_driver.py

```
def try_run(self, audio: AudioArtifact, prompts: Optional[list[str]] = None) ->␣
↪TextArtifact:
    additional_params = {}

    if prompts is not None:
        additional_params["prompt"] = ", ".join(prompts)

    transcription = self.client.audio.transcriptions.create(
        # Even though were not actually providing a file to the client, the API still␣
↪requires that we send a file
        # name. We set the file name to use the same format as the audio file so that␣
↪the API can reject
        # it if the format is unsupported.
        model=self.model,
        file=(f"a.{audio.format}", io.BytesIO(audio.value)),
        response_format="json",
        **additional_params,
    )

    return TextArtifact(value=transcription.text)
```

openai_audio_transcription_driver

Bases: *BaseAudioTranscriptionDriver* “)

Source Code in griptape/drivers/audio_transcription/openai_audio_transcription_driver.py

```
@define
class OpenAiAudioTranscriptionDriver(BaseAudioTranscriptionDriver):
    api_type: Optional[str] = field(default=openai.api_type, kw_only=True)
    api_version: Optional[str] = field(default=openai.api_version, kw_only=True,␣
↪metadata={"serializable": True})
    base_url: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    organization: Optional[str] = field(default=openai.organization, kw_only=True,␣
↪metadata={"serializable": True})
    _client: Optional[openai.OpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.OpenAI:
        return openai.OpenAI(api_key=self.api_key, base_url=self.base_url,␣
↪organization=self.organization)

    def try_run(self, audio: AudioArtifact, prompts: Optional[list[str]] = None) ->␣
↪TextArtifact:
        additional_params = {}

        if prompts is not None:
            additional_params["prompt"] = ", ".join(prompts)
```

(continues on next page)

(continued from previous page)

```
        transcription = self.client.audio.transcriptions.create(
            # Even though were not actually providing a file to the client, the API␣
↪still requires that we send a file
            # name. We set the file name to use the same format as the audio file so␣
↪that the API can reject
            # it if the format is unsupported.
            model=self.model,
            file=(f"a.{audio.format}", io.BytesIO(audio.value)),
            response_format="json",
            **additional_params,
        )

        return TextArtifact(value=transcription.text)
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `api_type = field(default=openai.api_type, kw_only=True)` class-attribute instance-attribute
- `api_version = field(default=openai.api_version, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `base_url = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `organization = field(default=openai.organization, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/audio_transcription/openai_audio_transcription_driver.py

```
@lazy_property()
def client(self) -> openai.OpenAI:
    return openai.OpenAI(api_key=self.api_key, base_url=self.base_url,␣
↪organization=self.organization)
```

try_run(audio, prompts=None)

Source Code in griptape/drivers/audio_transcription/openai_audio_transcription_driver.py

```
def try_run(self, audio: AudioArtifact, prompts: Optional[list[str]] = None) ->␣
↪TextArtifact:
    additional_params = {}

    if prompts is not None:
        additional_params["prompt"] = ", ".join(prompts)

    transcription = self.client.audio.transcriptions.create(
        # Even though were not actually providing a file to the client, the API still␣
```

(continues on next page)

(continued from previous page)

```
↪requires that we send a file
        # name. We set the file name to use the same format as the audio file so that␣
↪the API can reject
        # it if the format is unsupported.
        model=self.model,
        file=(f"a.{audio.format}", io.BytesIO(audio.value)),
        response_format="json",
        **additional_params,
    )

    return TextArtifact(value=transcription.text)
```

embedding

- `__all__ = ['BaseEmbeddingDriver']` module-attribute

Bases: *SerializableMixin* “) , *ExponentialBackoffMixin* ”) , ABC

Attributes

Name	Type	Description
<a href="./#griptape.drivers.embedding.BaseEmbeddingDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.BaseEmbeddingDriver.model)"><code class="language-python">	*str*	The name of the model to use.
<a href="./#griptape.drivers.embedding.BaseEmbeddingDriver.tokenizer" title="tokenizer = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.drivers.embedding.BaseEmbeddingDriver.tokenizer)"><code class="language-python">	<a href="../../tokenizers#griptape.tokenizers.BaseTokenizer" title="BaseTokenizer (griptape.tokenizers.BaseTokenizer)"><code class="language-python">Optional[BaseTokenizer]</c	An instance of *BaseTokenizer* to use when calculating tokens.

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
@define
class BaseEmbeddingDriver(SerializableMixin, ExponentialBackoffMixin, ABC):
    """Base Embedding Driver.

    Attributes:
        model: The name of the model to use.
        tokenizer: An instance of `BaseTokenizer` to use when calculating tokens.
    """

    model: str = field(kw_only=True, metadata={"serializable": True})
    tokenizer: Optional[BaseTokenizer] = field(default=None, kw_only=True)
    chunker: Optional[BaseChunker] = field(init=False)

    def __attrs_post_init__(self) -> None:
        self.chunker = TextChunker(tokenizer=self.tokenizer) if self.tokenizer else␣
↪None

    def embed_text_artifact(
        self, artifact: TextArtifact, *, vector_operation: VectorOperation | None =␣
↪None
    ) -> list[float]:
        warnings.warn(
            "`BaseEmbeddingDriver.embed_text_artifact` is deprecated and will be␣
↪removed in a future release. `BaseEmbeddingDriver.embed` is a drop-in replacement.",
            DeprecationWarning,
            stacklevel=2,
        )
        return self.embed(artifact, vector_operation=vector_operation)

    def embed_string(self, string: str, *, vector_operation: VectorOperation | None =␣
↪None) -> list[float]:
        warnings.warn(
            "`BaseEmbeddingDriver.embed_string` is deprecated and will be removed in␣
↪a future release. `BaseEmbeddingDriver.embed` is a drop-in replacement.",
            DeprecationWarning,
            stacklevel=2,
        )
        return self.embed(string, vector_operation=vector_operation)

    def embed(
        self, value: str | TextArtifact | ImageArtifact, *, vector_operation:␣
↪VectorOperation | None = None
    ) -> list[float]:
        for attempt in self.retrying():
            with attempt:
                if isinstance(value, str):
                    if (
                        self.tokenizer is not None
                        and self.tokenizer.count_tokens(value) > self.tokenizer.max_
↪input_tokens
                    ):
                        return self._embed_long_string(value, vector_operation=vector_
↪operation)
```

(continues on next page)

(continued from previous page)

```
                    return self.try_embed_chunk(value, vector_operation=vector_
↪operation)
                if isinstance(value, TextArtifact):
                    return self.embed(value.to_text(), vector_operation=vector_
↪operation)
                if isinstance(value, ImageArtifact):
                    return self.try_embed_artifact(value, vector_operation=vector_
↪operation)
        raise RuntimeError("Failed to embed string.")

    def try_embed_artifact(
        self, artifact: TextArtifact | ImageArtifact, *, vector_operation:␣
↪VectorOperation | None = None
    ) -> list[float]:
        # TODO: Mark as abstract method for griptape 2.0
        if isinstance(artifact, TextArtifact):
            return self.try_embed_chunk(artifact.value, vector_operation=vector_
↪operation)
        raise ValueError(f"{self.__class__.__name__} does not support embedding␣
↪images.")

    @abstractmethod
    def try_embed_chunk(self, chunk: str, *, vector_operation: VectorOperation | None␣
↪= None) -> list[float]:
        # TODO: Remove for griptape 2.0, subclasses should implement `try_embed_
↪artifact` instead
        ...

    def _embed_long_string(self, string: str, *, vector_operation: VectorOperation |␣
↪None = None) -> list[float]:
        """Embeds a string that is too long to embed in one go.

        Adapted from: https://github.com/openai/openai-cookbook/blob/
↪683e5f5a71bc7a1b0e5b7a35e087f53cc55fceea/examples/Embedding_long_inputs.ipynb
        """
        chunks = self.chunker.chunk(string)  # pyright:␣
↪ignore[reportOptionalMemberAccess] In practice this is never None

        embedding_chunks = []
        length_chunks = []
        for chunk in chunks:
            embedding_chunks.append(self.embed(chunk.value, vector_operation=vector_
↪operation))
            length_chunks.append(len(chunk))

        # generate weighted averages
        embedding_chunks = np.average(embedding_chunks, axis=0, weights=length_chunks)

        # normalize length to 1
        embedding_chunks = embedding_chunks / np.linalg.norm(embedding_chunks)

        return embedding_chunks.tolist()
```

- `chunker = field(init=False)` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=None, kw_only=True)` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
def __attrs_post_init__(self) -> None:
    self.chunker = TextChunker(tokenizer=self.tokenizer) if self.tokenizer else None
```

_embed_long_string(string, *, vector_operation=None)

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
def _embed_long_string(self, string: str, *, vector_operation: VectorOperation | None␣
↪= None) -> list[float]:
    """Embeds a string that is too long to embed in one go.

    Adapted from: https://github.com/openai/openai-cookbook/blob/
↪683e5f5a71bc7a1b0e5b7a35e087f53cc55fceea/examples/Embedding_long_inputs.ipynb
    """
    chunks = self.chunker.chunk(string)  # pyright:␣
↪ignore[reportOptionalMemberAccess] In practice this is never None

    embedding_chunks = []
    length_chunks = []
    for chunk in chunks:
        embedding_chunks.append(self.embed(chunk.value, vector_operation=vector_
↪operation))
        length_chunks.append(len(chunk))

    # generate weighted averages
    embedding_chunks = np.average(embedding_chunks, axis=0, weights=length_chunks)

    # normalize length to 1
    embedding_chunks = embedding_chunks / np.linalg.norm(embedding_chunks)

    return embedding_chunks.tolist()
```

embed(value, *, vector_operation=None)

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
def embed(
    self, value: str | TextArtifact | ImageArtifact, *, vector_operation:␣
↪VectorOperation | None = None
) -> list[float]:
    for attempt in self.retrying():
        with attempt:
            if isinstance(value, str):
```

(continues on next page)

(continued from previous page)

```
                if (
                    self.tokenizer is not None
                    and self.tokenizer.count_tokens(value) > self.tokenizer.max_input_
↪tokens
                ):
                    return self._embed_long_string(value, vector_operation=vector_
↪operation)
                return self.try_embed_chunk(value, vector_operation=vector_operation)
            if isinstance(value, TextArtifact):
                return self.embed(value.to_text(), vector_operation=vector_operation)
            if isinstance(value, ImageArtifact):
                return self.try_embed_artifact(value, vector_operation=vector_
↪operation)
        raise RuntimeError("Failed to embed string.")
```

embed_string(string, *, vector_operation=None)

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
def embed_string(self, string: str, *, vector_operation: VectorOperation | None =␣
↪None) -> list[float]:
    warnings.warn(
        "`BaseEmbeddingDriver.embed_string` is deprecated and will be removed in a␣
↪future release. `BaseEmbeddingDriver.embed` is a drop-in replacement.",
        DeprecationWarning,
        stacklevel=2,
    )
    return self.embed(string, vector_operation=vector_operation)
```

embed_text_artifact(artifact, *, vector_operation=None)

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
def embed_text_artifact(
    self, artifact: TextArtifact, *, vector_operation: VectorOperation | None = None
) -> list[float]:
    warnings.warn(
        "`BaseEmbeddingDriver.embed_text_artifact` is deprecated and will be removed␣
↪in a future release. `BaseEmbeddingDriver.embed` is a drop-in replacement.",
        DeprecationWarning,
        stacklevel=2,
    )
    return self.embed(artifact, vector_operation=vector_operation)
```

try_embed_artifact(artifact, *, vector_operation=None)

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
def try_embed_artifact(
    self, artifact: TextArtifact | ImageArtifact, *, vector_operation:␣
↪VectorOperation | None = None
```

(continues on next page)

(continued from previous page)

```
) -> list[float]:
    # TODO: Mark as abstract method for griptape 2.0
    if isinstance(artifact, TextArtifact):
        return self.try_embed_chunk(artifact.value, vector_operation=vector_operation)
    raise ValueError(f"{self.__class__.__name__} does not support embedding images.")
```

try_embed_chunk(chunk, *, vector_operation=None)abstractmethod

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
@abstractmethod
def try_embed_chunk(self, chunk: str, *, vector_operation: VectorOperation | None =␣
↪None) -> list[float]:
    # TODO: Remove for griptape 2.0, subclasses should implement `try_embed_artifact`␣
↪instead
    ...
```

amazon_bedrock

- `__all__ = ['AmazonBedrockCohereEmbeddingDriver', 'AmazonBedrockTitanEmbeddingDriver']` module-attribute

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<code class="language-python">	*str*	Embedding model name. Defaults to DEFAULT_MODEL.

continues on next page

Table 131 – continued from previous page

Name	Type	Description
<code class="language-python">	*str*	Defaults to *search_query*. Prepends special tokens to differentiate each type from one another: *search_document* when you encode documents for embeddings that you store in a vector database. *search_query* when querying your vector DB to find relevant documents.
<code class="language-python">	*Session*	Optionally provide custom *boto3.Session*.

continues on next page

Table 131 – continued from previous page

Name	Type	Description
<code class="language-python">	<code class="language-python">BaseTokenizer</code>	Optionally provide custom *BedrockCohereTokenizer*.
<a href="amazon_bedrock.mdx#griptape.drivers.embedding.amazon_bedrock.AmazonBedrockCohereEmbeddingDriver.client" title="client() (griptape.drivers.embedding.amazon_bedrock.AmazonBedrockCohereEmbeddingDriver.client)"><code class="language-python">	*BedrockRuntimeClient*	Optionally provide custom *bedrock-runtime* client.

Source Code in griptape/drivers/embedding/amazon_bedrock_cohere_embedding_driver.py

```
@define
class AmazonBedrockCohereEmbeddingDriver(BaseEmbeddingDriver):
    """Amazon Bedrock Cohere Embedding Driver.

    Attributes:
        model: Embedding model name. Defaults to DEFAULT_MODEL.
        input_type: Defaults to `search_query`. Prepends special tokens to␣
↪differentiate each type from one another:
            `search_document` when you encode documents for embeddings that you store␣
```

(continues on next page)

(continued from previous page)

```
↪in a vector database.
            `search_query` when querying your vector DB to find relevant documents.
        session: Optionally provide custom `boto3.Session`.
        tokenizer: Optionally provide custom `BedrockCohereTokenizer`.
        client: Optionally provide custom `bedrock-runtime` client.
    """

    DEFAULT_MODEL = "cohere.embed-english-v3"

    model: str = field(default=DEFAULT_MODEL, kw_only=True)
    input_type: str = field(default="search_query", kw_only=True)
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: AmazonBedrockTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
    _client: Optional[BedrockRuntimeClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> BedrockRuntimeClient:
        return self.session.client("bedrock-runtime")

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        payload = {"input_type": self.input_type, "texts": [chunk]}

        response = self.client.invoke_model(
            body=json.dumps(payload),
            modelId=self.model,
            accept="*/*",
            contentType="application/json",
        )
        response_body = json.loads(response.get("body").read())

        return response_body.get("embeddings")[0]
```

- `DEFAULT_MODEL = 'cohere.embed-english-v3'` class-attribute instance-attribute
- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `input_type = field(default='search_query', kw_only=True)` class-attribute instance-attribute
- `model = field(default=DEFAULT_MODEL, kw_only=True)` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: AmazonBedrockTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/amazon_bedrock_cohere_embedding_driver.py

```
@lazy_property()
def client(self) -> BedrockRuntimeClient:
    return self.session.client("bedrock-runtime")
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/amazon_bedrock_cohere_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    payload = {"input_type": self.input_type, "texts": [chunk]}

    response = self.client.invoke_model(
        body=json.dumps(payload),
        modelId=self.model,
        accept="*/*",
        contentType="application/json",
    )
    response_body = json.loads(response.get("body").read())

    return response_body.get("embeddings")[0]
```

AmazonBedrockTitanEmbeddingDriver

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<code class="language-python">	*str*	Embedding model name. Defaults to DEFAULT_MODEL.
<code class="language-python">	<code class="language-python">BaseTokenizer</code>	Optionally provide custom *BedrockTitanTokenizer*.

continues on next page

Table 132 – continued from previous page

Name	Type	Description
<a href="amazon_bedrock.mdx#griptape.drivers.embedding.amazon_bedrock.AmazonBedrockTitanEmbeddingDriver.session" title="session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True) class-attribute instance-attribute (griptape.drivers.embedding.amazon_bedrock.AmazonBedrockTitanEmbeddingDriver.session)"><code class="language-python">	*Session*	Optionally provide custom *boto3.Session*.
<a href="amazon_bedrock.mdx#griptape.drivers.embedding.amazon_bedrock.AmazonBedrockTitanEmbeddingDriver.client" title="client() (griptape.drivers.embedding.amazon_bedrock.AmazonBedrockTitanEmbeddingDriver.client)"><code class="language-python">	*BedrockRuntimeClient*	Optionally provide custom *bedrock-runtime* client.

Source Code in griptape/drivers/embedding/amazon_bedrock_titan_embedding_driver.py

```
@define
class AmazonBedrockTitanEmbeddingDriver(BaseEmbeddingDriver):
    """Amazon Bedrock Titan Embedding Driver.

    Attributes:
        model: Embedding model name. Defaults to DEFAULT_MODEL.
        tokenizer: Optionally provide custom `BedrockTitanTokenizer`.
        session: Optionally provide custom `boto3.Session`.
        client: Optionally provide custom `bedrock-runtime` client.
    """
```

(continues on next page)

(continued from previous page)

```
    DEFAULT_MODEL = "amazon.titan-embed-text-v1"

    model: str = field(default=DEFAULT_MODEL, kw_only=True, metadata={"serializable":␣
↪True})
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: AmazonBedrockTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
    _client: Optional[BedrockRuntimeClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> BedrockRuntimeClient:
        return self.session.client("bedrock-runtime")

    def try_embed_artifact(self, artifact: TextArtifact | ImageArtifact, **kwargs) ->␣
↪list[float]:
        if isinstance(artifact, TextArtifact):
            return self.try_embed_chunk(artifact.value)
        return self._invoke_model({"inputImage": base64.b64encode(artifact.value).
↪decode()})["embedding"]

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        return self._invoke_model(
            {
                "inputText": chunk,
            }
        )["embedding"]

    def _invoke_model(self, payload: dict) -> dict[str, Any]:
        response = self.client.invoke_model(
            body=json.dumps(payload),
            modelId=self.model,
            accept="application/json",
            contentType="application/json",
        )
        return json.loads(response.get("body").read())
```

- `DEFAULT_MODEL = 'amazon.titan-embed-text-v1'` class-attribute instance-attribute
- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: AmazonBedrockTokenizer(model=self.`

`model), takes_self=True), kw_only=True)` class-attribute instance-attribute

_invoke_model(payload)

Source Code in griptape/drivers/embedding/amazon_bedrock_titan_embedding_driver.py

```
def _invoke_model(self, payload: dict) -> dict[str, Any]:
    response = self.client.invoke_model(
        body=json.dumps(payload),
        modelId=self.model,
        accept="application/json",
        contentType="application/json",
    )
    return json.loads(response.get("body").read())
```

client()

Source Code in griptape/drivers/embedding/amazon_bedrock_titan_embedding_driver.py

```
@lazy_property()
def client(self) -> BedrockRuntimeClient:
    return self.session.client("bedrock-runtime")
```

try_embed_artifact(artifact, **kwargs)

Source Code in griptape/drivers/embedding/amazon_bedrock_titan_embedding_driver.py

```
def try_embed_artifact(self, artifact: TextArtifact | ImageArtifact, **kwargs) ->␣
↪list[float]:
    if isinstance(artifact, TextArtifact):
        return self.try_embed_chunk(artifact.value)
    return self._invoke_model({"inputImage": base64.b64encode(artifact.value).
↪decode()})["embedding"]
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/amazon_bedrock_titan_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    return self._invoke_model(
        {
            "inputText": chunk,
        }
    )["embedding"]
```

amazon_bedrock_cohere_embedding_driver

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="amazon_bedrock_cohere_embedding_driver.mdx#griptape.drivers.embedding.amazon_bedrock_cohere_embedding_driver.AmazonBedrockCohereEmbeddingDriver.model" title="model = field(default=DEFAULT_MODEL, kw_only=True) class-attribute instance-attribute (griptape.drivers.embedding.amazon_bedrock_cohere_embedding_driver.AmazonBedrockCohereEmbeddingDriver.model)"><code class="language-python">	*str*	Embedding model name. Defaults to DEFAULT_MODEL.

continues on next page

Table 133 – continued from previous page

Name	Type	Description
<code class="language-python">	*str*	Defaults to *search_query*. Prepends special tokens to differentiate each type from one another: *search_document* when you encode documents for embeddings that you store in a vector database. *search_query* when querying your vector DB to find relevant documents.

continues on next page

Table 133 – continued from previous page

Name	Type	Description
<code class="language-python">	*Session*	Optionally provide custom *boto3.Session*.

continues on next page

Table 133 – continued from previous page

Name	Type	Description
<code class="language-python">	<code class="language-python">BaseTokenizer</code>	Optionally provide custom *BedrockCohereTokenizer*.
<a href="amazon_bedrock_cohere_embedding_driver.mdx#griptape.drivers.embedding.amazon_bedrock_cohere_embedding_driver.AmazonBedrockCohereEmbeddingDriver.client" title="client() (griptape.drivers.embedding.amazon_bedrock_cohere_embedding_driver.AmazonBedrockCohereEmbeddingDriver.client)"><code class="language-python">	*BedrockRuntimeClient*	Optionally provide custom *bedrock-runtime* client.

Source Code in griptape/drivers/embedding/amazon_bedrock_cohere_embedding_driver.py

```
@define
class AmazonBedrockCohereEmbeddingDriver(BaseEmbeddingDriver):
    """Amazon Bedrock Cohere Embedding Driver.

    Attributes:
        model: Embedding model name. Defaults to DEFAULT_MODEL.
        input_type: Defaults to `search_query`. Prepends special tokens to␣
↪differentiate each type from one another:
            `search_document` when you encode documents for embeddings that you store␣
↪in a vector database.
            `search_query` when querying your vector DB to find relevant documents.
        session: Optionally provide custom `boto3.Session`.
        tokenizer: Optionally provide custom `BedrockCohereTokenizer`.
        client: Optionally provide custom `bedrock-runtime` client.
    """

    DEFAULT_MODEL = "cohere.embed-english-v3"

    model: str = field(default=DEFAULT_MODEL, kw_only=True)
    input_type: str = field(default="search_query", kw_only=True)
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: AmazonBedrockTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
    _client: Optional[BedrockRuntimeClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> BedrockRuntimeClient:
        return self.session.client("bedrock-runtime")

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        payload = {"input_type": self.input_type, "texts": [chunk]}

        response = self.client.invoke_model(
            body=json.dumps(payload),
            modelId=self.model,
            accept="*/*",
            contentType="application/json",
        )
        response_body = json.loads(response.get("body").read())

        return response_body.get("embeddings")[0]
```

- `DEFAULT_MODEL = 'cohere.embed-english-v3'` class-attribute instance-attribute
- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute

- `input_type = field(default='search_query', kw_only=True)` class-attribute instance-attribute
- `model = field(default=DEFAULT_MODEL, kw_only=True)` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: AmazonBedrockTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/amazon_bedrock_cohere_embedding_driver.py

```
@lazy_property()
def client(self) -> BedrockRuntimeClient:
    return self.session.client("bedrock-runtime")
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/amazon_bedrock_cohere_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    payload = {"input_type": self.input_type, "texts": [chunk]}

    response = self.client.invoke_model(
        body=json.dumps(payload),
        modelId=self.model,
        accept="*/*",
        contentType="application/json",
    )
    response_body = json.loads(response.get("body").read())

    return response_body.get("embeddings")[0]
```

amazon_bedrock_titan_embedding_driver

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<code class="language-python">	*str*	Embedding model name. Defaults to DEFAULT_MODEL.

continues on next page

Table 134 – continued from previous page

Name	Type	Description
<code class="language-python">	<code class="language-python">BaseTokenizer</code>	Optionally provide custom *BedrockTitanTokenizer*.

continues on next page

Table 134 – continued from previous page

Name	Type	Description
<a href="amazon_bedrock_titan_embedding_driver.mdx#griptape.drivers.embedding.amazon_bedrock_titan_embedding_driver.AmazonBedrockTitanEmbeddingDriver.session" title="session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True) class-attribute instance-attribute (griptape.drivers.embedding.amazon_bedrock_titan_embedding_driver.AmazonBedrockTitanEmbeddingDriver.session)"><code class="language-python">	*Session*	Optionally provide custom *boto3.Session*.
<a href="amazon_bedrock_titan_embedding_driver.mdx#griptape.drivers.embedding.amazon_bedrock_titan_embedding_driver.AmazonBedrockTitanEmbeddingDriver.client" title="client() (griptape.drivers.embedding.amazon_bedrock_titan_embedding_driver.AmazonBedrockTitanEmbeddingDriver.client)"><code class="language-python">	*BedrockRuntimeClient*	Optionally provide custom *bedrock-runtime* client.

Source Code in griptape/drivers/embedding/amazon_bedrock_titan_embedding_driver.py

```
@define
class AmazonBedrockTitanEmbeddingDriver(BaseEmbeddingDriver):
```

(continues on next page)

(continued from previous page)

```
    """Amazon Bedrock Titan Embedding Driver.

    Attributes:
        model: Embedding model name. Defaults to DEFAULT_MODEL.
        tokenizer: Optionally provide custom `BedrockTitanTokenizer`.
        session: Optionally provide custom `boto3.Session`.
        client: Optionally provide custom `bedrock-runtime` client.
    """

    DEFAULT_MODEL = "amazon.titan-embed-text-v1"

    model: str = field(default=DEFAULT_MODEL, kw_only=True, metadata={"serializable":␣
↪True})
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: AmazonBedrockTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
    _client: Optional[BedrockRuntimeClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> BedrockRuntimeClient:
        return self.session.client("bedrock-runtime")

    def try_embed_artifact(self, artifact: TextArtifact | ImageArtifact, **kwargs) ->␣
↪list[float]:
        if isinstance(artifact, TextArtifact):
            return self.try_embed_chunk(artifact.value)
        return self._invoke_model({"inputImage": base64.b64encode(artifact.value).
↪decode()})["embedding"]

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        return self._invoke_model(
            {
                "inputText": chunk,
            }
        )["embedding"]

    def _invoke_model(self, payload: dict) -> dict[str, Any]:
        response = self.client.invoke_model(
            body=json.dumps(payload),
            modelId=self.model,
            accept="application/json",
            contentType="application/json",
        )
        return json.loads(response.get("body").read())
```

- `DEFAULT_MODEL = 'amazon.titan-embed-text-v1'` class-attribute instance-attribute

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: AmazonBedrockTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute

_invoke_model(payload)

Source Code in griptape/drivers/embedding/amazon_bedrock_titan_embedding_driver.py

```
def _invoke_model(self, payload: dict) -> dict[str, Any]:
    response = self.client.invoke_model(
        body=json.dumps(payload),
        modelId=self.model,
        accept="application/json",
        contentType="application/json",
    )
    return json.loads(response.get("body").read())
```

client()

Source Code in griptape/drivers/embedding/amazon_bedrock_titan_embedding_driver.py

```
@lazy_property()
def client(self) -> BedrockRuntimeClient:
    return self.session.client("bedrock-runtime")
```

try_embed_artifact(artifact, **kwargs)

Source Code in griptape/drivers/embedding/amazon_bedrock_titan_embedding_driver.py

```
def try_embed_artifact(self, artifact: TextArtifact | ImageArtifact, **kwargs) ->␣
↪list[float]:
    if isinstance(artifact, TextArtifact):
        return self.try_embed_chunk(artifact.value)
    return self._invoke_model({"inputImage": base64.b64encode(artifact.value).
↪decode()})["embedding"]
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/amazon_bedrock_titan_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    return self._invoke_model(
        {
            "inputText": chunk,
        }
    )["embedding"]
```

amazon_sagemaker_jumpstart

- `__all__ = ['AmazonSageMakerJumpstartEmbeddingDriver']` module-attribute

Bases: *BaseEmbeddingDriver* “)

Source Code in griptape/drivers/embedding/amazon_sagemaker_jumpstart_embedding_driver.py

```
@define
class AmazonSageMakerJumpstartEmbeddingDriver(BaseEmbeddingDriver):
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    endpoint: str = field(kw_only=True, metadata={"serializable": True})
    custom_attributes: str = field(default="accept_eula=true", kw_only=True, metadata=
↪{"serializable": True})
    inference_component_name: Optional[str] = field(default=None, kw_only=True,␣
↪metadata={"serializable": True})
    _client: Optional[SageMakerRuntimeClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> SageMakerRuntimeClient:
        return self.session.client("sagemaker-runtime")

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        payload = {"text_inputs": chunk, "mode": "embedding"}

        endpoint_response = self.client.invoke_endpoint(
            EndpointName=self.endpoint,
            ContentType="application/json",
            Body=json.dumps(payload).encode("utf-8"),
            CustomAttributes=self.custom_attributes,
            **(
                {"InferenceComponentName": self.inference_component_name}
                if self.inference_component_name is not None
                else {}
            ),
        )

        response = json.loads(endpoint_response.get("Body").read().decode("utf-8"))

        if "embedding" in response:
            embedding = response["embedding"]

            if embedding:
                if isinstance(embedding[0], list):
                    return embedding[0]
                return embedding
            raise ValueError("model response is empty")
        raise ValueError("invalid response from model")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `custom_attributes = field(default='accept_eula=true', kw_only=True,`

`metadata={'serializable': True})` class-attribute instance-attribute

- `endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `inference_component_name = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/amazon_sagemaker_jumpstart_embedding_driver.py

```
@lazy_property()
def client(self) -> SageMakerRuntimeClient:
    return self.session.client("sagemaker-runtime")
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/amazon_sagemaker_jumpstart_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    payload = {"text_inputs": chunk, "mode": "embedding"}

    endpoint_response = self.client.invoke_endpoint(
        EndpointName=self.endpoint,
        ContentType="application/json",
        Body=json.dumps(payload).encode("utf-8"),
        CustomAttributes=self.custom_attributes,
        **(
            {"InferenceComponentName": self.inference_component_name}
            if self.inference_component_name is not None
            else {}
        ),
    )

    response = json.loads(endpoint_response.get("Body").read().decode("utf-8"))

    if "embedding" in response:
        embedding = response["embedding"]

        if embedding:
            if isinstance(embedding[0], list):
                return embedding[0]
            return embedding
        raise ValueError("model response is empty")
    raise ValueError("invalid response from model")
```

amazon_sagemaker_jumpstart_embedding_driver

Bases: *BaseEmbeddingDriver* “)

Source Code in griptape/drivers/embedding/amazon_sagemaker_jumpstart_embedding_driver.py

```
@define
class AmazonSageMakerJumpstartEmbeddingDriver(BaseEmbeddingDriver):
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    endpoint: str = field(kw_only=True, metadata={"serializable": True})
    custom_attributes: str = field(default="accept_eula=true", kw_only=True, metadata=
↪{"serializable": True})
    inference_component_name: Optional[str] = field(default=None, kw_only=True,␣
↪metadata={"serializable": True})
    _client: Optional[SageMakerRuntimeClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> SageMakerRuntimeClient:
        return self.session.client("sagemaker-runtime")

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        payload = {"text_inputs": chunk, "mode": "embedding"}

        endpoint_response = self.client.invoke_endpoint(
            EndpointName=self.endpoint,
            ContentType="application/json",
            Body=json.dumps(payload).encode("utf-8"),
            CustomAttributes=self.custom_attributes,
            **(
                {"InferenceComponentName": self.inference_component_name}
                if self.inference_component_name is not None
                else {}
            ),
        )

        response = json.loads(endpoint_response.get("Body").read().decode("utf-8"))

        if "embedding" in response:
            embedding = response["embedding"]

            if embedding:
                if isinstance(embedding[0], list):
                    return embedding[0]
                return embedding
            raise ValueError("model response is empty")
        raise ValueError("invalid response from model")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `custom_attributes = field(default='accept_eula=true', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

- `endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `inference_component_name = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/amazon_sagemaker_jumpstart_embedding_driver.py

```
@lazy_property()
def client(self) -> SageMakerRuntimeClient:
    return self.session.client("sagemaker-runtime")
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/amazon_sagemaker_jumpstart_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    payload = {"text_inputs": chunk, "mode": "embedding"}

    endpoint_response = self.client.invoke_endpoint(
        EndpointName=self.endpoint,
        ContentType="application/json",
        Body=json.dumps(payload).encode("utf-8"),
        CustomAttributes=self.custom_attributes,
        **(
            {"InferenceComponentName": self.inference_component_name}
            if self.inference_component_name is not None
            else {}
        ),
    )

    response = json.loads(endpoint_response.get("Body").read().decode("utf-8"))

    if "embedding" in response:
        embedding = response["embedding"]

        if embedding:
            if isinstance(embedding[0], list):
                return embedding[0]
            return embedding
        raise ValueError("model response is empty")
    raise ValueError("invalid response from model")
```

azure_openai_embedding_driver

Bases: *OpenAiEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="azure_openai_embedding_driver.mdx#griptape.drivers.embedding.azure_openai_embedding_driver.AzureOpenAiEmbeddingDriver.azure_deployment" title="azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.azure_openai_embedding_driver.AzureOpenAiEmbeddingDriver.azure_deployment)"><code class="language-python">	*str*	An optional Azure OpenAi deployment id. Defaults to the model name.
<a href="azure_openai_embedding_driver.mdx#griptape.drivers.embedding.azure_openai_embedding_driver.AzureOpenAiEmbeddingDriver.azure_endpoint" title="azure_endpoint = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.azure_openai_embedding_driver.AzureOpenAiEmbeddingDriver.azure_endpoint)"><code class="language-python">	*str*	An Azure OpenAi endpoint.

continues on next page

Table 135 – continued from previous page

Name	Type	Description
<a href="azure_openai_embedding_driver.mdx#griptape.drivers.embedding.azure_openai_embedding_driver.AzureOpenAiEmbeddingDriver.azure_ad_token" title="azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.embedding.azure_openai_embedding_driver.AzureOpenAiEmbeddingDriver.azure_ad_token)"><code class="language-python">	*Optional[str]*	An optional Azure Active Directory token.
<a href="azure_openai_embedding_driver.mdx#griptape.drivers.embedding.azure_openai_embedding_driver.AzureOpenAiEmbeddingDriver.azure_ad_token_provider" title="azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.embedding.azure_openai_embedding_driver.AzureOpenAiEmbeddingDriver.azure_ad_token_provider)"><code class="language-python">	*Optional[Callable[[], str]]*	An optional Azure Active Directory token provider.

continues on next page

Table 135 – continued from previous page

Name	Type	Description
<a href="azure_openai_embedding_driver.mdx#griptape.drivers.embedding.azure_openai_embedding_driver.AzureOpenAiEmbeddingDriver.api_version" title="api_version = field(default='2024-10-21', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.azure_openai_embedding_driver.AzureOpenAiEmbeddingDriver.api_version)"><code class="language-python">	*str*	An Azure OpenAi API version.
<a href="azure_openai_embedding_driver.mdx#griptape.drivers.embedding.azure_openai_embedding_driver.AzureOpenAiEmbeddingDriver.tokenizer" title="tokenizer = field(default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.drivers.embedding.azure_openai_embedding_driver.AzureOpenAiEmbeddingDriver.tokenizer)"><code class="language-python">	<a href="../../tokenizers#griptape.tokenizers.OpenAiTokenizer" title="OpenAiTokenizer (griptape.tokenizers.OpenAiTokenizer)"><code class="language-python">OpenAiTokenizer</code></	An *OpenAiTokenizer*.

continues on next page

Table 135 – continued from previous page

Name	Type	Description
<a href="azure_openai_embedding_driver.mdx#griptape.drivers.embedding.azure_openai_embedding_driver.AzureOpenAiEmbeddingDriver.client" title="client() (griptape.drivers.embedding.azure_openai_embedding_driver.AzureOpenAiEmbeddingDriver.client)"><code class="language-python">	*AzureOpenAI*	An *openai.AzureOpenAI* client.

Source Code in griptape/drivers/embedding/azure_openai_embedding_driver.py

```
@define
class AzureOpenAiEmbeddingDriver(OpenAiEmbeddingDriver):
    """Azure OpenAi Embedding Driver.

    Attributes:
        azure_deployment: An optional Azure OpenAi deployment id. Defaults to the␣
↪model name.
        azure_endpoint: An Azure OpenAi endpoint.
        azure_ad_token: An optional Azure Active Directory token.
        azure_ad_token_provider: An optional Azure Active Directory token provider.
        api_version: An Azure OpenAi API version.
        tokenizer: An `OpenAiTokenizer`.
        client: An `openai.AzureOpenAI` client.
    """

    azure_deployment: str = field(
        kw_only=True,
        default=Factory(lambda self: self.model, takes_self=True),
        metadata={"serializable": True},
    )
    azure_endpoint: str = field(kw_only=True, metadata={"serializable": True})
    azure_ad_token: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": False})
    azure_ad_token_provider: Optional[Callable[[], str]] = field(
        kw_only=True,
        default=None,
        metadata={"serializable": False},
    )
    api_version: str = field(default="2024-10-21", kw_only=True, metadata={
↪"serializable": True})
    tokenizer: OpenAiTokenizer = field(
        default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_
```

(continues on next page)

(continued from previous page)

```
↪self=True),
        kw_only=True,
    )
    _client: Optional[openai.AzureOpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.AzureOpenAI:
        return openai.AzureOpenAI(
            organization=self.organization,
            api_key=self.api_key,
            api_version=self.api_version,
            azure_endpoint=self.azure_endpoint,
            azure_deployment=self.azure_deployment,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_version = field(default='2024-10-21', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True})` class-attribute instance-attribute
- `azure_endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/azure_openai_embedding_driver.py

```
@lazy_property()
def client(self) -> openai.AzureOpenAI:
    return openai.AzureOpenAI(
        organization=self.organization,
        api_key=self.api_key,
        api_version=self.api_version,
        azure_endpoint=self.azure_endpoint,
        azure_deployment=self.azure_deployment,
        azure_ad_token=self.azure_ad_token,
        azure_ad_token_provider=self.azure_ad_token_provider,
    )
```

base_embedding_driver

- `VectorOperation = Literal['query', 'upsert']` module-attribute

Bases: *SerializableMixin* “) , *ExponentialBackoffMixin* ”) , ABC

Attributes

Name	Type	Description
<a href="base_embedding_driver.mdx#griptape.drivers.embedding.base_embedding_driver.BaseEmbeddingDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.base_embedding_driver.BaseEmbeddingDriver.model)"><code class="language-python">	*str*	The name of the model to use.
<a href="base_embedding_driver.mdx#griptape.drivers.embedding.base_embedding_driver.BaseEmbeddingDriver.tokenizer" title="tokenizer = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.drivers.embedding.base_embedding_driver.BaseEmbeddingDriver.tokenizer)"><code class="language-python">	<a href="../../tokenizers#griptape.tokenizers.BaseTokenizer" title="BaseTokenizer (griptape.tokenizers.BaseTokenizer)"><code class="language-python">Optional[BaseTokenizer]</c	An instance of *BaseTokenizer* to use when calculating tokens.

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
@define
class BaseEmbeddingDriver(SerializableMixin, ExponentialBackoffMixin, ABC):
    """Base Embedding Driver.

    Attributes:
```

(continues on next page)

(continued from previous page)

```
        model: The name of the model to use.
        tokenizer: An instance of `BaseTokenizer` to use when calculating tokens.
    """

    model: str = field(kw_only=True, metadata={"serializable": True})
    tokenizer: Optional[BaseTokenizer] = field(default=None, kw_only=True)
    chunker: Optional[BaseChunker] = field(init=False)

    def __attrs_post_init__(self) -> None:
        self.chunker = TextChunker(tokenizer=self.tokenizer) if self.tokenizer else␣
↪None

    def embed_text_artifact(
        self, artifact: TextArtifact, *, vector_operation: VectorOperation | None =␣
↪None
    ) -> list[float]:
        warnings.warn(
            "`BaseEmbeddingDriver.embed_text_artifact` is deprecated and will be␣
↪removed in a future release. `BaseEmbeddingDriver.embed` is a drop-in replacement.",
            DeprecationWarning,
            stacklevel=2,
        )
        return self.embed(artifact, vector_operation=vector_operation)

    def embed_string(self, string: str, *, vector_operation: VectorOperation | None =␣
↪None) -> list[float]:
        warnings.warn(
            "`BaseEmbeddingDriver.embed_string` is deprecated and will be removed in␣
↪a future release. `BaseEmbeddingDriver.embed` is a drop-in replacement.",
            DeprecationWarning,
            stacklevel=2,
        )
        return self.embed(string, vector_operation=vector_operation)

    def embed(
        self, value: str | TextArtifact | ImageArtifact, *, vector_operation:␣
↪VectorOperation | None = None
    ) -> list[float]:
        for attempt in self.retrying():
            with attempt:
                if isinstance(value, str):
                    if (
                        self.tokenizer is not None
                        and self.tokenizer.count_tokens(value) > self.tokenizer.max_
↪input_tokens
                    ):
                        return self._embed_long_string(value, vector_operation=vector_
↪operation)
                    return self.try_embed_chunk(value, vector_operation=vector_
↪operation)
                if isinstance(value, TextArtifact):
                    return self.embed(value.to_text(), vector_operation=vector_
```

(continues on next page)

(continued from previous page)

```
↪operation)
                if isinstance(value, ImageArtifact):
                    return self.try_embed_artifact(value, vector_operation=vector_
↪operation)
        raise RuntimeError("Failed to embed string.")

    def try_embed_artifact(
        self, artifact: TextArtifact | ImageArtifact, *, vector_operation:␣
↪VectorOperation | None = None
    ) -> list[float]:
        # TODO: Mark as abstract method for griptape 2.0
        if isinstance(artifact, TextArtifact):
            return self.try_embed_chunk(artifact.value, vector_operation=vector_
↪operation)
        raise ValueError(f"{self.__class__.__name__} does not support embedding␣
↪images.")

    @abstractmethod
    def try_embed_chunk(self, chunk: str, *, vector_operation: VectorOperation | None␣
↪= None) -> list[float]:
        # TODO: Remove for griptape 2.0, subclasses should implement `try_embed_
↪artifact` instead
        ...

    def _embed_long_string(self, string: str, *, vector_operation: VectorOperation |␣
↪None = None) -> list[float]:
        """Embeds a string that is too long to embed in one go.

        Adapted from: https://github.com/openai/openai-cookbook/blob/
↪683e5f5a71bc7a1b0e5b7a35e087f53cc55fceea/examples/Embedding_long_inputs.ipynb
        """
        chunks = self.chunker.chunk(string)  # pyright:␣
↪ignore[reportOptionalMemberAccess] In practice this is never None

        embedding_chunks = []
        length_chunks = []
        for chunk in chunks:
            embedding_chunks.append(self.embed(chunk.value, vector_operation=vector_
↪operation))
            length_chunks.append(len(chunk))

        # generate weighted averages
        embedding_chunks = np.average(embedding_chunks, axis=0, weights=length_chunks)

        # normalize length to 1
        embedding_chunks = embedding_chunks / np.linalg.norm(embedding_chunks)

        return embedding_chunks.tolist()
```

- `chunker = field(init=False)` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

- `tokenizer = field(default=None, kw_only=True)` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
def __attrs_post_init__(self) -> None:
    self.chunker = TextChunker(tokenizer=self.tokenizer) if self.tokenizer else None
```

_embed_long_string(string, *, vector_operation=None)

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
def _embed_long_string(self, string: str, *, vector_operation: VectorOperation | None␣
↪= None) -> list[float]:
    """Embeds a string that is too long to embed in one go.

    Adapted from: https://github.com/openai/openai-cookbook/blob/
↪683e5f5a71bc7a1b0e5b7a35e087f53cc55fceea/examples/Embedding_long_inputs.ipynb
    """
    chunks = self.chunker.chunk(string)  # pyright:␣
↪ignore[reportOptionalMemberAccess] In practice this is never None

    embedding_chunks = []
    length_chunks = []
    for chunk in chunks:
        embedding_chunks.append(self.embed(chunk.value, vector_operation=vector_
↪operation))
        length_chunks.append(len(chunk))

    # generate weighted averages
    embedding_chunks = np.average(embedding_chunks, axis=0, weights=length_chunks)

    # normalize length to 1
    embedding_chunks = embedding_chunks / np.linalg.norm(embedding_chunks)

    return embedding_chunks.tolist()
```

embed(value, *, vector_operation=None)

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
def embed(
    self, value: str | TextArtifact | ImageArtifact, *, vector_operation:␣
↪VectorOperation | None = None
) -> list[float]:
    for attempt in self.retrying():
        with attempt:
            if isinstance(value, str):
                if (
                    self.tokenizer is not None
                    and self.tokenizer.count_tokens(value) > self.tokenizer.max_input_
↪tokens
```

(continues on next page)

(continued from previous page)

```
                ):
                    return self._embed_long_string(value, vector_operation=vector_
↪operation)
                return self.try_embed_chunk(value, vector_operation=vector_operation)
            if isinstance(value, TextArtifact):
                return self.embed(value.to_text(), vector_operation=vector_operation)
            if isinstance(value, ImageArtifact):
                return self.try_embed_artifact(value, vector_operation=vector_
↪operation)
    raise RuntimeError("Failed to embed string.")
```

embed_string(string, *, vector_operation=None)

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
def embed_string(self, string: str, *, vector_operation: VectorOperation | None =␣
↪None) -> list[float]:
    warnings.warn(
        "`BaseEmbeddingDriver.embed_string` is deprecated and will be removed in a␣
↪future release. `BaseEmbeddingDriver.embed` is a drop-in replacement.",
        DeprecationWarning,
        stacklevel=2,
    )
    return self.embed(string, vector_operation=vector_operation)
```

embed_text_artifact(artifact, *, vector_operation=None)

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
def embed_text_artifact(
    self, artifact: TextArtifact, *, vector_operation: VectorOperation | None = None
) -> list[float]:
    warnings.warn(
        "`BaseEmbeddingDriver.embed_text_artifact` is deprecated and will be removed␣
↪in a future release. `BaseEmbeddingDriver.embed` is a drop-in replacement.",
        DeprecationWarning,
        stacklevel=2,
    )
    return self.embed(artifact, vector_operation=vector_operation)
```

try_embed_artifact(artifact, *, vector_operation=None)

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
def try_embed_artifact(
    self, artifact: TextArtifact | ImageArtifact, *, vector_operation:␣
↪VectorOperation | None = None
) -> list[float]:
    # TODO: Mark as abstract method for griptape 2.0
    if isinstance(artifact, TextArtifact):
```

(continues on next page)

(continued from previous page)

```
        return self.try_embed_chunk(artifact.value, vector_operation=vector_operation)
    raise ValueError(f"{self.__class__.__name__} does not support embedding images.")
```

try_embed_chunk(chunk, *, vector_operation=None)abstractmethod

Source Code in griptape/drivers/embedding/base_embedding_driver.py

```
@abstractmethod
def try_embed_chunk(self, chunk: str, *, vector_operation: VectorOperation | None =␣
↪None) -> list[float]:
    # TODO: Remove for griptape 2.0, subclasses should implement `try_embed_artifact`␣
↪instead
    ...
```

cohere

- `__all__ = ['CohereEmbeddingDriver']` module-attribute

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="cohere.mdx#griptape.drivers.embedding.cohere.CohereEmbeddingDriver.api_key" title="api_key = field(kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.embedding.cohere.CohereEmbeddingDriver.api_key)"><code class="language-python">	*str*	Cohere API key.

continues on next page

Table 137 – continued from previous page

Name	Type	Description
<a href="base_embedding_driver.mdx#griptape.drivers.embedding.base_embedding_driver.BaseEmbeddingDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.cohere.CohereEmbeddingDriver.model)"><code class="language-python">	*str*	Cohere model name.
<a href="cohere.mdx#griptape.drivers.embedding.cohere.CohereEmbeddingDriver.client" title="client() (griptape.drivers.embedding.cohere.CohereEmbeddingDriver.client)"><code class="language-python">	*Client*	Custom *cohere.Client*.
<a href="cohere.mdx#griptape.drivers.embedding.cohere.CohereEmbeddingDriver.tokenizer" title="tokenizer = field(default=Factory(lambda self: CohereTokenizer(model=self.model, client=self.client), takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.drivers.embedding.cohere.CohereEmbeddingDriver.tokenizer)"><code class="language-python">	<a href="../../tokenizers#griptape.tokenizers.CohereTokenizer" title="CohereTokenizer (griptape.tokenizers.CohereTokenizer)"><code class="language-python">CohereTokenizer</code></a	Custom *CohereTokenizer*.

continues on next page

Table 137 – continued from previous page

Name	Type	Description
<a href="cohere.mdx#griptape.drivers.embedding.cohere.CohereEmbeddingDriver.input_type" title="input_type = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.cohere.CohereEmbeddingDriver.input_type)"><code class="language-python">	*str*	Cohere embedding input type.

Source Code in griptape/drivers/embedding/cohere_embedding_driver.py

```
@define
class CohereEmbeddingDriver(BaseEmbeddingDriver):
    """Cohere Embedding Driver.

    Attributes:
        api_key: Cohere API key.
        model:  Cohere model name.
        client: Custom `cohere.Client`.
        tokenizer: Custom `CohereTokenizer`.
        input_type: Cohere embedding input type.
    """

    DEFAULT_MODEL = "models/embedding-001"

    api_key: str = field(kw_only=True, metadata={"serializable": False})
    input_type: str = field(kw_only=True, metadata={"serializable": True})
    _client: Optional[Client] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})
    tokenizer: CohereTokenizer = field(
        default=Factory(lambda self: CohereTokenizer(model=self.model, client=self.
↪client), takes_self=True),
        kw_only=True,
    )

    @lazy_property()
    def client(self) -> Client:
        return import_optional_dependency("cohere").Client(self.api_key)

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        result = self.client.embed(texts=[chunk], model=self.model, input_type=self.
```

(continues on next page)

(continued from previous page)

```
↪input_type)

        if isinstance(result.embeddings, list):
            return result.embeddings[0]
        raise ValueError("Non-float embeddings are not supported.")
```

- `DEFAULT_MODEL = 'models/embedding-001'` class-attribute instance-attribute
- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `input_type = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: CohereTokenizer(model=self.model, client=self.client), takes_self=True), kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/cohere_embedding_driver.py

```
@lazy_property()
def client(self) -> Client:
    return import_optional_dependency("cohere").Client(self.api_key)
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/cohere_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    result = self.client.embed(texts=[chunk], model=self.model, input_type=self.input_
↪type)

    if isinstance(result.embeddings, list):
        return result.embeddings[0]
    raise ValueError("Non-float embeddings are not supported.")
```

cohere_embedding_driver

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="cohere_embedding_driver.mdx#griptape.drivers.embedding.cohere_embedding_driver.CohereEmbeddingDriver.api_key" title="api_key = field(kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.embedding.cohere_embedding_driver.CohereEmbeddingDriver.api_key)"><code class="language-python">	*str*	Cohere API key.
<a href="./#griptape.drivers.embedding.BaseEmbeddingDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.cohere_embedding_driver.CohereEmbeddingDriver.model)"><code class="language-python">	*str*	Cohere model name.
<a href="cohere_embedding_driver.mdx#griptape.drivers.embedding.cohere_embedding_driver.CohereEmbeddingDriver.client" title="client() (griptape.drivers.embedding.cohere_embedding_driver.CohereEmbeddingDriver.client)"><code class="language-python">	*Client*	Custom *cohere.Client*.

continues on next page

Table 138 – continued from previous page

Name	Type	Description
<a href="cohere_embedding_driver.mdx#griptape.drivers.embedding.cohere_embedding_driver.CohereEmbeddingDriver.tokenizer" title="tokenizer = field(default=Factory(lambda self: CohereTokenizer(model=self.model, client=self.client), takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.drivers.embedding.cohere_embedding_driver.CohereEmbeddingDriver.tokenizer)"><code class="language-python">	<a href="../../tokenizers#griptape.tokenizers.CohereTokenizer" title="CohereTokenizer (griptape.tokenizers.CohereTokenizer)"><code class="language-python">CohereTokenizer</code></a	Custom *CohereTokenizer*.
<a href="cohere_embedding_driver.mdx#griptape.drivers.embedding.cohere_embedding_driver.CohereEmbeddingDriver.input_type" title="input_type = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.cohere_embedding_driver.CohereEmbeddingDriver.input_type)"><code class="language-python">	*str*	Cohere embedding input type.

Source Code in griptape/drivers/embedding/cohere_embedding_driver.py

```
@define
class CohereEmbeddingDriver(BaseEmbeddingDriver):
    """Cohere Embedding Driver.
```

(continues on next page)

(continued from previous page)

```
    Attributes:
        api_key: Cohere API key.
        model:  Cohere model name.
        client: Custom `cohere.Client`.
        tokenizer: Custom `CohereTokenizer`.
        input_type: Cohere embedding input type.
    """

    DEFAULT_MODEL = "models/embedding-001"

    api_key: str = field(kw_only=True, metadata={"serializable": False})
    input_type: str = field(kw_only=True, metadata={"serializable": True})
    _client: Optional[Client] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})
    tokenizer: CohereTokenizer = field(
        default=Factory(lambda self: CohereTokenizer(model=self.model, client=self.
↪client), takes_self=True),
        kw_only=True,
    )

    @lazy_property()
    def client(self) -> Client:
        return import_optional_dependency("cohere").Client(self.api_key)

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        result = self.client.embed(texts=[chunk], model=self.model, input_type=self.
↪input_type)

        if isinstance(result.embeddings, list):
            return result.embeddings[0]
        raise ValueError("Non-float embeddings are not supported.")
```

- `DEFAULT_MODEL = 'models/embedding-001'` class-attribute instance-attribute
- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `input_type = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: CohereTokenizer(model=self.model, client=self.client), takes_self=True), kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/cohere_embedding_driver.py

```
@lazy_property()
def client(self) -> Client:
    return import_optional_dependency("cohere").Client(self.api_key)
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/cohere_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    result = self.client.embed(texts=[chunk], model=self.model, input_type=self.input_
↪type)

    if isinstance(result.embeddings, list):
        return result.embeddings[0]
    raise ValueError("Non-float embeddings are not supported.")
```

dummy

- `__all__ = ['DummyEmbeddingDriver']` module-attribute

Bases: *BaseEmbeddingDriver* “)

Source Code in griptape/drivers/embedding/dummy_embedding_driver.py

```
@define
class DummyEmbeddingDriver(BaseEmbeddingDriver):
    model: None = field(init=False, default=None, kw_only=True)

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        raise DummyError(__class__.__name__, "try_embed_chunk")
```

- `model = field(init=False, default=None, kw_only=True)` class-attribute instance-attribute

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/dummy_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    raise DummyError(__class__.__name__, "try_embed_chunk")
```

dummy_embedding_driver

Bases: *BaseEmbeddingDriver* “)

Source Code in griptape/drivers/embedding/dummy_embedding_driver.py

```
@define
class DummyEmbeddingDriver(BaseEmbeddingDriver):
    model: None = field(init=False, default=None, kw_only=True)

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        raise DummyError(__class__.__name__, "try_embed_chunk")
```

- `model = field(init=False, default=None, kw_only=True)` class-attribute instance-attribute

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/dummy_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    raise DummyError(__class__.__name__, "try_embed_chunk")
```

google

- `__all__ = ['GoogleEmbeddingDriver']` module-attribute

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="google.mdx#griptape.drivers.embedding.google.GoogleEmbeddingDriver.api_key" title="api_key = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.embedding.google.GoogleEmbedding-Driver.api_key)"><code class="language-python">	*Optional[str]*	Google API key.
<a href="google.mdx#griptape.drivers.embedding.google.GoogleEmbeddingDriver.model" title="model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.google.GoogleEmbedding-Driver.model)"><code class="language-python">	*str*	Google model name.

continues on next page

Table 139 – continued from previous page

Name	Type	Description
<a href="google.mdx#griptape.drivers.embedding.google.GoogleEmbeddingDriver.task_type" title="task_type = field(default='retrieval_document', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.google.GoogleEmbeddingDriver.task_type)"><code class="language-python">	*str*	Embedding model task type (<https://ai.google.dev/tutorials/python_quick-start#use_embeddings>). Defaults to *retrieval_document*.
<a href="google.mdx#griptape.drivers.embedding.google.GoogleEmbeddingDriver.title" title="title = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.google.GoogleEmbeddingDriver.title)"><code class="language-python">	*Optional[str]*	Optional title for the content. Only works with *retrieval_document* task type.

Source Code in griptape/drivers/embedding/google_embedding_driver.py

Optional title for the content. Only works with `retrieval_document` task type.

- `DEFAULT_MODEL = 'models/embedding-001'` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `task_type = field(default='retrieval_document', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `title = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_params(chunk)

Source Code in griptape/drivers/embedding/google_embedding_driver.py

```
def _params(self, chunk: str) -> dict:
    return {"input": chunk, "model": self.model}
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/google_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    genai = import_optional_dependency("google.generativeai")
    genai.configure(api_key=self.api_key)

    result = genai.embed_content(model=self.model, content=chunk, task_type=self.task_
↪type, title=self.title)

    return result["embedding"]
```

google_embedding_driver

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="google_embedding_driver.mdx#griptape.drivers.embedding.google_embedding_driver.GoogleEmbeddingDriver.api_key" title="api_key = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.embedding.google_embedding_driver.GoogleEmbedding-Driver.api_key)"><code class="language-python">	*Optional[str]*	Google API key.

continues on next page

Table 140 – continued from previous page

Name	Type	Description
<a href="google_embedding_driver.mdx#griptape.drivers.embedding.google_embedding_driver.GoogleEmbeddingDriver.model" title="model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.google_embedding_driver.GoogleEmbedding-Driver.model)"><code class="language-python">	*str*	Google model name.
<a href="google_embedding_driver.mdx#griptape.drivers.embedding.google_embedding_driver.GoogleEmbedding-Driver.task_type" title="task_type = field(default='retrieval_document', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.google_embedding_driver.GoogleEmbedding-Driver.task_type)"><code class="language-python">	*str*	Embedding model task type (<https://ai.google.dev/tutorials/python_quick-start#use_embeddings>). Defaults to *retrieval_document*.

continues on next page

Table 140 – continued from previous page

Name	Type	Description
<a href="google_embedding_driver.mdx#griptape.drivers.embedding.google_embedding_driver.GoogleEmbeddingDriver.title" title="title = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.google_embedding_driver.GoogleEmbeddingDriver.title)"><code class="language-python">	*Optional[str]*	Optional title for the content. Only works with *retrieval_document* task type.

Source Code in griptape/drivers/embedding/google_embedding_driver.py

Optional title for the content. Only works with `retrieval_document` task type.

- `DEFAULT_MODEL = 'models/embedding-001'` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `task_type = field(default='retrieval_document', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `title = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_params(chunk)

Source Code in griptape/drivers/embedding/google_embedding_driver.py

```
def _params(self, chunk: str) -> dict:
    return {"input": chunk, "model": self.model}
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/google_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    genai = import_optional_dependency("google.generativeai")
    genai.configure(api_key=self.api_key)
```

(continues on next page)

(continued from previous page)

```
    result = genai.embed_content(model=self.model, content=chunk, task_type=self.task_
↪type, title=self.title)

    return result["embedding"]
```

huggingface_hub

- `__all__ = ['HuggingFaceHubEmbeddingDriver']` module-attribute

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="hugging-face_hub.mdx#grip-tape.drivers.em-bedding.hugging-face_hub.Hugging-FaceHubEmbedding-Driver.api_token" title="api_token = field(kw_only=True, metadata={'seri-alizable': True})">class-attribute instance-attribute (griptape.drivers.em-bedding.hugging-face_hub.Hugging-FaceHubEmbed-dingDriver.api_to-ken)"><code class="language-python">	*str*	Hugging Face Hub API to-ken.

continues on next page

Table 141 – continued from previous page

Name	Type	Description
<a href="base_embedding_driver.mdx#griptape.drivers.embedding.base_embedding_driver.BaseEmbeddingDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.huggingface_hub.HuggingFaceHubEmbeddingDriver.model)"><code class="language-python">	*str*	Hugging Face Hub model name.
<a href="huggingface_hub.mdx#griptape.drivers.embedding.huggingface_hub.HuggingFaceHubEmbeddingDriver.client" title="client() (griptape.drivers.embedding.huggingface_hub.HuggingFaceHubEmbeddingDriver.client)"><code class="language-python">	*InferenceClient*	Custom *InferenceApi*.

Source Code in griptape/drivers/embedding/huggingface_hub_embedding_driver.py

```
@define
class HuggingFaceHubEmbeddingDriver(BaseEmbeddingDriver):
    """Hugging Face Hub Embedding Driver.

    Attributes:
        api_token: Hugging Face Hub API token.
        model: Hugging Face Hub model name.
        client: Custom `InferenceApi`.
    """

    api_token: str = field(kw_only=True, metadata={"serializable": True})
    _client: Optional[InferenceClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )
```

(continues on next page)

(continued from previous page)

```
    @lazy_property()
    def client(self) -> InferenceClient:
        return import_optional_dependency("huggingface_hub").InferenceClient(
            model=self.model,
            token=self.api_token,
        )

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        response = self.client.feature_extraction(chunk)

        return [float(val) for val in response.flatten().tolist()]
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_token = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/huggingface_hub_embedding_driver.py

```
@lazy_property()
def client(self) -> InferenceClient:
    return import_optional_dependency("huggingface_hub").InferenceClient(
        model=self.model,
        token=self.api_token,
    )
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/huggingface_hub_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    response = self.client.feature_extraction(chunk)

    return [float(val) for val in response.flatten().tolist()]
```

huggingface_hub_embedding_driver

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="hugging-face_hub_embed-ding_driver.mdx#grip-tape.drivers.em-bedding.hugging-face_hub_embed-ding_driver.Hugging-FaceHubEmbedding-Driver.api_token" title="api_token = field(kw_only=True, metadata={'seri-alizable': True}) class-attribute instance-attribute (griptape.drivers.em-bedding.hugging-face_hub_embed-ding_driver.Hug-gingFaceHubEmbed-dingDriver.api_to-ken)"><code class="language-python">	*str*	Hugging Face Hub API to-ken.
<a href="./#grip-tape.drivers.embed-ding.BaseEmbed-dingDriver.model" title="model = field(kw_only=True, metadata={'seri-alizable': True}) class-attribute instance-attribute (griptape.drivers.em-bedding.hugging-face_hub_embed-ding_driver.Hugging-FaceHubEmbedding-Driver.model)"><code class="language-python">	*str*	Hugging Face Hub model name.

continues on next page

Table 142 – continued from previous page

Name	Type	Description
<a href="hugging-face_hub_embed-ding_driver.mdx#grip-tape.drivers.em-bedding.hugging-face_hub_embed-ding_driver.Hug-gingFaceHubEmbed-dingDriver.client" title="client() (grip-tape.drivers.em-bedding.hugging-face_hub_embed-ding_driver.Hugging-FaceHubEmbedding-Driver.client)"><code class="language-python">	*InferenceClient*	Custom *InferenceApi*.

Source Code in griptape/drivers/embedding/huggingface_hub_embedding_driver.py

```
@define
class HuggingFaceHubEmbeddingDriver(BaseEmbeddingDriver):
    """Hugging Face Hub Embedding Driver.

    Attributes:
        api_token: Hugging Face Hub API token.
        model: Hugging Face Hub model name.
        client: Custom `InferenceApi`.
    """

    api_token: str = field(kw_only=True, metadata={"serializable": True})
    _client: Optional[InferenceClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> InferenceClient:
        return import_optional_dependency("huggingface_hub").InferenceClient(
            model=self.model,
            token=self.api_token,
        )

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        response = self.client.feature_extraction(chunk)

        return [float(val) for val in response.flatten().tolist()]
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_token = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/huggingface_hub_embedding_driver.py

```
@lazy_property()
def client(self) -> InferenceClient:
    return import_optional_dependency("huggingface_hub").InferenceClient(
        model=self.model,
        token=self.api_token,
    )
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/huggingface_hub_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    response = self.client.feature_extraction(chunk)

    return [float(val) for val in response.flatten().tolist()]
```

nvidia_nim

- `__all__ = ['NvidiaNimEmbeddingDriver']` module-attribute

Bases: *OpenAiEmbeddingDriver* “)

Source Code in griptape/drivers/embedding/nvidia_nim_embedding_driver.py

```
@define
class NvidiaNimEmbeddingDriver(OpenAiEmbeddingDriver):
    """Nvidia Embedding Driver. The API is OpenAI compatible, but requires an extra␣
↪parameter input_type."""

    def try_embed_chunk(self, chunk: str, *, vector_operation: VectorOperation | None␣
↪= None, **kwargs) -> list[float]:
        if vector_operation not in get_args(VectorOperation):
            raise ValueError(f"invalid value for vector_operation, must be one of␣
↪{get_args(VectorOperation)}")

        extra_body = {
            "input_type": "query" if vector_operation == "query" else "passage",
        }

        return self.client.embeddings.create(**self._params(chunk), extra_body=extra_␣
↪body).data[0].embedding
```

try_embed_chunk(chunk, *, vector_operation=None, **kwargs)

Source Code in griptape/drivers/embedding/nvidia_nim_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, *, vector_operation: VectorOperation | None =␣
↪None, **kwargs) -> list[float]:
    if vector_operation not in get_args(VectorOperation):
        raise ValueError(f"invalid value for vector_operation, must be one of {get_␣
```

(continues on next page)

(continued from previous page)

```
↪args(VectorOperation)}")

    extra_body = {
        "input_type": "query" if vector_operation == "query" else "passage",
    }

    return self.client.embeddings.create(**self._params(chunk), extra_body=extra_
↪body).data[0].embedding
```

nvidia_nim_embedding_driver

Bases: *OpenAiEmbeddingDriver* “)

Source Code in griptape/drivers/embedding/nvidia_nim_embedding_driver.py

```
@define
class NvidiaNimEmbeddingDriver(OpenAiEmbeddingDriver):
    """Nvidia Embedding Driver. The API is OpenAI compatible, but requires an extra␣
↪parameter input_type."""

    def try_embed_chunk(self, chunk: str, *, vector_operation: VectorOperation | None␣
↪= None, **kwargs) -> list[float]:
        if vector_operation not in get_args(VectorOperation):
            raise ValueError(f"invalid value for vector_operation, must be one of
↪{get_args(VectorOperation)}")

        extra_body = {
            "input_type": "query" if vector_operation == "query" else "passage",
        }

        return self.client.embeddings.create(**self._params(chunk), extra_body=extra_
↪body).data[0].embedding
```

try_embed_chunk(chunk, *, vector_operation=None, **kwargs)

Source Code in griptape/drivers/embedding/nvidia_nim_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, *, vector_operation: VectorOperation | None =␣
↪None, **kwargs) -> list[float]:
    if vector_operation not in get_args(VectorOperation):
        raise ValueError(f"invalid value for vector_operation, must be one of {get_
↪args(VectorOperation)}")

    extra_body = {
        "input_type": "query" if vector_operation == "query" else "passage",
    }

    return self.client.embeddings.create(**self._params(chunk), extra_body=extra_
↪body).data[0].embedding
```

ollama

- `__all__ = ['OllamaEmbeddingDriver']` module-attribute

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="ollama.mdx#griptape.drivers.embedding.ollama.OllamaEmbeddingDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.ollama.OllamaEmbeddingDriver.model)"><code class="language-python">	*str*	Ollama embedding model name.
<a href="ollama.mdx#griptape.drivers.embedding.ollama.OllamaEmbeddingDriver.host" title="host = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.ollama.OllamaEmbeddingDriver.host)"><code class="language-python">	*Optional[str]*	Optional Ollama host.

continues on next page

Table 143 – continued from previous page

Name	Type	Description
<a href="ollama.mdx#griptape.drivers.embedding.ollama.OllamaEmbeddingDriver.client" title="client() (griptape.drivers.embedding.ollama.OllamaEmbeddingDriver.client)"><code class="language-python">	*Client*	Ollama *Client*.

Source Code in griptape/drivers/embedding/ollama_embedding_driver.py

```
@define
class OllamaEmbeddingDriver(BaseEmbeddingDriver):
    """Ollama Embedding Driver.

    Attributes:
        model: Ollama embedding model name.
        host: Optional Ollama host.
        client: Ollama `Client`.
    """

    model: str = field(kw_only=True, metadata={"serializable": True})
    host: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    _client: Optional[Client] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> Client:
        return import_optional_dependency("ollama").Client(host=self.host)

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        return list(self.client.embeddings(model=self.model, prompt=chunk)["embedding
↪"])
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `host = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/ollama_embedding_driver.py

```
@lazy_property()
def client(self) -> Client:
    return import_optional_dependency("ollama").Client(host=self.host)
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/ollama_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    return list(self.client.embeddings(model=self.model, prompt=chunk)["embedding"])
```

ollama_embedding_driver

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="ollama_embedding_driver.mdx#griptape.drivers.embedding.ollama_embedding_driver.OllamaEmbeddingDriver.model" title="model = field(kw_only=True, metadata={'serializable': True})" class-attribute instance-attribute (griptape.drivers.embedding.ollama_embedding_driver.OllamaEmbeddingDriver.model)"><code class="language-python">	*str*	Ollama embedding model name.

continues on next page

Table 144 – continued from previous page

Name	Type	Description
<a href="ollama_embedding_driver.mdx#griptape.drivers.embedding.ollama_embedding_driver.OllamaEmbeddingDriver.host" title="host = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.ollama_embedding_driver.OllamaEmbeddingDriver.host)"><code class="language-python">	*Optional[str]*	Optional Ollama host.
<a href="ollama_embedding_driver.mdx#griptape.drivers.embedding.ollama_embedding_driver.OllamaEmbeddingDriver.client" title="client() (griptape.drivers.embedding.ollama_embedding_driver.OllamaEmbeddingDriver.client)"><code class="language-python">	*Client*	Ollama *Client*.

Source Code in griptape/drivers/embedding/ollama_embedding_driver.py

```
@define
class OllamaEmbeddingDriver(BaseEmbeddingDriver):
    """Ollama Embedding Driver.

    Attributes:
        model: Ollama embedding model name.
        host: Optional Ollama host.
        client: Ollama `Client`.
    """

    model: str = field(kw_only=True, metadata={"serializable": True})
    host: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    _client: Optional[Client] = field(default=None, kw_only=True, alias="client",␣
```

(continues on next page)

(continued from previous page)

```
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> Client:
        return import_optional_dependency("ollama").Client(host=self.host)

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        return list(self.client.embeddings(model=self.model, prompt=chunk)["embedding
↪"])
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `host = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/ollama_embedding_driver.py

```
@lazy_property()
def client(self) -> Client:
    return import_optional_dependency("ollama").Client(host=self.host)
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/ollama_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    return list(self.client.embeddings(model=self.model, prompt=chunk)["embedding"])
```

openai

- `__all__ = ['AzureOpenAiEmbeddingDriver', 'OpenAiEmbeddingDriver']` module-attribute

Bases: *OpenAiEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="openai.mdx#griptape.drivers.embedding.openai.AzureOpenAiEmbeddingDriver.azure_deployment" title="azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.openai.AzureOpenAiEmbeddingDriver.azure_deployment)"><code class="language-python">	*str*	An optional Azure OpenAi deployment id. Defaults to the model name.
<a href="openai.mdx#griptape.drivers.embedding.openai.AzureOpenAiEmbeddingDriver.azure_endpoint" title="azure_endpoint = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.openai.AzureOpenAiEmbeddingDriver.azure_endpoint)"><code class="language-python">	*str*	An Azure OpenAi endpoint.

continues on next page

Table 145 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.embedding.openai.AzureOpenAiEmbeddingDriver.azure_ad_token" title="azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.embedding.openai.AzureOpenAiEmbeddingDriver.azure_ad_token)"><code class="language-python">	*Optional[str]*	An optional Azure Active Directory token.
<a href="openai.mdx#griptape.drivers.embedding.openai.AzureOpenAiEmbeddingDriver.azure_ad_token_provider" title="azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.embedding.openai.AzureOpenAiEmbeddingDriver.azure_ad_token_provider)"><code class="language-python">	*Optional[Callable[[], str]]*	An optional Azure Active Directory token provider.

continues on next page

Table 145 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.embedding.openai.AzureOpenAiEmbeddingDriver.api_version" title="api_version = field(default='2024-10-21', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.openai.AzureOpenAiEmbeddingDriver.api_version)"><code class="language-python">	*str*	An Azure OpenAi API version.
<a href="openai.mdx#griptape.drivers.embedding.openai.AzureOpenAiEmbeddingDriver.tokenizer" title="tokenizer = field(default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.drivers.embedding.openai.AzureOpenAiEmbeddingDriver.tokenizer)"><code class="language-python">	<a href="../../tokenizers#griptape.tokenizers.OpenAiTokenizer" title="OpenAiTokenizer (griptape.tokenizers.OpenAiTokenizer)"><code class="language-python">OpenAiTokenizer</code></	An *OpenAiTokenizer*.

continues on next page

Table 145 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.embedding.openai.AzureOpenAiEmbeddingDriver.client" title="client() (griptape.drivers.embedding.openai.AzureOpenAiEmbeddingDriver.client)"><code class="language-python">	*AzureOpenAI*	An *openai.AzureOpenAI* client.

Source Code in griptape/drivers/embedding/azure_openai_embedding_driver.py

```
@define
class AzureOpenAiEmbeddingDriver(OpenAiEmbeddingDriver):
    """Azure OpenAi Embedding Driver.

    Attributes:
        azure_deployment: An optional Azure OpenAi deployment id. Defaults to the␣
↪model name.
        azure_endpoint: An Azure OpenAi endpoint.
        azure_ad_token: An optional Azure Active Directory token.
        azure_ad_token_provider: An optional Azure Active Directory token provider.
        api_version: An Azure OpenAi API version.
        tokenizer: An `OpenAiTokenizer`.
        client: An `openai.AzureOpenAI` client.
    """

    azure_deployment: str = field(
        kw_only=True,
        default=Factory(lambda self: self.model, takes_self=True),
        metadata={"serializable": True},
    )
    azure_endpoint: str = field(kw_only=True, metadata={"serializable": True})
    azure_ad_token: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": False})
    azure_ad_token_provider: Optional[Callable[[], str]] = field(
        kw_only=True,
        default=None,
        metadata={"serializable": False},
    )
    api_version: str = field(default="2024-10-21", kw_only=True, metadata={
↪"serializable": True})
    tokenizer: OpenAiTokenizer = field(
        default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
```

(continues on next page)

(continued from previous page)

```
    _client: Optional[openai.AzureOpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.AzureOpenAI:
        return openai.AzureOpenAI(
            organization=self.organization,
            api_key=self.api_key,
            api_version=self.api_version,
            azure_endpoint=self.azure_endpoint,
            azure_deployment=self.azure_deployment,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_version = field(default='2024-10-21', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True})` class-attribute instance-attribute
- `azure_endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/azure_openai_embedding_driver.py

```
@lazy_property()
def client(self) -> openai.AzureOpenAI:
    return openai.AzureOpenAI(
        organization=self.organization,
        api_key=self.api_key,
        api_version=self.api_version,
        azure_endpoint=self.azure_endpoint,
        azure_deployment=self.azure_deployment,
        azure_ad_token=self.azure_ad_token,
        azure_ad_token_provider=self.azure_ad_token_provider,
    )
```

OpenAiEmbeddingDriver

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="openai.mdx#griptape.drivers.embedding.openai.OpenAiEmbeddingDriver.model" title="model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.openai.OpenAiEmbeddingDriver.model)"><code class="language-python">	*str*	OpenAI embedding model name. Defaults to *text-embedding-3-small*.
<a href="openai.mdx#griptape.drivers.embedding.openai.OpenAiEmbeddingDriver.base_url" title="base_url = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.openai.OpenAiEmbeddingDriver.base_url)"><code class="language-python">	*Optional[str]*	API URL. Defaults to OpenAI's v1 API URL.

continues on next page

Table 146 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.embedding.openai.OpenAiEmbeddingDriver.api_key" title="api_key = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.embedding.openai.OpenAiEmbeddingDriver.api_key)"><code class="language-python">	*Optional[str]*	API key to pass directly. Defaults to *OPENAI_API_KEY* environment variable.
<a href="openai.mdx#griptape.drivers.embedding.openai.OpenAiEmbeddingDriver.organization" title="organization = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.openai.OpenAiEmbeddingDriver.organization)"><code class="language-python">	*Optional[str]*	OpenAI organization. Defaults to 'OPENAI_ORGANIZATION' environment variable.

continues on next page

Table 146 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.embedding.openai.OpenAiEmbeddingDriver.tokenizer" title="tokenizer = field(default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.drivers.embedding.openai.OpenAiEmbeddingDriver.tokenizer)"><code class="language-python">	<a href="../../tokenizers#griptape.tokenizers.OpenAiTokenizer" title="OpenAiTokenizer (griptape.tokenizers.OpenAiTokenizer)"><code class="language-python">OpenAiTokenizer</code></a	Optionally provide custom *OpenAiTokenizer*.
<a href="openai.mdx#griptape.drivers.embedding.openai.OpenAiEmbeddingDriver.client" title="client() (griptape.drivers.embedding.openai.OpenAiEmbeddingDriver.client)"><code class="language-python">	*OpenAI*	Optionally provide custom *openai.OpenAI* client.
azure_deployment	*OpenAI*	An Azure OpenAi deployment id.
azure_endpoint	*OpenAI*	An Azure OpenAi endpoint.
azure_ad_token	*OpenAI*	An optional Azure Active Directory token.
azure_ad_token_provider	*OpenAI*	An optional Azure Active Directory token provider.
api_version	*OpenAI*	An Azure OpenAi API version.

Source Code in griptape/drivers/embedding/openai_embedding_driver.py

```
@define
class OpenAiEmbeddingDriver(BaseEmbeddingDriver):
    """OpenAI Embedding Driver.
```

(continues on next page)

(continued from previous page)

```
    Attributes:
        model: OpenAI embedding model name. Defaults to `text-embedding-3-small`.
        base_url: API URL. Defaults to OpenAIs v1 API URL.
        api_key: API key to pass directly. Defaults to `OPENAI_API_KEY` environment␣
↪variable.
        organization: OpenAI organization. Defaults to OPENAI_ORGANIZATION␣
↪environment variable.
        tokenizer: Optionally provide custom `OpenAiTokenizer`.
        client: Optionally provide custom `openai.OpenAI` client.
        azure_deployment: An Azure OpenAi deployment id.
        azure_endpoint: An Azure OpenAi endpoint.
        azure_ad_token: An optional Azure Active Directory token.
        azure_ad_token_provider: An optional Azure Active Directory token provider.
        api_version: An Azure OpenAi API version.
    """

    DEFAULT_MODEL = "text-embedding-3-small"

    model: str = field(default=DEFAULT_MODEL, kw_only=True, metadata={"serializable":␣
↪True})
    base_url: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    organization: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    tokenizer: OpenAiTokenizer = field(
        default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
    _client: Optional[openai.OpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.OpenAI:
        return openai.OpenAI(api_key=self.api_key, base_url=self.base_url,␣
↪organization=self.organization)

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        # Address a performance issue in older ada models
        # https://github.com/openai/openai-python/issues/418#issuecomment-1525939500
        if self.model.endswith("001"):
            chunk = chunk.replace("\n", " ")
        return self.client.embeddings.create(**self._params(chunk)).data[0].embedding

    def _params(self, chunk: str) -> dict:
        return {"input": chunk, "model": self.model}
```

- `DEFAULT_MODEL = 'text-embedding-3-small'` class-attribute instance-attribute
- `_client = field(default=None, kw_only=True, alias='client',`

`metadata={'serializable': False})` class-attribute instance-attribute

- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `base_url = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `organization = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute

_params(chunk)

Source Code in griptape/drivers/embedding/openai_embedding_driver.py

```
def _params(self, chunk: str) -> dict:
    return {"input": chunk, "model": self.model}
```

client()

Source Code in griptape/drivers/embedding/openai_embedding_driver.py

```
@lazy_property()
def client(self) -> openai.OpenAI:
    return openai.OpenAI(api_key=self.api_key, base_url=self.base_url,␣
↪organization=self.organization)
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/openai_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    # Address a performance issue in older ada models
    # https://github.com/openai/openai-python/issues/418#issuecomment-1525939500
    if self.model.endswith("001"):
        chunk = chunk.replace("\n", " ")
    return self.client.embeddings.create(**self._params(chunk)).data[0].embedding
```

openai_embedding_driver

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="openai_embedding_driver.mdx#griptape.drivers.embedding.openai_embedding_driver.OpenAiEmbeddingDriver.model" title="model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.openai_embedding_driver.OpenAiEmbeddingDriver.model)"><code class="language-python">	*str*	OpenAI embedding model name. Defaults to *text-embedding-3-small*.
<a href="openai_embedding_driver.mdx#griptape.drivers.embedding.openai_embedding_driver.OpenAiEmbeddingDriver.base_url" title="base_url = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.openai_embedding_driver.OpenAiEmbeddingDriver.base_url)"><code class="language-python">	*Optional[str]*	API URL. Defaults to OpenAI's v1 API URL.

continues on next page

Table 147 – continued from previous page

Name	Type	Description
<a href="openai_embedding_driver.mdx#griptape.drivers.embedding.openai_embedding_driver.OpenAiEmbeddingDriver.api_key" title="api_key = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.embedding.openai_embedding_driver.OpenAiEmbeddingDriver.api_key)"><code class="language-python">	*Optional[str]*	API key to pass directly. Defaults to *OPENAI_API_KEY* environment variable.
<a href="openai_embedding_driver.mdx#griptape.drivers.embedding.openai_embedding_driver.OpenAiEmbeddingDriver.organization" title="organization = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.openai_embedding_driver.OpenAiEmbeddingDriver.organization)"><code class="language-python">	*Optional[str]*	OpenAI organization. Defaults to 'OPENAI_ORGANIZATION' environment variable.

continues on next page

Table 147 – continued from previous page

Name	Type	Description
<a href="openai_embedding_driver.mdx#griptape.drivers.embedding.openai_embedding_driver.OpenAiEmbeddingDriver.tokenizer" title="tokenizer = field(default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.drivers.embedding.openai_embedding_driver.OpenAiEmbeddingDriver.tokenizer)"><code class="language-python">	<a href="../../tokenizers#griptape.tokenizers.OpenAiTokenizer" title="OpenAiTokenizer (griptape.tokenizers.OpenAiTokenizer)"><code class="language-python">OpenAiTokenizer</code></a	Optionally provide custom *OpenAiTokenizer*.
<a href="openai_embedding_driver.mdx#griptape.drivers.embedding.openai_embedding_driver.OpenAiEmbeddingDriver.client" title="client() (griptape.drivers.embedding.openai_embedding_driver.OpenAiEmbeddingDriver.client)"><code class="language-python">	*OpenAI*	Optionally provide custom *openai.OpenAI* client.
azure_deployment	*OpenAI*	An Azure OpenAi deployment id.
azure_endpoint	*OpenAI*	An Azure OpenAi endpoint.
azure_ad_token	*OpenAI*	An optional Azure Active Directory token.
azure_ad_token_provider	*OpenAI*	An optional Azure Active Directory token provider.
api_version	*OpenAI*	An Azure OpenAi API version.

Source Code in griptape/drivers/embedding/openai_embedding_driver.py

```
@define
class OpenAiEmbeddingDriver(BaseEmbeddingDriver):
    """OpenAI Embedding Driver.

    Attributes:
        model: OpenAI embedding model name. Defaults to `text-embedding-3-small`.
        base_url: API URL. Defaults to OpenAIs v1 API URL.
        api_key: API key to pass directly. Defaults to `OPENAI_API_KEY` environment␣
↪variable.
        organization: OpenAI organization. Defaults to OPENAI_ORGANIZATION␣
↪environment variable.
        tokenizer: Optionally provide custom `OpenAiTokenizer`.
        client: Optionally provide custom `openai.OpenAI` client.
        azure_deployment: An Azure OpenAi deployment id.
        azure_endpoint: An Azure OpenAi endpoint.
        azure_ad_token: An optional Azure Active Directory token.
        azure_ad_token_provider: An optional Azure Active Directory token provider.
        api_version: An Azure OpenAi API version.
    """

    DEFAULT_MODEL = "text-embedding-3-small"

    model: str = field(default=DEFAULT_MODEL, kw_only=True, metadata={"serializable":␣
↪True})
    base_url: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    organization: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    tokenizer: OpenAiTokenizer = field(
        default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
    _client: Optional[openai.OpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.OpenAI:
        return openai.OpenAI(api_key=self.api_key, base_url=self.base_url,␣
↪organization=self.organization)

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        # Address a performance issue in older ada models
        # https://github.com/openai/openai-python/issues/418#issuecomment-1525939500
        if self.model.endswith("001"):
            chunk = chunk.replace("\n", " ")
        return self.client.embeddings.create(**self._params(chunk)).data[0].embedding

    def _params(self, chunk: str) -> dict:
        return {"input": chunk, "model": self.model}
```

- `DEFAULT_MODEL = 'text-embedding-3-small'` class-attribute instance-attribute
- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `base_url = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `organization = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute

_params(chunk)

Source Code in griptape/drivers/embedding/openai_embedding_driver.py

```
def _params(self, chunk: str) -> dict:
    return {"input": chunk, "model": self.model}
```

client()

Source Code in griptape/drivers/embedding/openai_embedding_driver.py

```
@lazy_property()
def client(self) -> openai.OpenAI:
    return openai.OpenAI(api_key=self.api_key, base_url=self.base_url,␣
↪organization=self.organization)
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/openai_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    # Address a performance issue in older ada models
    # https://github.com/openai/openai-python/issues/418#issuecomment-1525939500
    if self.model.endswith("001"):
        chunk = chunk.replace("\n", " ")
    return self.client.embeddings.create(**self._params(chunk)).data[0].embedding
```

voyageai

- `__all__ = ['VoyageAiEmbeddingDriver']` module-attribute

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="voyageai.mdx#griptape.drivers.embedding.voyageai.VoyageAiEmbeddingDriver.model" title="model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.voyageai.VoyageAiEmbeddingDriver.model)"><code class="language-python">	*str*	VoyageAI embedding model name. Defaults to *voyage-large-2*.
<a href="voyageai.mdx#griptape.drivers.embedding.voyageai.VoyageAiEmbeddingDriver.api_key" title="api_key = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.embedding.voyageai.VoyageAiEmbeddingDriver.api_key)"><code class="language-python">	*Optional[str]*	API key to pass directly. Defaults to *VOYAGE_API_KEY* environment variable.

continues on next page

Table 148 – continued from previous page

Name	Type	Description
<a href="voyageai.mdx#griptape.drivers.embedding.voyageai.VoyageAiEmbeddingDriver.tokenizer" title="tokenizer = field(default=Factory(lambda self: VoyageAiTokenizer(model=self.model, api_key=self.api_key), takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.drivers.embedding.voyageai.VoyageAiEmbeddingDriver.tokenizer)"><code class="language-python">	<a href="../../tokenizers#griptape.tokenizers.VoyageAiTokenizer" title="VoyageAiTokenizer (griptape.tokenizers.VoyageAiTokenizer)"><code class="language-python">VoyageAiTokenizer</code>	Optionally provide custom *VoyageAiTokenizer*.
<a href="voyageai.mdx#griptape.drivers.embedding.voyageai.VoyageAiEmbeddingDriver.client" title="client() (griptape.drivers.embedding.voyageai.VoyageAiEmbeddingDriver.client)"><code class="language-python">	*Any*	Optionally provide custom VoyageAI *Client*.

continues on next page

Table 148 – continued from previous page

Name	Type	Description
<a href="voyageai.mdx#griptape.drivers.embedding.voyageai.VoyageAiEmbeddingDriver.input_type" title="input_type = field(default='document', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.voyageai.VoyageAiEmbeddingDriver.input_type)"><code class="language-python">	*str*	VoyageAI input type. Defaults to *document*.

Source Code in griptape/drivers/embedding/voyageai_embedding_driver.py

```
@define
class VoyageAiEmbeddingDriver(BaseEmbeddingDriver):
    """VoyageAI Embedding Driver.

    Attributes:
        model: VoyageAI embedding model name. Defaults to `voyage-large-2`.
        api_key: API key to pass directly. Defaults to `VOYAGE_API_KEY` environment␣
↪variable.
        tokenizer: Optionally provide custom `VoyageAiTokenizer`.
        client: Optionally provide custom VoyageAI `Client`.
        input_type: VoyageAI input type. Defaults to `document`.
    """

    DEFAULT_MODEL = "voyage-large-2"

    model: str = field(default=DEFAULT_MODEL, kw_only=True, metadata={"serializable":␣
↪True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    tokenizer: VoyageAiTokenizer = field(
        default=Factory(lambda self: VoyageAiTokenizer(model=self.model, api_key=self.
↪api_key), takes_self=True),
        kw_only=True,
    )
    input_type: str = field(default="document", kw_only=True, metadata={"serializable
↪": True})
    _client: Optional[Client] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})
```

(continues on next page)

(continued from previous page)

```
    @lazy_property()
    def client(self) -> Any:
        return import_optional_dependency("voyageai").Client(api_key=self.api_key)

    def try_embed_artifact(self, artifact: TextArtifact | ImageArtifact, **kwargs) ->␣
↪list[float]:
        if isinstance(artifact, TextArtifact):
            return self.try_embed_chunk(artifact.value, **kwargs)
        pil_image = import_optional_dependency("PIL.Image")

        return self.client.multimodal_embed([[pil_image.open(BytesIO(artifact.
↪value))]], model=self.model).embeddings[0]

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        return self.client.embed([chunk], model=self.model, input_type=self.input_
↪type).embeddings[0]
```

- `DEFAULT_MODEL = 'voyage-large-2'` class-attribute instance-attribute
- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `input_type = field(default='document', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: VoyageAiTokenizer(model=self.model, api_key=self.api_key), takes_self=True), kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/voyageai_embedding_driver.py

```
@lazy_property()
def client(self) -> Any:
    return import_optional_dependency("voyageai").Client(api_key=self.api_key)
```

try_embed_artifact(artifact, **kwargs)

Source Code in griptape/drivers/embedding/voyageai_embedding_driver.py

```
def try_embed_artifact(self, artifact: TextArtifact | ImageArtifact, **kwargs) ->␣
↪list[float]:
    if isinstance(artifact, TextArtifact):
        return self.try_embed_chunk(artifact.value, **kwargs)
    pil_image = import_optional_dependency("PIL.Image")

    return self.client.multimodal_embed([[pil_image.open(BytesIO(artifact.value))]],␣
↪model=self.model).embeddings[0]
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/voyageai_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    return self.client.embed([chunk], model=self.model, input_type=self.input_type).
↪embeddings[0]
```

voyageai_embedding_driver

Bases: *BaseEmbeddingDriver* “)

Attributes

Name	Type	Description
<a href="voyageai_embedding_driver.mdx#griptape.drivers.embedding.voyageai_embedding_driver.VoyageAiEmbeddingDriver.model" title="model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.voyageai_embedding_driver.VoyageAiEmbeddingDriver.model)"><code class="language-python">	*str*	VoyageAI embedding model name. Defaults to *voyage-large-2*.

continues on next page

Table 149 – continued from previous page

Name	Type	Description
<a href="voyageai_embedding_driver.mdx#griptape.drivers.embedding.voyageai_embedding_driver.VoyageAiEmbeddingDriver.api_key" title="api_key = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.embedding.voyageai_embedding_driver.VoyageAiEmbeddingDriver.api_key)"><code class="language-python">	*Optional[str]*	API key to pass directly. Defaults to *VOYAGE_API_KEY* environment variable.
<a href="voyageai_embedding_driver.mdx#griptape.drivers.embedding.voyageai_embedding_driver.VoyageAiEmbeddingDriver.tokenizer" title="tokenizer = field(default=Factory(lambda self: VoyageAiTokenizer(model=self.model, api_key=self.api_key), takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.drivers.embedding.voyageai_embedding_driver.VoyageAiEmbeddingDriver.tokenizer)"><code class="language-python">	<a href="../../tokenizers#griptape.tokenizers.VoyageAiTokenizer" title="VoyageAiTokenizer (griptape.tokenizers.VoyageAiTokenizer)"><code class="language-python">VoyageAiTokenizer</code>	Optionally provide custom *VoyageAiTokenizer*.

continues on next page

Table 149 – continued from previous page

Name	Type	Description
<a href="voyageai_embedding_driver.mdx#griptape.drivers.embedding.voyageai_embedding_driver.VoyageAiEmbeddingDriver.client" title="client() (griptape.drivers.embedding.voyageai_embedding_driver.VoyageAiEmbeddingDriver.client)"><code class="language-python">	*Any*	Optionally provide custom VoyageAI *Client*.
<a href="voyageai_embedding_driver.mdx#griptape.drivers.embedding.voyageai_embedding_driver.VoyageAiEmbeddingDriver.input_type" title="input_type = field(default='document', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.embedding.voyageai_embedding_driver.VoyageAiEmbeddingDriver.input_type)"><code class="language-python">	*str*	VoyageAI input type. Defaults to *document*.

Source Code in griptape/drivers/embedding/voyageai_embedding_driver.py

```
@define
class VoyageAiEmbeddingDriver(BaseEmbeddingDriver):
    """VoyageAI Embedding Driver.

    Attributes:
        model: VoyageAI embedding model name. Defaults to `voyage-large-2`.
        api_key: API key to pass directly. Defaults to `VOYAGE_API_KEY` environment␣
↪variable.
        tokenizer: Optionally provide custom `VoyageAiTokenizer`.
        client: Optionally provide custom VoyageAI `Client`.
```

(continues on next page)

(continued from previous page)

```
        input_type: VoyageAI input type. Defaults to `document`.
    """

    DEFAULT_MODEL = "voyage-large-2"

    model: str = field(default=DEFAULT_MODEL, kw_only=True, metadata={"serializable":␣
↪True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    tokenizer: VoyageAiTokenizer = field(
        default=Factory(lambda self: VoyageAiTokenizer(model=self.model, api_key=self.
↪api_key), takes_self=True),
        kw_only=True,
    )
    input_type: str = field(default="document", kw_only=True, metadata={"serializable
↪": True})
    _client: Optional[Client] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> Any:
        return import_optional_dependency("voyageai").Client(api_key=self.api_key)

    def try_embed_artifact(self, artifact: TextArtifact | ImageArtifact, **kwargs) ->␣
↪list[float]:
        if isinstance(artifact, TextArtifact):
            return self.try_embed_chunk(artifact.value, **kwargs)
        pil_image = import_optional_dependency("PIL.Image")

        return self.client.multimodal_embed([[pil_image.open(BytesIO(artifact.
↪value))]], model=self.model).embeddings[0]

    def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
        return self.client.embed([chunk], model=self.model, input_type=self.input_
↪type).embeddings[0]
```

- `DEFAULT_MODEL = 'voyage-large-2'` class-attribute instance-attribute
- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `input_type = field(default='document', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(default=DEFAULT_MODEL, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: VoyageAiTokenizer(model=self.model, api_key=self.api_key), takes_self=True), kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/embedding/voyageai_embedding_driver.py

```
@lazy_property()
def client(self) -> Any:
    return import_optional_dependency("voyageai").Client(api_key=self.api_key)
```

try_embed_artifact(artifact, **kwargs)

Source Code in griptape/drivers/embedding/voyageai_embedding_driver.py

```
def try_embed_artifact(self, artifact: TextArtifact | ImageArtifact, **kwargs) ->␣
↪list[float]:
    if isinstance(artifact, TextArtifact):
        return self.try_embed_chunk(artifact.value, **kwargs)
    pil_image = import_optional_dependency("PIL.Image")

    return self.client.multimodal_embed([[pil_image.open(BytesIO(artifact.value))]],␣
↪model=self.model).embeddings[0]
```

try_embed_chunk(chunk, **kwargs)

Source Code in griptape/drivers/embedding/voyageai_embedding_driver.py

```
def try_embed_chunk(self, chunk: str, **kwargs) -> list[float]:
    return self.client.embed([chunk], model=self.model, input_type=self.input_type).
↪embeddings[0]
```

event_listener

- `__all__ = ['BaseEventListenerDriver']` module-attribute

Bases: *FuturesExecutorMixin* “) , *ExponentialBackoffMixin* ”) , ABC

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
@define
class BaseEventListenerDriver(FuturesExecutorMixin, ExponentialBackoffMixin, ABC):
    batched: bool = field(default=True, kw_only=True)
    batch_size: int = field(default=10, kw_only=True)

    _batch: list[dict] = field(default=Factory(list), kw_only=True)

    @property
    def batch(self) -> list[dict]:
        return self._batch

    def publish_event(self, event: BaseEvent | dict) -> None:
        event_payload = event if isinstance(event, dict) else event.to_dict()

        with self.create_futures_executor() as futures_executor:
            if self.batched:
                self._batch.append(event_payload)
```

(continues on next page)

(continued from previous page)

```
                if len(self.batch) >= self.batch_size:
                    futures_executor.submit(with_contextvars(self._safe_publish_event_
↪payload_batch), self.batch)
                    self._batch = []
            else:
                futures_executor.submit(with_contextvars(self._safe_publish_event_
↪payload), event_payload)

    def flush_events(self) -> None:
        if self.batch:
            with self.create_futures_executor() as futures_executor:
                futures_executor.submit(with_contextvars(self._safe_publish_event_
↪payload_batch), self.batch)
            self._batch = []

    @abstractmethod
    def try_publish_event_payload(self, event_payload: dict) -> None: ...

    @abstractmethod
    def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None: ...

    def _safe_publish_event_payload(self, event_payload: dict) -> None:
        try:
            for attempt in self.retrying():
                with attempt:
                    self.try_publish_event_payload(event_payload)
        except Exception:
            logger.warning("Failed to publish event after %s attempts", self.max_
↪attempts, exc_info=True)

    def _safe_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None:
        try:
            for attempt in self.retrying():
                with attempt:
                    self.try_publish_event_payload_batch(event_payload_batch)
        except Exception:
            logger.warning("Failed to publish event batch after %s attempts", self.
↪max_attempts, exc_info=True)
```

- `_batch = field(default=Factory(list), kw_only=True)` class-attribute instance-attribute
- `batch` property
- `batch_size = field(default=10, kw_only=True)` class-attribute instance-attribute
- `batched = field(default=True, kw_only=True)` class-attribute instance-attribute

_safe_publish_event_payload(event_payload)

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
def _safe_publish_event_payload(self, event_payload: dict) -> None:
    try:
        for attempt in self.retrying():
            with attempt:
                self.try_publish_event_payload(event_payload)
    except Exception:
        logger.warning("Failed to publish event after %s attempts", self.max_attempts,
↪ exc_info=True)
```

_safe_publish_event_payload_batch(event_payload_batch)

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
def _safe_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None:
    try:
        for attempt in self.retrying():
            with attempt:
                self.try_publish_event_payload_batch(event_payload_batch)
    except Exception:
        logger.warning("Failed to publish event batch after %s attempts", self.max_
↪attempts, exc_info=True)
```

flush_events()

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
def flush_events(self) -> None:
    if self.batch:
        with self.create_futures_executor() as futures_executor:
            futures_executor.submit(with_contextvars(self._safe_publish_event_payload_
↪batch), self.batch)
        self._batch = []
```

publish_event(event)

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
def publish_event(self, event: BaseEvent | dict) -> None:
    event_payload = event if isinstance(event, dict) else event.to_dict()

    with self.create_futures_executor() as futures_executor:
        if self.batched:
            self._batch.append(event_payload)
            if len(self.batch) >= self.batch_size:
                futures_executor.submit(with_contextvars(self._safe_publish_event_
↪payload_batch), self.batch)
                self._batch = []
        else:
```

(continues on next page)

(continued from previous page)

```
            futures_executor.submit(with_contextvars(self._safe_publish_event_
↪payload), event_payload)
```

try_publish_event_payload(event_payload)abstractmethod

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
@abstractmethod
def try_publish_event_payload(self, event_payload: dict) -> None: ...
```

try_publish_event_payload_batch(event_payload_batch)abstractmethod

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
@abstractmethod
def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None: ..
↪.
```

amazon_sqs

- `__all__ = ['AmazonSqsEventListenerDriver']` module-attribute

Bases: *BaseEventListenerDriver* “)

Source Code in griptape/drivers/event_listener/amazon_sqs_event_listener_driver.py

```
@define
class AmazonSqsEventListenerDriver(BaseEventListenerDriver):
    queue_url: str = field(kw_only=True)
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    _client: Optional[SQSClient] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> SQSClient:
        return self.session.client("sqs")

    def try_publish_event_payload(self, event_payload: dict) -> None:
        self.client.send_message(QueueUrl=self.queue_url, MessageBody=json.
↪dumps(event_payload))

    def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None:
        entries: Sequence[SendMessageBatchRequestEntryTypeDef] = [
            {"Id": str(event_payload["id"]), "MessageBody": json.dumps(event_payload)}
            for event_payload in event_payload_batch
        ]

        self.client.send_message_batch(QueueUrl=self.queue_url, Entries=entries)
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute

- `queue_url = field(kw_only=True)` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/event_listener/amazon_sqs_event_listener_driver.py

```
@lazy_property()
def client(self) -> SQSClient:
    return self.session.client("sqs")
```

try_publish_event_payload(event_payload)

Source Code in griptape/drivers/event_listener/amazon_sqs_event_listener_driver.py

```
def try_publish_event_payload(self, event_payload: dict) -> None:
    self.client.send_message(QueueUrl=self.queue_url, MessageBody=json.dumps(event_
↪payload))
```

try_publish_event_payload_batch(event_payload_batch)

Source Code in griptape/drivers/event_listener/amazon_sqs_event_listener_driver.py

```
def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None:
    entries: Sequence[SendMessageBatchRequestEntryTypeDef] = [
        {"Id": str(event_payload["id"]), "MessageBody": json.dumps(event_payload)}
        for event_payload in event_payload_batch
    ]

    self.client.send_message_batch(QueueUrl=self.queue_url, Entries=entries)
```

amazon_sqs_event_listener_driver

Bases: *BaseEventListenerDriver* “)

Source Code in griptape/drivers/event_listener/amazon_sqs_event_listener_driver.py

```
@define
class AmazonSqsEventListenerDriver(BaseEventListenerDriver):
    queue_url: str = field(kw_only=True)
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    _client: Optional[SQSClient] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> SQSClient:
        return self.session.client("sqs")

    def try_publish_event_payload(self, event_payload: dict) -> None:
        self.client.send_message(QueueUrl=self.queue_url, MessageBody=json.
```

(continues on next page)

(continued from previous page)

```
↪dumps(event_payload))

    def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None:
        entries: Sequence[SendMessageBatchRequestEntryTypeDef] = [
            {"Id": str(event_payload["id"]), "MessageBody": json.dumps(event_payload)}
            for event_payload in event_payload_batch
        ]

        self.client.send_message_batch(QueueUrl=self.queue_url, Entries=entries)
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `queue_url = field(kw_only=True)` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/event_listener/amazon_sqs_event_listener_driver.py

```
@lazy_property()
def client(self) -> SQSClient:
    return self.session.client("sqs")
```

try_publish_event_payload(event_payload)

Source Code in griptape/drivers/event_listener/amazon_sqs_event_listener_driver.py

```
def try_publish_event_payload(self, event_payload: dict) -> None:
    self.client.send_message(QueueUrl=self.queue_url, MessageBody=json.dumps(event_
↪payload))
```

try_publish_event_payload_batch(event_payload_batch)

Source Code in griptape/drivers/event_listener/amazon_sqs_event_listener_driver.py

```
def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None:
    entries: Sequence[SendMessageBatchRequestEntryTypeDef] = [
        {"Id": str(event_payload["id"]), "MessageBody": json.dumps(event_payload)}
        for event_payload in event_payload_batch
    ]

    self.client.send_message_batch(QueueUrl=self.queue_url, Entries=entries)
```

aws_iot_core

- `__all__ = ['AwsIotCoreEventListenerDriver']` module-attribute

Bases: *BaseEventListenerDriver* “)

Source Code in griptape/drivers/event_listener/aws_iot_core_event_listener_driver.py

```
@define
class AwsIotCoreEventListenerDriver(BaseEventListenerDriver):
    iot_endpoint: str = field(kw_only=True)
    topic: str = field(kw_only=True)
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    _client: Optional[IoTDataPlaneClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> IoTDataPlaneClient:
        return self.session.client("iot-data")

    def try_publish_event_payload(self, event_payload: dict) -> None:
        self.client.publish(topic=self.topic, payload=json.dumps(event_payload))

    def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None:
        self.client.publish(topic=self.topic, payload=json.dumps(event_payload_batch))
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `iot_endpoint = field(kw_only=True)` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `topic = field(kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/event_listener/aws_iot_core_event_listener_driver.py

```
@lazy_property()
def client(self) -> IoTDataPlaneClient:
    return self.session.client("iot-data")
```

try_publish_event_payload(event_payload)

Source Code in griptape/drivers/event_listener/aws_iot_core_event_listener_driver.py

```
def try_publish_event_payload(self, event_payload: dict) -> None:
    self.client.publish(topic=self.topic, payload=json.dumps(event_payload))
```

try_publish_event_payload_batch(event_payload_batch)

Source Code in griptape/drivers/event_listener/aws_iot_core_event_listener_driver.py

```
def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None:
    self.client.publish(topic=self.topic, payload=json.dumps(event_payload_batch))
```

aws_iot_core_event_listener_driver

Bases: *BaseEventListenerDriver* “)

Source Code in griptape/drivers/event_listener/aws_iot_core_event_listener_driver.py

```
@define
class AwsIotCoreEventListenerDriver(BaseEventListenerDriver):
    iot_endpoint: str = field(kw_only=True)
    topic: str = field(kw_only=True)
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    _client: Optional[IoTDataPlaneClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> IoTDataPlaneClient:
        return self.session.client("iot-data")

    def try_publish_event_payload(self, event_payload: dict) -> None:
        self.client.publish(topic=self.topic, payload=json.dumps(event_payload))

    def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None:
        self.client.publish(topic=self.topic, payload=json.dumps(event_payload_batch))
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `iot_endpoint = field(kw_only=True)` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `topic = field(kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/event_listener/aws_iot_core_event_listener_driver.py

```
@lazy_property()
def client(self) -> IoTDataPlaneClient:
    return self.session.client("iot-data")
```

try_publish_event_payload(event_payload)

Source Code in griptape/drivers/event_listener/aws_iot_core_event_listener_driver.py

```
def try_publish_event_payload(self, event_payload: dict) -> None:
    self.client.publish(topic=self.topic, payload=json.dumps(event_payload))
```

try_publish_event_payload_batch(event_payload_batch)

Source Code in griptape/drivers/event_listener/aws_iot_core_event_listener_driver.py

```
def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None:
    self.client.publish(topic=self.topic, payload=json.dumps(event_payload_batch))
```

base_event_listener_driver

- `logger = logging.getLogger(__name__)` module-attribute

Bases: *FuturesExecutorMixin* “) , *ExponentialBackoffMixin* ”) , ABC

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
@define
class BaseEventListenerDriver(FuturesExecutorMixin, ExponentialBackoffMixin, ABC):
    batched: bool = field(default=True, kw_only=True)
    batch_size: int = field(default=10, kw_only=True)

    _batch: list[dict] = field(default=Factory(list), kw_only=True)

    @property
    def batch(self) -> list[dict]:
        return self._batch

    def publish_event(self, event: BaseEvent | dict) -> None:
        event_payload = event if isinstance(event, dict) else event.to_dict()

        with self.create_futures_executor() as futures_executor:
            if self.batched:
                self._batch.append(event_payload)
                if len(self.batch) >= self.batch_size:
                    futures_executor.submit(with_contextvars(self._safe_publish_event_
↪payload_batch), self.batch)
                    self._batch = []
            else:
                futures_executor.submit(with_contextvars(self._safe_publish_event_
↪payload), event_payload)

    def flush_events(self) -> None:
        if self.batch:
            with self.create_futures_executor() as futures_executor:
                futures_executor.submit(with_contextvars(self._safe_publish_event_
↪payload_batch), self.batch)
            self._batch = []

    @abstractmethod
    def try_publish_event_payload(self, event_payload: dict) -> None: ...

    @abstractmethod
    def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None: ...
```

(continues on next page)

(continued from previous page)

```
    def _safe_publish_event_payload(self, event_payload: dict) -> None:
        try:
            for attempt in self.retrying():
                with attempt:
                    self.try_publish_event_payload(event_payload)
        except Exception:
            logger.warning("Failed to publish event after %s attempts", self.max_
↪attempts, exc_info=True)

    def _safe_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None:
        try:
            for attempt in self.retrying():
                with attempt:
                    self.try_publish_event_payload_batch(event_payload_batch)
        except Exception:
            logger.warning("Failed to publish event batch after %s attempts", self.
↪max_attempts, exc_info=True)
```

- `_batch = field(default=Factory(list), kw_only=True)` class-attribute instance-attribute
- `batch` property
- `batch_size = field(default=10, kw_only=True)` class-attribute instance-attribute
- `batched = field(default=True, kw_only=True)` class-attribute instance-attribute

_safe_publish_event_payload(event_payload)

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
def _safe_publish_event_payload(self, event_payload: dict) -> None:
    try:
        for attempt in self.retrying():
            with attempt:
                self.try_publish_event_payload(event_payload)
    except Exception:
        logger.warning("Failed to publish event after %s attempts", self.max_attempts,
↪ exc_info=True)
```

_safe_publish_event_payload_batch(event_payload_batch)

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
def _safe_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None:
    try:
        for attempt in self.retrying():
            with attempt:
                self.try_publish_event_payload_batch(event_payload_batch)
    except Exception:
        logger.warning("Failed to publish event batch after %s attempts", self.max_
↪attempts, exc_info=True)
```

flush_events()

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
def flush_events(self) -> None:
    if self.batch:
        with self.create_futures_executor() as futures_executor:
            futures_executor.submit(with_contextvars(self._safe_publish_event_payload_
↪batch), self.batch)
        self._batch = []
```

publish_event(event)

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
def publish_event(self, event: BaseEvent | dict) -> None:
    event_payload = event if isinstance(event, dict) else event.to_dict()

    with self.create_futures_executor() as futures_executor:
        if self.batched:
            self._batch.append(event_payload)
            if len(self.batch) >= self.batch_size:
                futures_executor.submit(with_contextvars(self._safe_publish_event_
↪payload_batch), self.batch)
                self._batch = []
        else:
            futures_executor.submit(with_contextvars(self._safe_publish_event_
↪payload), event_payload)
```

try_publish_event_payload(event_payload)abstractmethod

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
@abstractmethod
def try_publish_event_payload(self, event_payload: dict) -> None: ...
```

try_publish_event_payload_batch(event_payload_batch)abstractmethod

Source Code in griptape/drivers/event_listener/base_event_listener_driver.py

```
@abstractmethod
def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None: ..
↪.
```

griptape_cloud

- `__all__ = ['GriptapeCloudEventListenerDriver']` module-attribute

Bases: *BaseEventListenerDriver* “)

Attributes

Name	Type	Description
<a href="griptape_cloud.mdx#griptape.drivers.event_listener.griptape_cloud.GriptapeCloudEventListenerDriver.base_url" title="base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_ 'https://cloud.griptape.ai')), kw_only=True) class-attribute instance-attribute (griptape.drivers.event_listener.griptape_cloud.GriptapeCloudEventListenerDriver.base_url)"><code class="language-python">	*str*	The base URL of Gen AI Builder. Defaults to the GT_CLOUD_BASE_URL environment variable.
<a href="griptape_cloud.mdx#griptape.drivers.event_listener.griptape_cloud.GriptapeCloudEventListenerDriver.api_key" title="api_key = field(default=Factory(lambda: os.getenv('GT_CLOUD_ kw_only=True) class-attribute instance-attribute (griptape.drivers.event_listener.griptape_cloud.GriptapeCloudEventListenerDriver.api_key)"><code class="language-python">	*Optional[str]*	The API key to authenticate with Gen AI Builder.

continues on next page

Table 150 – continued from previous page

Name	Type	Description
<a href="griptape_cloud.mdx#griptape.drivers.event_listener.griptape_cloud.GriptapeCloudEventListenerDriver.headers" title="headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.drivers.event_listener.griptape_cloud.GriptapeCloudEventListenerDriver.headers)"><code class="language-python">	*dict*	The headers to use when making requests to Gen AI Builder. Defaults to include the Authorization header.
<a href="griptape_cloud.mdx#griptape.drivers.event_listener.griptape_cloud.GriptapeCloudEventListenerDriver.structure_run_id" title="structure_run_id = field(default=Factory(lambda: os.getenv('GT_CLOUD_STRUCTURE_RUN_ID')), kw_only=True) class-attribute instance-attribute (griptape.drivers.event_listener.griptape_cloud.GriptapeCloudEventListenerDriver.structure_run_id)"><code class="language-python">	*Optional[str]*	The ID of the Structure Run to publish events to. Defaults to the GT_CLOUD_STRUCTURE_RUN_ID environment variable.

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
@define
```

(continues on next page)

(continued from previous page)

```python
class GriptapeCloudEventListenerDriver(BaseEventListenerDriver):
    """Driver for publishing events to Gen AI Builder.

    Attributes:
        base_url: The base URL of Gen AI Builder. Defaults to the GT_CLOUD_BASE_URL environment variable.
        api_key: The API key to authenticate with Gen AI Builder.
        headers: The headers to use when making requests to Gen AI Builder. Defaults to include the Authorization header.
        structure_run_id: The ID of the Structure Run to publish events to. Defaults to the GT_CLOUD_STRUCTURE_RUN_ID environment variable.
    """

    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.griptape.ai")),
        kw_only=True,
    )
    api_key: Optional[str] = field(default=Factory(lambda: os.getenv("GT_CLOUD_API_KEY")), kw_only=True)
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"}, takes_self=True),
        kw_only=True,
    )
    structure_run_id: Optional[str] = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_STRUCTURE_RUN_ID")), kw_only=True
    )

    @structure_run_id.validator  # pyright: ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_run_id(self, _: Attribute, structure_run_id: Optional[str]) -> None:
        if structure_run_id is None:
            raise ValueError(
                "structure_run_id must be set either in the constructor or as an environment variable (GT_CLOUD_STRUCTURE_RUN_ID).",
            )

    @api_key.validator  # pyright: ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_api_key(self, _: Attribute, api_key: Optional[str]) -> None:
        if api_key is None:
            raise ValueError(
                "No value was found for the GT_CLOUD_API_KEY environment variable. "
                "This environment variable is required when running in Gen AI Builder for authorization. "
                "You can generate a Gen AI Builder API Key by visiting https://cloud.griptape.ai/keys . "
                "Specify it as an environment variable when creating a Managed Structure in Gen AI Builder."
            )
```

(continues on next page)

(continued from previous page)

```
    def publish_event(self, event: BaseEvent | dict) -> None:
        from griptape.observability.observability import Observability

        event_payload = event.to_dict() if isinstance(event, BaseEvent) else event

        span_id = Observability.get_span_id()
        if span_id is not None:
            event_payload["span_id"] = span_id

        super().publish_event(event_payload)

    def try_publish_event_payload(self, event_payload: dict) -> None:
        self._post_event(self._get_event_request(event_payload))

    def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None:
        self._post_event([self._get_event_request(event_payload) for event_payload in␣
↪event_payload_batch])

    def _get_event_request(self, event_payload: dict) -> dict:
        return {
            "payload": event_payload,
            "timestamp": event_payload.get("timestamp", time.time()),
            "type": event_payload.get("type", "UserEvent"),
        }

    def _post_event(self, json: list[dict] | dict) -> None:
        requests.post(
            url=griptape_cloud_url(self.base_url, f"api/structure-runs/{self.
↪structure_run_id}/events"),
            json=json,
            headers=self.headers,
        ).raise_for_status()
```

- `api_key = field(default=Factory(lambda: os.getenv('GT_CLOUD_API_KEY')), kw_only=True)` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')), kw_only=True)` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `structure_run_id = field(default=Factory(lambda: os.getenv('GT_CLOUD_STRUCTURE_RUN_ID')), kw_only=True)` class-attribute instance-attribute

_get_event_request(event_payload)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
def _get_event_request(self, event_payload: dict) -> dict:
    return {
        "payload": event_payload,
        "timestamp": event_payload.get("timestamp", time.time()),
```

(continues on next page)

(continued from previous page)

```
        "type": event_payload.get("type", "UserEvent"),
    }
```

_post_event(json)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
def _post_event(self, json: list[dict] | dict) -> None:
    requests.post(
        url=griptape_cloud_url(self.base_url, f"api/structure-runs/{self.structure_
↪run_id}/events"),
        json=json,
        headers=self.headers,
    ).raise_for_status()
```

publish_event(event)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
def publish_event(self, event: BaseEvent | dict) -> None:
    from griptape.observability.observability import Observability

    event_payload = event.to_dict() if isinstance(event, BaseEvent) else event

    span_id = Observability.get_span_id()
    if span_id is not None:
        event_payload["span_id"] = span_id

    super().publish_event(event_payload)
```

try_publish_event_payload(event_payload)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
def try_publish_event_payload(self, event_payload: dict) -> None:
    self._post_event(self._get_event_request(event_payload))
```

try_publish_event_payload_batch(event_payload_batch)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None:
    self._post_event([self._get_event_request(event_payload) for event_payload in␣
↪event_payload_batch])
```

validate*api_key(*, api_key)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
@api_key.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_api_key(self, _: Attribute, api_key: Optional[str]) -> None:
    if api_key is None:
        raise ValueError(
            "No value was found for the GT_CLOUD_API_KEY environment variable. "
            "This environment variable is required when running in Gen AI Builder for␣
↪authorization. "
            "You can generate a Gen AI Builder API Key by visiting https://cloud.
↪griptape.ai/keys . "
            "Specify it as an environment variable when creating a Managed Structure␣
↪in Gen AI Builder."
        )
```

validate*run_id(*, structure_run_id)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
@structure_run_id.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_run_id(self, _: Attribute, structure_run_id: Optional[str]) -> None:
    if structure_run_id is None:
        raise ValueError(
            "structure_run_id must be set either in the constructor or as an␣
↪environment variable (GT_CLOUD_STRUCTURE_RUN_ID).",
        )
```

griptape_cloud_event_listener_driver

Bases: *BaseEventListenerDriver* “)

Attributes

Name	Type	Description
<a href="griptape_cloud_event_listener_driver.mdx#griptape.drivers.event_listener.griptape_cloud_event_listener_driver.GriptapeCloudEventListenerDriver.base_url" title="base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_ 'https://cloud.griptape.ai')), kw_only=True) class-attribute instance-attribute (griptape.drivers.event_listener.griptape_cloud_event_listener_driver.GriptapeCloudEventListenerDriver.base_url)"><code class="language-python">	*str*	The base URL of Gen AI Builder. Defaults to the GT_CLOUD_BASE_URL environment variable.
<a href="griptape_cloud_event_listener_driver.mdx#griptape.drivers.event_listener.griptape_cloud_event_listener_driver.GriptapeCloudEventListenerDriver.api_key" title="api_key = field(default=Factory(lambda: os.getenv('GT_CLOUD_ kw_only=True) class-attribute instance-attribute (griptape.drivers.event_listener.griptape_cloud_event_listener_driver.GriptapeCloudEventListenerDriver.api_key)"><code class="language-python">	*Optional[str]*	The API key to authenticate with Gen AI Builder.

continues on next page

Table 151 – continued from previous page

Name	Type	Description
<a href="grip-tape_cloud_event_listener_driver.mdx#grip-tape.drivers.event_listener.grip-tape_cloud_event_listener_driver.GripTapeCloudEventListenerDriver.headers" title="headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True) class-attribute instance-attribute (grip-tape.drivers.event_listener.grip-tape_cloud_event_listener_driver.GripTapeCloudEventListenerDriver.headers)"><code class="language-python">	*dict*	The headers to use when making requests to Gen AI Builder. Defaults to include the Authorization header.

continues on next page

Table 151 – continued from previous page

Name	Type	Description
<a href="griptape_cloud_event_listener_driver.mdx#griptape.drivers.event_listener.griptape_cloud_event_listener_driver.GriptapeCloudEventListenerDriver.structure_run_id" title="structure_run_id = field(default=Factory(lambda: os.getenv('GT_CLOUD_STRUCTURE_RUN_ID')), kw_only=True) class-attribute instance-attribute (griptape.drivers.event_listener.griptape_cloud_event_listener_driver.GriptapeCloudEventListenerDriver.structure_run_id)"><code class="language-python">	*Optional[str]*	The ID of the Structure Run to publish events to. Defaults to the GT_CLOUD_STRUCTURE_RUN_ID environment variable.

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
@define
class GriptapeCloudEventListenerDriver(BaseEventListenerDriver):
    """Driver for publishing events to Gen AI Builder.

    Attributes:
        base_url: The base URL of Gen AI Builder. Defaults to the GT_CLOUD_BASE_URL␣
↪environment variable.
        api_key: The API key to authenticate with Gen AI Builder.
        headers: The headers to use when making requests to Gen AI Builder. Defaults␣
↪to include the Authorization header.
        structure_run_id: The ID of the Structure Run to publish events to. Defaults␣
↪to the GT_CLOUD_STRUCTURE_RUN_ID environment variable.
    """

    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
        kw_only=True,
    )
    api_key: Optional[str] = field(default=Factory(lambda: os.getenv("GT_CLOUD_API_KEY
↪")), kw_only=True)
```

(continues on next page)

(continued from previous page)

```
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        kw_only=True,
    )
    structure_run_id: Optional[str] = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_STRUCTURE_RUN_ID")), kw_only=True
    )

    @structure_run_id.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_run_id(self, _: Attribute, structure_run_id: Optional[str]) -> None:
        if structure_run_id is None:
            raise ValueError(
                "structure_run_id must be set either in the constructor or as an␣
↪environment variable (GT_CLOUD_STRUCTURE_RUN_ID).",
            )

    @api_key.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_api_key(self, _: Attribute, api_key: Optional[str]) -> None:
        if api_key is None:
            raise ValueError(
                "No value was found for the GT_CLOUD_API_KEY environment variable. "
                "This environment variable is required when running in Gen AI Builder␣
↪for authorization. "
                "You can generate a Gen AI Builder API Key by visiting https://cloud.
↪griptape.ai/keys . "
                "Specify it as an environment variable when creating a Managed␣
↪Structure in Gen AI Builder."
            )

    def publish_event(self, event: BaseEvent | dict) -> None:
        from griptape.observability.observability import Observability

        event_payload = event.to_dict() if isinstance(event, BaseEvent) else event

        span_id = Observability.get_span_id()
        if span_id is not None:
            event_payload["span_id"] = span_id

        super().publish_event(event_payload)

    def try_publish_event_payload(self, event_payload: dict) -> None:
        self._post_event(self._get_event_request(event_payload))

    def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None:
        self._post_event([self._get_event_request(event_payload) for event_payload in␣
↪event_payload_batch])

    def _get_event_request(self, event_payload: dict) -> dict:
```

(continues on next page)

(continued from previous page)

```
        return {
            "payload": event_payload,
            "timestamp": event_payload.get("timestamp", time.time()),
            "type": event_payload.get("type", "UserEvent"),
        }

    def _post_event(self, json: list[dict] | dict) -> None:
        requests.post(
            url=griptape_cloud_url(self.base_url, f"api/structure-runs/{self.
↪structure_run_id}/events"),
            json=json,
            headers=self.headers,
        ).raise_for_status()
```

- `api_key = field(default=Factory(lambda: os.getenv('GT_CLOUD_API_KEY')), kw_only=True)` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')), kw_only=True)` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `structure_run_id = field(default=Factory(lambda: os.getenv('GT_CLOUD_STRUCTURE_RUN_ID')), kw_only=True)` class-attribute instance-attribute

_get_event_request(event_payload)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
def _get_event_request(self, event_payload: dict) -> dict:
    return {
        "payload": event_payload,
        "timestamp": event_payload.get("timestamp", time.time()),
        "type": event_payload.get("type", "UserEvent"),
    }
```

_post_event(json)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
def _post_event(self, json: list[dict] | dict) -> None:
    requests.post(
        url=griptape_cloud_url(self.base_url, f"api/structure-runs/{self.structure_
↪run_id}/events"),
        json=json,
        headers=self.headers,
    ).raise_for_status()
```

publish_event(event)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
def publish_event(self, event: BaseEvent | dict) -> None:
    from griptape.observability.observability import Observability

    event_payload = event.to_dict() if isinstance(event, BaseEvent) else event

    span_id = Observability.get_span_id()
    if span_id is not None:
        event_payload["span_id"] = span_id

    super().publish_event(event_payload)
```

try_publish_event_payload(event_payload)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
def try_publish_event_payload(self, event_payload: dict) -> None:
    self._post_event(self._get_event_request(event_payload))
```

try_publish_event_payload_batch(event_payload_batch)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None:
    self._post_event([self._get_event_request(event_payload) for event_payload in␣
↪event_payload_batch])
```

validate*api_key(*, api_key)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
@api_key.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_api_key(self, _: Attribute, api_key: Optional[str]) -> None:
    if api_key is None:
        raise ValueError(
            "No value was found for the GT_CLOUD_API_KEY environment variable. "
            "This environment variable is required when running in Gen AI Builder for␣
↪authorization. "
            "You can generate a Gen AI Builder API Key by visiting https://cloud.
↪griptape.ai/keys . "
            "Specify it as an environment variable when creating a Managed Structure␣
↪in Gen AI Builder."
        )
```

validate*run_id(*, structure_run_id)

Source Code in griptape/drivers/event_listener/griptape_cloud_event_listener_driver.py

```
@structure_run_id.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_run_id(self, _: Attribute, structure_run_id: Optional[str]) -> None:
    if structure_run_id is None:
        raise ValueError(
            "structure_run_id must be set either in the constructor or as an␣
↪environment variable (GT_CLOUD_STRUCTURE_RUN_ID).",
        )
```

pusher

- `__all__ = ['PusherEventListenerDriver']` module-attribute

Bases: *BaseEventListenerDriver* “)

Source Code in griptape/drivers/event_listener/pusher_event_listener_driver.py

```
@define
class PusherEventListenerDriver(BaseEventListenerDriver):
    app_id: str = field(kw_only=True, metadata={"serializable": True})
    key: str = field(kw_only=True, metadata={"serializable": True})
    secret: str = field(kw_only=True, metadata={"serializable": False})
    cluster: str = field(kw_only=True, metadata={"serializable": True})
    channel: str = field(kw_only=True, metadata={"serializable": True})
    event_name: str = field(kw_only=True, metadata={"serializable": True})
    ssl: bool = field(default=True, kw_only=True, metadata={"serializable": True})
    _client: Optional[Pusher] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> Pusher:
        return import_optional_dependency("pusher").Pusher(
            app_id=self.app_id,
            key=self.key,
            secret=self.secret,
            cluster=self.cluster,
            ssl=self.ssl,
        )

    def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None:
        data = [
            {"channel": self.channel, "name": self.event_name, "data": event_payload}
            for event_payload in event_payload_batch
        ]

        self.client.trigger_batch(data)

    def try_publish_event_payload(self, event_payload: dict) -> None:
        self.client.trigger(channels=self.channel, event_name=self.event_name,␣
↪data=event_payload)
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute

- `app_id = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `channel = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `cluster = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `event_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `secret = field(kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `ssl = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/event_listener/pusher_event_listener_driver.py

```
@lazy_property()
def client(self) -> Pusher:
    return import_optional_dependency("pusher").Pusher(
        app_id=self.app_id,
        key=self.key,
        secret=self.secret,
        cluster=self.cluster,
        ssl=self.ssl,
    )
```

try_publish_event_payload(event_payload)

Source Code in griptape/drivers/event_listener/pusher_event_listener_driver.py

```
def try_publish_event_payload(self, event_payload: dict) -> None:
    self.client.trigger(channels=self.channel, event_name=self.event_name, data=event_
↪payload)
```

try_publish_event_payload_batch(event_payload_batch)

Source Code in griptape/drivers/event_listener/pusher_event_listener_driver.py

```
def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None:
    data = [
        {"channel": self.channel, "name": self.event_name, "data": event_payload}
        for event_payload in event_payload_batch
    ]

    self.client.trigger_batch(data)
```

pusher_event_listener_driver

Bases: *BaseEventListenerDriver* “)

Source Code in griptape/drivers/event_listener/pusher_event_listener_driver.py

```
@define
class PusherEventListenerDriver(BaseEventListenerDriver):
    app_id: str = field(kw_only=True, metadata={"serializable": True})
    key: str = field(kw_only=True, metadata={"serializable": True})
    secret: str = field(kw_only=True, metadata={"serializable": False})
    cluster: str = field(kw_only=True, metadata={"serializable": True})
    channel: str = field(kw_only=True, metadata={"serializable": True})
    event_name: str = field(kw_only=True, metadata={"serializable": True})
    ssl: bool = field(default=True, kw_only=True, metadata={"serializable": True})
    _client: Optional[Pusher] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> Pusher:
        return import_optional_dependency("pusher").Pusher(
            app_id=self.app_id,
            key=self.key,
            secret=self.secret,
            cluster=self.cluster,
            ssl=self.ssl,
        )

    def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None:
        data = [
            {"channel": self.channel, "name": self.event_name, "data": event_payload}
            for event_payload in event_payload_batch
        ]

        self.client.trigger_batch(data)

    def try_publish_event_payload(self, event_payload: dict) -> None:
        self.client.trigger(channels=self.channel, event_name=self.event_name,␣
↪data=event_payload)
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `app_id = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `channel = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `cluster = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `event_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

- `secret = field(kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `ssl = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/event_listener/pusher_event_listener_driver.py

```
@lazy_property()
def client(self) -> Pusher:
    return import_optional_dependency("pusher").Pusher(
        app_id=self.app_id,
        key=self.key,
        secret=self.secret,
        cluster=self.cluster,
        ssl=self.ssl,
    )
```

try_publish_event_payload(event_payload)

Source Code in griptape/drivers/event_listener/pusher_event_listener_driver.py

```
def try_publish_event_payload(self, event_payload: dict) -> None:
    self.client.trigger(channels=self.channel, event_name=self.event_name, data=event_
↪payload)
```

try_publish_event_payload_batch(event_payload_batch)

Source Code in griptape/drivers/event_listener/pusher_event_listener_driver.py

```
def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None:
    data = [
        {"channel": self.channel, "name": self.event_name, "data": event_payload}
        for event_payload in event_payload_batch
    ]

    self.client.trigger_batch(data)
```

webhook

- `__all__ = ['WebhookEventListenerDriver']` module-attribute

Bases: *BaseEventListenerDriver* “)

Source Code in griptape/drivers/event_listener/webhook_event_listener_driver.py

```
@define
class WebhookEventListenerDriver(BaseEventListenerDriver):
    webhook_url: str = field(kw_only=True)
    headers: Optional[dict] = field(default=None, kw_only=True)

    def try_publish_event_payload(self, event_payload: dict) -> None:
```

(continues on next page)

(continued from previous page)

```
        response = requests.post(url=self.webhook_url, json=event_payload,␣
↪headers=self.headers)
        response.raise_for_status()

    def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None:
        response = requests.post(url=self.webhook_url, json=event_payload_batch,␣
↪headers=self.headers)
        response.raise_for_status()
```

- `headers = field(default=None, kw_only=True)` class-attribute instance-attribute
- `webhook_url = field(kw_only=True)` class-attribute instance-attribute

try_publish_event_payload(event_payload)

Source Code in griptape/drivers/event_listener/webhook_event_listener_driver.py

```
def try_publish_event_payload(self, event_payload: dict) -> None:
    response = requests.post(url=self.webhook_url, json=event_payload, headers=self.
↪headers)
    response.raise_for_status()
```

try_publish_event_payload_batch(event_payload_batch)

Source Code in griptape/drivers/event_listener/webhook_event_listener_driver.py

```
def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None:
    response = requests.post(url=self.webhook_url, json=event_payload_batch,␣
↪headers=self.headers)
    response.raise_for_status()
```

webhook_event_listener_driver

Bases: *BaseEventListenerDriver* “)

Source Code in griptape/drivers/event_listener/webhook_event_listener_driver.py

```
@define
class WebhookEventListenerDriver(BaseEventListenerDriver):
    webhook_url: str = field(kw_only=True)
    headers: Optional[dict] = field(default=None, kw_only=True)

    def try_publish_event_payload(self, event_payload: dict) -> None:
        response = requests.post(url=self.webhook_url, json=event_payload,␣
↪headers=self.headers)
        response.raise_for_status()

    def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) ->␣
↪None:
        response = requests.post(url=self.webhook_url, json=event_payload_batch,␣
↪headers=self.headers)
        response.raise_for_status()
```

- `headers = field(default=None, kw_only=True)` class-attribute instance-attribute
- `webhook_url = field(kw_only=True)` class-attribute instance-attribute

try_publish_event_payload(event_payload)

Source Code in griptape/drivers/event_listener/webhook_event_listener_driver.py

```
def try_publish_event_payload(self, event_payload: dict) -> None:
    response = requests.post(url=self.webhook_url, json=event_payload, headers=self.
↪headers)
    response.raise_for_status()
```

try_publish_event_payload_batch(event_payload_batch)

Source Code in griptape/drivers/event_listener/webhook_event_listener_driver.py

```
def try_publish_event_payload_batch(self, event_payload_batch: list[dict]) -> None:
    response = requests.post(url=self.webhook_url, json=event_payload_batch,␣
↪headers=self.headers)
    response.raise_for_status()
```

file_manager

- `__all__ = ['BaseFileManagerDriver']` module-attribute

Bases:

ABC

Attributes

Name	Type	Description
default_loader	``	The default loader to use for loading file contents into artifacts.
loaders	``	Dictionary of file extension specific loaders to use for loading file contents into artifacts.

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
@define
class BaseFileManagerDriver(ABC):
    """BaseFileManagerDriver can be used to list, load, and save files.

    Attributes:
        default_loader: The default loader to use for loading file contents into␣
↪artifacts.
        loaders: Dictionary of file extension specific loaders to use for loading␣
↪file contents into artifacts.
```

(continues on next page)

(continued from previous page)

```
    """

    _workdir: str = field(kw_only=True, alias="workdir")
    encoding: Optional[str] = field(default=None, kw_only=True)

    @property
    @abstractmethod
    def workdir(self) -> str: ...

    @workdir.setter
    @abstractmethod
    def workdir(self, value: str) -> None: ...

    def list_files(self, path: str) -> TextArtifact:
        entries = self.try_list_files(path)
        return TextArtifact("\n".join(list(entries)))

    @abstractmethod
    def try_list_files(self, path: str) -> list[str]: ...

    def load_file(self, path: str) -> BlobArtifact | TextArtifact:
        if self.encoding is None:
            return BlobArtifact(self.try_load_file(path))
        return TextArtifact(self.try_load_file(path).decode(encoding=self.encoding),␣
↪encoding=self.encoding)

    @abstractmethod
    def try_load_file(self, path: str) -> bytes: ...

    def save_file(self, path: str, value: bytes | str) -> InfoArtifact:
        if isinstance(value, str):
            value = value.encode() if self.encoding is None else value.
↪encode(encoding=self.encoding)
        elif isinstance(value, (bytearray, memoryview)):
            raise ValueError(f"Unsupported type: {type(value)}")

        location = self.try_save_file(path, value)

        return InfoArtifact(f"Successfully saved file at: {location}")

    @abstractmethod
    def try_save_file(self, path: str, value: bytes) -> str: ...

    def load_artifact(self, path: str) -> BaseArtifact:
        response = self.try_load_file(path)
        return BaseArtifact.from_json(
            response.decode() if self.encoding is None else response.
↪decode(encoding=self.encoding)
        )

    def save_artifact(self, path: str, artifact: BaseArtifact) -> InfoArtifact:
        artifact_json = artifact.to_json()
```

(continues on next page)

(continued from previous page)

```
        value = artifact_json.encode() if self.encoding is None else artifact_json.
↪encode(encoding=self.encoding)

        location = self.try_save_file(path, value)

        return InfoArtifact(f"Successfully saved artifact at: {location}")
```

- `_workdir = field(kw_only=True, alias='workdir')` class-attribute instance-attribute
- `encoding = field(default=None, kw_only=True)` class-attribute instance-attribute
- `workdir` abstractmethod property writable

list_files(path)

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
def list_files(self, path: str) -> TextArtifact:
    entries = self.try_list_files(path)
    return TextArtifact("\n".join(list(entries)))
```

load_artifact(path)

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
def load_artifact(self, path: str) -> BaseArtifact:
    response = self.try_load_file(path)
    return BaseArtifact.from_json(
        response.decode() if self.encoding is None else response.decode(encoding=self.
↪encoding)
    )
```

load_file(path)

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
def load_file(self, path: str) -> BlobArtifact | TextArtifact:
    if self.encoding is None:
        return BlobArtifact(self.try_load_file(path))
    return TextArtifact(self.try_load_file(path).decode(encoding=self.encoding),␣
↪encoding=self.encoding)
```

save_artifact(path, artifact)

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
def save_artifact(self, path: str, artifact: BaseArtifact) -> InfoArtifact:
    artifact_json = artifact.to_json()
    value = artifact_json.encode() if self.encoding is None else artifact_json.
↪encode(encoding=self.encoding)

    location = self.try_save_file(path, value)
```

(continues on next page)

(continued from previous page)

```
    return InfoArtifact(f"Successfully saved artifact at: {location}")
```

save_file(path, value)

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
def save_file(self, path: str, value: bytes | str) -> InfoArtifact:
    if isinstance(value, str):
        value = value.encode() if self.encoding is None else value.
↪encode(encoding=self.encoding)
    elif isinstance(value, (bytearray, memoryview)):
        raise ValueError(f"Unsupported type: {type(value)}")

    location = self.try_save_file(path, value)

    return InfoArtifact(f"Successfully saved file at: {location}")
```

try_list_files(path)abstractmethod

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
@abstractmethod
def try_list_files(self, path: str) -> list[str]: ...
```

try_load_file(path)abstractmethod

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
@abstractmethod
def try_load_file(self, path: str) -> bytes: ...
```

try_save_file(path, value)abstractmethod

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
@abstractmethod
def try_save_file(self, path: str, value: bytes) -> str: ...
```

amazon_s3

- `__all__ = ['AmazonS3FileManagerDriver']` module-attribute

Bases: *BaseFileManagerDriver* “)

Attributes

Name	Type	Description
<a href="amazon_s3.mdx#griptape.drivers.file_manager.amazon_s3.AmazonS3FileManagerDriver.session" title="session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True) class-attribute instance-attribute (griptape.drivers.file_manager.amazon_s3.AmazonS3FileManagerDriver.session)"><code class="language-python">	*Session*	The boto3 session to use for S3 operations.
<a href="amazon_s3.mdx#griptape.drivers.file_manager.amazon_s3.AmazonS3FileManagerDriver.bucket" title="bucket = field(kw_only=True) class-attribute instance-attribute (griptape.drivers.file_manager.amazon_s3.AmazonS3FileManagerDriver.bucket)"><code class="language-python">	*str*	The name of the S3 bucket.
<a href="amazon_s3.mdx#griptape.drivers.file_manager.amazon_s3.AmazonS3FileManagerDriver.workdir" title="workdir property writable (griptape.drivers.file_manager.amazon_s3.AmazonS3FileManagerDriver.workdir)"><code class="language-python">	*str*	The absolute working directory (must start with "/"). List, load, and save operations will be performed relative to this directory.

continues on next page

Table 153 – continued from previous page

Name	Type	Description
<a href="amazon_s3.mdx#griptape.drivers.file_manager.amazon_s3.AmazonS3FileManagerDriver.client" title="client() (griptape.drivers.file_manager.amazon_s3.AmazonS3FileManagerDriver.client)"><code class="language-python">	*S3Client*	The S3 client to use for S3 operations.

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
@define
class AmazonS3FileManagerDriver(BaseFileManagerDriver):
    """AmazonS3FileManagerDriver can be used to list, load, and save files in an␣
↪Amazon S3 bucket.

    Attributes:
        session: The boto3 session to use for S3 operations.
        bucket: The name of the S3 bucket.
        workdir: The absolute working directory (must start with "/"). List, load,␣
↪and save
            operations will be performed relative to this directory.
        client: The S3 client to use for S3 operations.
    """

    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    bucket: str = field(kw_only=True)
    _workdir: str = field(default="/", kw_only=True, alias="workdir")
    _client: Optional[S3Client] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @property
    def workdir(self) -> str:
        if self._workdir.startswith("/"):
            return self._workdir
        return f"/{self._workdir}"

    @workdir.setter
    def workdir(self, value: str) -> None:
        self._workdir = value

    @lazy_property()
    def client(self) -> S3Client:
        return self.session.client("s3")
```

(continues on next page)

(continued from previous page)

```
    def try_list_files(self, path: str) -> list[str]:
        full_key = self._to_dir_full_key(path)
        files_and_dirs = self._list_files_and_dirs(full_key)
        if len(files_and_dirs) == 0:
            if len(self._list_files_and_dirs(full_key.rstrip("/"), max_items=1)) > 0:
                raise NotADirectoryError
            raise FileNotFoundError
        return files_and_dirs

    def try_load_file(self, path: str) -> bytes:
        botocore = import_optional_dependency("botocore")
        full_key = self._to_full_key(path)

        if self._is_a_directory(full_key):
            raise IsADirectoryError

        try:
            response = self.client.get_object(Bucket=self.bucket, Key=full_key)
            return response["Body"].read()
        except botocore.exceptions.ClientError as e:
            if e.response["Error"]["Code"] in {"NoSuchKey", "404"}:
                raise FileNotFoundError from e
            raise e

    def try_save_file(self, path: str, value: bytes) -> str:
        full_key = self._to_full_key(path)
        if self._is_a_directory(full_key):
            raise IsADirectoryError
        self.client.put_object(Bucket=self.bucket, Key=full_key, Body=value)
        return f"s3://{self.bucket}/{full_key}"

    def _to_full_key(self, path: str) -> str:
        path = path.lstrip("/")
        full_key = f"{self.workdir}/{path}"
        # Need to keep the trailing slash if it was there,
        # because it means the path is a directory.
        ended_with_slash = path.endswith("/")

        full_key = self._normpath(full_key)

        if ended_with_slash:
            full_key += "/"
        return full_key.lstrip("/")

    def _to_dir_full_key(self, path: str) -> str:
        full_key = self._to_full_key(path)
        # S3 "directories" always end with a slash, except for the root.
        if full_key != "" and not full_key.endswith("/"):
            full_key += "/"
        return full_key

    def _list_files_and_dirs(self, full_key: str, **kwargs) -> list[str]:
```

(continues on next page)

(continued from previous page)

```
        max_items = kwargs.get("max_items")
        pagination_config: PaginatorConfigTypeDef = {}
        if max_items is not None:
            pagination_config["MaxItems"] = max_items

        paginator = self.client.get_paginator("list_objects_v2")
        pages = paginator.paginate(
            Bucket=self.bucket,
            Prefix=full_key,
            Delimiter="/",
            PaginationConfig=pagination_config,
        )
        files_and_dirs = []
        for page in pages:
            for obj in page.get("CommonPrefixes", []):
                prefix = obj.get("Prefix", "")
                directory = prefix[len(full_key) :].rstrip("/")
                files_and_dirs.append(directory)

            for obj in page.get("Contents", []):
                key = obj.get("Key", "")
                file = key[len(full_key) :]
                files_and_dirs.append(file)
        return files_and_dirs

    def _is_a_directory(self, full_key: str) -> bool:
        botocore = import_optional_dependency("botocore")
        if full_key == "" or full_key.endswith("/"):
            return True

        try:
            self.client.head_object(Bucket=self.bucket, Key=full_key)
        except botocore.exceptions.ClientError as e:
            if e.response["Error"]["Code"] in {"NoSuchKey", "404"}:
                return len(self._list_files_and_dirs(full_key, max_items=1)) > 0
            raise e

        return False

    def _normpath(self, path: str) -> str:
        unix_path = path.replace("\", "/")
        parts = unix_path.split("/")
        stack = []

        for part in parts:
            if part == "" or part == ".":
                continue
            if part == "..":
                if stack:
                    stack.pop()
            else:
                stack.append(part)
```

(continues on next page)

(continued from previous page)

```
        return "/".join(stack)
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `_workdir = field(default='/', kw_only=True, alias='workdir')` class-attribute instance-attribute
- `bucket = field(kw_only=True)` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `workdir` property writable

_is_a_directory(full_key)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def _is_a_directory(self, full_key: str) -> bool:
    botocore = import_optional_dependency("botocore")
    if full_key == "" or full_key.endswith("/"):
        return True

    try:
        self.client.head_object(Bucket=self.bucket, Key=full_key)
    except botocore.exceptions.ClientError as e:
        if e.response["Error"]["Code"] in {"NoSuchKey", "404"}:
            return len(self._list_files_and_dirs(full_key, max_items=1)) > 0
        raise e

    return False
```

_list_files_and_dirs(full_key, **kwargs)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def _list_files_and_dirs(self, full_key: str, **kwargs) -> list[str]:
    max_items = kwargs.get("max_items")
    pagination_config: PaginatorConfigTypeDef = {}
    if max_items is not None:
        pagination_config["MaxItems"] = max_items

    paginator = self.client.get_paginator("list_objects_v2")
    pages = paginator.paginate(
        Bucket=self.bucket,
        Prefix=full_key,
        Delimiter="/",
        PaginationConfig=pagination_config,
    )
    files_and_dirs = []
    for page in pages:
        for obj in page.get("CommonPrefixes", []):
```

(continues on next page)

(continued from previous page)

```
            prefix = obj.get("Prefix", "")
            directory = prefix[len(full_key) :].rstrip("/")
            files_and_dirs.append(directory)

        for obj in page.get("Contents", []):
            key = obj.get("Key", "")
            file = key[len(full_key) :]
            files_and_dirs.append(file)
    return files_and_dirs
```

_normpath(path)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def _normpath(self, path: str) -> str:
    unix_path = path.replace("\", "/")
    parts = unix_path.split("/")
    stack = []

    for part in parts:
        if part == "" or part == ".":
            continue
        if part == "..":
            if stack:
                stack.pop()
        else:
            stack.append(part)

    return "/".join(stack)
```

_to_dir_full_key(path)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def _to_dir_full_key(self, path: str) -> str:
    full_key = self._to_full_key(path)
    # S3 "directories" always end with a slash, except for the root.
    if full_key != "" and not full_key.endswith("/"):
        full_key += "/"
    return full_key
```

_to_full_key(path)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def _to_full_key(self, path: str) -> str:
    path = path.lstrip("/")
    full_key = f"{self.workdir}/{path}"
    # Need to keep the trailing slash if it was there,
    # because it means the path is a directory.
    ended_with_slash = path.endswith("/")
```

(continues on next page)

(continued from previous page)

```
    full_key = self._normpath(full_key)

    if ended_with_slash:
        full_key += "/"
    return full_key.lstrip("/")
```

client()

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
@lazy_property()
def client(self) -> S3Client:
    return self.session.client("s3")
```

try_list_files(path)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def try_list_files(self, path: str) -> list[str]:
    full_key = self._to_dir_full_key(path)
    files_and_dirs = self._list_files_and_dirs(full_key)
    if len(files_and_dirs) == 0:
        if len(self._list_files_and_dirs(full_key.rstrip("/"), max_items=1)) > 0:
            raise NotADirectoryError
        raise FileNotFoundError
    return files_and_dirs
```

try_load_file(path)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def try_load_file(self, path: str) -> bytes:
    botocore = import_optional_dependency("botocore")
    full_key = self._to_full_key(path)

    if self._is_a_directory(full_key):
        raise IsADirectoryError

    try:
        response = self.client.get_object(Bucket=self.bucket, Key=full_key)
        return response["Body"].read()
    except botocore.exceptions.ClientError as e:
        if e.response["Error"]["Code"] in {"NoSuchKey", "404"}:
            raise FileNotFoundError from e
        raise e
```

try_save_file(path, value)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def try_save_file(self, path: str, value: bytes) -> str:
    full_key = self._to_full_key(path)
    if self._is_a_directory(full_key):
        raise IsADirectoryError
    self.client.put_object(Bucket=self.bucket, Key=full_key, Body=value)
    return f"s3://{self.bucket}/{full_key}"
```

amazon_s3_file_manager_driver

Bases: *BaseFileManagerDriver* “)

Attributes

Name	Type	Description
<a href="amazon_s3_file_manager_driver.mdx#griptape.drivers.file_manager.amazon_s3_file_manager_driver.AmazonS3FileManagerDriver.session" title="session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True) class-attribute instance-attribute (griptape.drivers.file_manager.amazon_s3_file_manager_driver.AmazonS3FileManagerDriver.session)"><code class="language-python">	*Session*	The boto3 session to use for S3 operations.

continues on next page

Table 154 – continued from previous page

Name	Type	Description
<a href="amazon_s3_file_manager_driver.mdx#griptape.drivers.file_manager.amazon_s3_file_manager_driver.AmazonS3FileManagerDriver.bucket" title="bucket = field(kw_only=True) class-attribute instance-attribute (griptape.drivers.file_manager.amazon_s3_file_manager_driver.AmazonS3FileManagerDriver.bucket)"><code class="language-python">	*str*	The name of the S3 bucket.
<a href="amazon_s3_file_manager_driver.mdx#griptape.drivers.file_manager.amazon_s3_file_manager_driver.AmazonS3FileManagerDriver.workdir" title="workdir property writable (griptape.drivers.file_manager.amazon_s3_file_manager_driver.AmazonS3FileManagerDriver.workdir)"><code class="language-python">	*str*	The absolute working directory (must start with "/"). List, load, and save operations will be performed relative to this directory.

continues on next page

Table 154 – continued from previous page

Name	Type	Description
<a href="amazon_s3_file_manager_driver.mdx#griptape.drivers.file_manager.amazon_s3_file_manager_driver.AmazonS3FileManagerDriver.client" title="client() (griptape.drivers.file_manager.amazon_s3_file_manager_driver.AmazonS3FileManagerDriver.client)"><code class="language-python">	*S3Client*	The S3 client to use for S3 operations.

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
@define
class AmazonS3FileManagerDriver(BaseFileManagerDriver):
    """AmazonS3FileManagerDriver can be used to list, load, and save files in an␣
↪Amazon S3 bucket.

    Attributes:
        session: The boto3 session to use for S3 operations.
        bucket: The name of the S3 bucket.
        workdir: The absolute working directory (must start with "/"). List, load,␣
↪and save
            operations will be performed relative to this directory.
        client: The S3 client to use for S3 operations.
    """

    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    bucket: str = field(kw_only=True)
    _workdir: str = field(default="/", kw_only=True, alias="workdir")
    _client: Optional[S3Client] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @property
    def workdir(self) -> str:
        if self._workdir.startswith("/"):
            return self._workdir
        return f"/{self._workdir}"

    @workdir.setter
    def workdir(self, value: str) -> None:
        self._workdir = value
```

(continues on next page)

(continued from previous page)

```
    @lazy_property()
    def client(self) -> S3Client:
        return self.session.client("s3")

    def try_list_files(self, path: str) -> list[str]:
        full_key = self._to_dir_full_key(path)
        files_and_dirs = self._list_files_and_dirs(full_key)
        if len(files_and_dirs) == 0:
            if len(self._list_files_and_dirs(full_key.rstrip("/"), max_items=1)) > 0:
                raise NotADirectoryError
            raise FileNotFoundError
        return files_and_dirs

    def try_load_file(self, path: str) -> bytes:
        botocore = import_optional_dependency("botocore")
        full_key = self._to_full_key(path)

        if self._is_a_directory(full_key):
            raise IsADirectoryError

        try:
            response = self.client.get_object(Bucket=self.bucket, Key=full_key)
            return response["Body"].read()
        except botocore.exceptions.ClientError as e:
            if e.response["Error"]["Code"] in {"NoSuchKey", "404"}:
                raise FileNotFoundError from e
            raise e

    def try_save_file(self, path: str, value: bytes) -> str:
        full_key = self._to_full_key(path)
        if self._is_a_directory(full_key):
            raise IsADirectoryError
        self.client.put_object(Bucket=self.bucket, Key=full_key, Body=value)
        return f"s3://{self.bucket}/{full_key}"

    def _to_full_key(self, path: str) -> str:
        path = path.lstrip("/")
        full_key = f"{self.workdir}/{path}"
        # Need to keep the trailing slash if it was there,
        # because it means the path is a directory.
        ended_with_slash = path.endswith("/")

        full_key = self._normpath(full_key)

        if ended_with_slash:
            full_key += "/"
        return full_key.lstrip("/")

    def _to_dir_full_key(self, path: str) -> str:
        full_key = self._to_full_key(path)
        # S3 "directories" always end with a slash, except for the root.
```

(continues on next page)

(continued from previous page)

```
        if full_key != "" and not full_key.endswith("/"):
            full_key += "/"
        return full_key

    def _list_files_and_dirs(self, full_key: str, **kwargs) -> list[str]:
        max_items = kwargs.get("max_items")
        pagination_config: PaginatorConfigTypeDef = {}
        if max_items is not None:
            pagination_config["MaxItems"] = max_items

        paginator = self.client.get_paginator("list_objects_v2")
        pages = paginator.paginate(
            Bucket=self.bucket,
            Prefix=full_key,
            Delimiter="/",
            PaginationConfig=pagination_config,
        )
        files_and_dirs = []
        for page in pages:
            for obj in page.get("CommonPrefixes", []):
                prefix = obj.get("Prefix", "")
                directory = prefix[len(full_key) :].rstrip("/")
                files_and_dirs.append(directory)

            for obj in page.get("Contents", []):
                key = obj.get("Key", "")
                file = key[len(full_key) :]
                files_and_dirs.append(file)
        return files_and_dirs

    def _is_a_directory(self, full_key: str) -> bool:
        botocore = import_optional_dependency("botocore")
        if full_key == "" or full_key.endswith("/"):
            return True

        try:
            self.client.head_object(Bucket=self.bucket, Key=full_key)
        except botocore.exceptions.ClientError as e:
            if e.response["Error"]["Code"] in {"NoSuchKey", "404"}:
                return len(self._list_files_and_dirs(full_key, max_items=1)) > 0
            raise e

        return False

    def _normpath(self, path: str) -> str:
        unix_path = path.replace("\", "/")
        parts = unix_path.split("/")
        stack = []

        for part in parts:
            if part == "" or part == ".":
                continue
```

(continues on next page)

(continued from previous page)

```
            if part == "..":
                if stack:
                    stack.pop()
            else:
                stack.append(part)

        return "/".join(stack)
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `_workdir = field(default='/', kw_only=True, alias='workdir')` class-attribute instance-attribute
- `bucket = field(kw_only=True)` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `workdir` property writable

_is_a_directory(full_key)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def _is_a_directory(self, full_key: str) -> bool:
    botocore = import_optional_dependency("botocore")
    if full_key == "" or full_key.endswith("/"):
        return True

    try:
        self.client.head_object(Bucket=self.bucket, Key=full_key)
    except botocore.exceptions.ClientError as e:
        if e.response["Error"]["Code"] in {"NoSuchKey", "404"}:
            return len(self._list_files_and_dirs(full_key, max_items=1)) > 0
        raise e

    return False
```

_list_files_and_dirs(full_key, **kwargs)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def _list_files_and_dirs(self, full_key: str, **kwargs) -> list[str]:
    max_items = kwargs.get("max_items")
    pagination_config: PaginatorConfigTypeDef = {}
    if max_items is not None:
        pagination_config["MaxItems"] = max_items

    paginator = self.client.get_paginator("list_objects_v2")
    pages = paginator.paginate(
        Bucket=self.bucket,
        Prefix=full_key,
        Delimiter="/",
```

(continues on next page)

(continued from previous page)

```
        PaginationConfig=pagination_config,
    )
    files_and_dirs = []
    for page in pages:
        for obj in page.get("CommonPrefixes", []):
            prefix = obj.get("Prefix", "")
            directory = prefix[len(full_key) :].rstrip("/")
            files_and_dirs.append(directory)

        for obj in page.get("Contents", []):
            key = obj.get("Key", "")
            file = key[len(full_key) :]
            files_and_dirs.append(file)
    return files_and_dirs
```

_normpath(path)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def _normpath(self, path: str) -> str:
    unix_path = path.replace("\", "/")
    parts = unix_path.split("/")
    stack = []

    for part in parts:
        if part == "" or part == ".":
            continue
        if part == "..":
            if stack:
                stack.pop()
        else:
            stack.append(part)

    return "/".join(stack)
```

_to_dir_full_key(path)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def _to_dir_full_key(self, path: str) -> str:
    full_key = self._to_full_key(path)
    # S3 "directories" always end with a slash, except for the root.
    if full_key != "" and not full_key.endswith("/"):
        full_key += "/"
    return full_key
```

_to_full_key(path)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def _to_full_key(self, path: str) -> str:
    path = path.lstrip("/")
    full_key = f"{self.workdir}/{path}"
    # Need to keep the trailing slash if it was there,
    # because it means the path is a directory.
    ended_with_slash = path.endswith("/")

    full_key = self._normpath(full_key)

    if ended_with_slash:
        full_key += "/"
    return full_key.lstrip("/")
```

client()

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
@lazy_property()
def client(self) -> S3Client:
    return self.session.client("s3")
```

try_list_files(path)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def try_list_files(self, path: str) -> list[str]:
    full_key = self._to_dir_full_key(path)
    files_and_dirs = self._list_files_and_dirs(full_key)
    if len(files_and_dirs) == 0:
        if len(self._list_files_and_dirs(full_key.rstrip("/"), max_items=1)) > 0:
            raise NotADirectoryError
        raise FileNotFoundError
    return files_and_dirs
```

try_load_file(path)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def try_load_file(self, path: str) -> bytes:
    botocore = import_optional_dependency("botocore")
    full_key = self._to_full_key(path)

    if self._is_a_directory(full_key):
        raise IsADirectoryError

    try:
        response = self.client.get_object(Bucket=self.bucket, Key=full_key)
        return response["Body"].read()
    except botocore.exceptions.ClientError as e:
        if e.response["Error"]["Code"] in {"NoSuchKey", "404"}:
            raise FileNotFoundError from e
        raise e
```

try_save_file(path, value)

Source Code in griptape/drivers/file_manager/amazon_s3_file_manager_driver.py

```
def try_save_file(self, path: str, value: bytes) -> str:
    full_key = self._to_full_key(path)
    if self._is_a_directory(full_key):
        raise IsADirectoryError
    self.client.put_object(Bucket=self.bucket, Key=full_key, Body=value)
    return f"s3://{self.bucket}/{full_key}"
```

base_file_manager_driver

Bases:

ABC

Attributes

Name	Type	Description
default_loader	``	The default loader to use for loading file contents into artifacts.
loaders	``	Dictionary of file extension specific loaders to use for loading file contents into artifacts.

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
@define
class BaseFileManagerDriver(ABC):
    """BaseFileManagerDriver can be used to list, load, and save files.

    Attributes:
        default_loader: The default loader to use for loading file contents into␣
↪artifacts.
        loaders: Dictionary of file extension specific loaders to use for loading␣
↪file contents into artifacts.
    """

    _workdir: str = field(kw_only=True, alias="workdir")
    encoding: Optional[str] = field(default=None, kw_only=True)

    @property
    @abstractmethod
    def workdir(self) -> str: ...

    @workdir.setter
    @abstractmethod
    def workdir(self, value: str) -> None: ...
```

(continues on next page)

(continued from previous page)

```
    def list_files(self, path: str) -> TextArtifact:
        entries = self.try_list_files(path)
        return TextArtifact("\n".join(list(entries)))

    @abstractmethod
    def try_list_files(self, path: str) -> list[str]: ...

    def load_file(self, path: str) -> BlobArtifact | TextArtifact:
        if self.encoding is None:
            return BlobArtifact(self.try_load_file(path))
        return TextArtifact(self.try_load_file(path).decode(encoding=self.encoding),␣
↪encoding=self.encoding)

    @abstractmethod
    def try_load_file(self, path: str) -> bytes: ...

    def save_file(self, path: str, value: bytes | str) -> InfoArtifact:
        if isinstance(value, str):
            value = value.encode() if self.encoding is None else value.
↪encode(encoding=self.encoding)
        elif isinstance(value, (bytearray, memoryview)):
            raise ValueError(f"Unsupported type: {type(value)}")

        location = self.try_save_file(path, value)

        return InfoArtifact(f"Successfully saved file at: {location}")

    @abstractmethod
    def try_save_file(self, path: str, value: bytes) -> str: ...

    def load_artifact(self, path: str) -> BaseArtifact:
        response = self.try_load_file(path)
        return BaseArtifact.from_json(
            response.decode() if self.encoding is None else response.
↪decode(encoding=self.encoding)
        )

    def save_artifact(self, path: str, artifact: BaseArtifact) -> InfoArtifact:
        artifact_json = artifact.to_json()
        value = artifact_json.encode() if self.encoding is None else artifact_json.
↪encode(encoding=self.encoding)

        location = self.try_save_file(path, value)

        return InfoArtifact(f"Successfully saved artifact at: {location}")
```

- `_workdir = field(kw_only=True, alias='workdir')` class-attribute instance-attribute
- `encoding = field(default=None, kw_only=True)` class-attribute instance-attribute
- `workdir` abstractmethod property writable

list_files(path)

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
def list_files(self, path: str) -> TextArtifact:
    entries = self.try_list_files(path)
    return TextArtifact("\n".join(list(entries)))
```

load_artifact(path)

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
def load_artifact(self, path: str) -> BaseArtifact:
    response = self.try_load_file(path)
    return BaseArtifact.from_json(
        response.decode() if self.encoding is None else response.decode(encoding=self.
↪encoding)
    )
```

load_file(path)

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
def load_file(self, path: str) -> BlobArtifact | TextArtifact:
    if self.encoding is None:
        return BlobArtifact(self.try_load_file(path))
    return TextArtifact(self.try_load_file(path).decode(encoding=self.encoding),␣
↪encoding=self.encoding)
```

save_artifact(path, artifact)

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
def save_artifact(self, path: str, artifact: BaseArtifact) -> InfoArtifact:
    artifact_json = artifact.to_json()
    value = artifact_json.encode() if self.encoding is None else artifact_json.
↪encode(encoding=self.encoding)

    location = self.try_save_file(path, value)

    return InfoArtifact(f"Successfully saved artifact at: {location}")
```

save_file(path, value)

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
def save_file(self, path: str, value: bytes | str) -> InfoArtifact:
    if isinstance(value, str):
        value = value.encode() if self.encoding is None else value.
↪encode(encoding=self.encoding)
    elif isinstance(value, (bytearray, memoryview)):
        raise ValueError(f"Unsupported type: {type(value)}")
```

(continues on next page)

(continued from previous page)

```
    location = self.try_save_file(path, value)

    return InfoArtifact(f"Successfully saved file at: {location}")
```

try_list_files(path)abstractmethod

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
@abstractmethod
def try_list_files(self, path: str) -> list[str]: ...
```

try_load_file(path)abstractmethod

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
@abstractmethod
def try_load_file(self, path: str) -> bytes: ...
```

try_save_file(path, value)abstractmethod

Source Code in griptape/drivers/file_manager/base_file_manager_driver.py

```
@abstractmethod
def try_save_file(self, path: str, value: bytes) -> str: ...
```

griptape_cloud

- `__all__ = ['GriptapeCloudFileManagerDriver']` module-attribute

Bases: *BaseFileManagerDriver* “)

Attributes

Name	Type	Description
<a href="grip-tape_cloud.mdx#grip-tape.drivers.file_manager.grip-tape_cloud.GripTapeCloudFileManagerDriver.bucket_id" title="bucket_id = field(default=Factory(lambda: os.getenv('GT_CLOUD_ kw_only=True) class-attribute instance-attribute (grip-tape.drivers.file_manager.grip-tape_cloud.GripTapeCloudFileManagerDriver.bucket_id)"><code class="language-python">	*Optional[str]*	The ID of the Bucket to list, load, and save Assets in. If not provided, the driver will attempt to retrieve the ID from the environment variable *GT_CLOUD_BUCKET_ID*.
<a href="grip-tape_cloud.mdx#grip-tape.drivers.file_manager.grip-tape_cloud.GripTapeCloudFileManagerDriver.workdir" title="workdir property writable (grip-tape.drivers.file_manager.grip-tape_cloud.GripTapeCloudFileManagerDriver.workdir)"><code class="language-python">	*str*	The working directory. List, load, and save operations will be performed relative to this directory.

continues on next page

Table 156 – continued from previous page

Name	Type	Description
<a href="griptape_cloud.mdx#griptape.drivers.file_manager.griptape_cloud.GriptapeCloudFileManagerDriver.base_url" title="base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_ 'https://cloud.griptape.ai'))) class-attribute instance-attribute (griptape.drivers.file_manager.griptape_cloud.GriptapeCloudFileManagerDriver.base_url)"><code class="language-python">	*str*	The base URL of the Gen AI Builder API. Defaults to the value of the environment variable *GT_CLOUD_BASE_URL* or *https://cloud.griptape.ai*.
<a href="griptape_cloud.mdx#griptape.drivers.file_manager.griptape_cloud.GriptapeCloudFileManagerDriver.api_key" title="api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_K class-attribute instance-attribute (griptape.drivers.file_manager.griptape_cloud.GriptapeCloudFileManagerDriver.api_key)"><code class="language-python">	*str*	The API key to use for authenticating with the Gen AI Builder API. If not provided, the driver will attempt to retrieve the API key from the environment variable *GT_CLOUD_API_KEY*.

Raises

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
@define
class GriptapeCloudFileManagerDriver(BaseFileManagerDriver):
    """GriptapeCloudFileManagerDriver can be used to list, load, and save files as␣
↪Assets in Gen AI Builder Buckets.

    Attributes:
```

(continues on next page)

(continued from previous page)

```
        bucket_id: The ID of the Bucket to list, load, and save Assets in. If not␣
↪provided, the driver will attempt to
            retrieve the ID from the environment variable `GT_CLOUD_BUCKET_ID`.
        workdir: The working directory. List, load, and save operations will be␣
↪performed relative to this directory.
        base_url: The base URL of the Gen AI Builder API. Defaults to the value of␣
↪the environment variable
            `GT_CLOUD_BASE_URL` or `https://cloud.griptape.ai`.
        api_key: The API key to use for authenticating with the Gen AI Builder API.␣
↪If not provided, the driver will
            attempt to retrieve the API key from the environment variable `GT_CLOUD_
↪API_KEY`.

    Raises:
        ValueError: If `api_key` is not provided, if `workdir` does not start with "/"
↪", or invalid `bucket_id` and/or `bucket_name` value(s) are provided.
    """

    bucket_id: Optional[str] = field(default=Factory(lambda: os.getenv("GT_CLOUD_
↪BUCKET_ID")), kw_only=True)
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]))
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        init=False,
    )
    _workdir: str = field(default="/", kw_only=True, alias="workdir")

    @property
    def workdir(self) -> str:
        if self._workdir.startswith("/"):
            return self._workdir
        return f"/{self._workdir}"

    @workdir.setter
    def workdir(self, value: str) -> None:
        self._workdir = value

    @bucket_id.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_bucket_id(self, _: Attribute, value: Optional[str]) -> str:
        if value is None:
            raise ValueError(f"{self.__class__.__name__} requires an Bucket ID")
        return value

    def __attrs_post_init__(self) -> None:
        try:
            self._call_api(method="get", path=f"/buckets/{self.bucket_id}").json()
```

(continues on next page)

(continued from previous page)

```
        except requests.exceptions.HTTPError as e:
            if e.response.status_code == 404:
                raise ValueError(f"No Bucket found with ID: {self.bucket_id}") from e
            raise ValueError(f"Unexpected error when retrieving Bucket with ID: {self.
↪bucket_id}") from e

    def try_list_files(self, path: str, postfix: str = "") -> list[str]:
        full_key = self._to_full_key(path)

        data = {"prefix": full_key}
        if postfix:
            data["postfix"] = postfix
        list_assets_response = self._call_api(
            method="get", path=f"/buckets/{self.bucket_id}/assets", params=data,␣
↪raise_for_status=False
        ).json()

        return [asset["name"] for asset in list_assets_response.get("assets", [])]

    def try_load_file(self, path: str) -> bytes:
        full_key = self._to_full_key(path)

        if self._is_a_directory(full_key):
            raise IsADirectoryError

        try:
            sas_url, headers = self._get_asset_url(full_key)
            response = requests.get(sas_url, headers=headers)
            response.raise_for_status()
            return response.content
        except requests.exceptions.HTTPError as e:
            if e.response.status_code == 404:
                raise FileNotFoundError from e
            raise e

    def try_save_file(self, path: str, value: bytes) -> str:
        full_key = self._to_full_key(path)

        if self._is_a_directory(full_key):
            raise IsADirectoryError

        self._call_api(
            method="put",
            path=f"/buckets/{self.bucket_id}/assets",
            json={"name": full_key},
            raise_for_status=True,
        )

        sas_url, headers = self._get_asset_url(full_key)
        response = requests.put(sas_url, data=value, headers=headers)
        response.raise_for_status()
```

(continues on next page)

(continued from previous page)

```
        return f"buckets/{self.bucket_id}/assets/{full_key}"

    def _get_asset_url(self, full_key: str) -> tuple[str, dict]:
        url_response = self._call_api(
            method="post", path=f"/buckets/{self.bucket_id}/asset-urls/{full_key}",␣
↪raise_for_status=True
        ).json()
        return url_response["url"], url_response.get("headers", {})

    def _get_url(self, path: str) -> str:
        path = path.lstrip("/")
        return griptape_cloud_url(self.base_url, f"api/{path}")

    def _call_api(
        self,
        method: str,
        path: str,
        json: Optional[dict] = None,
        params: Optional[dict] = None,
        *,
        raise_for_status: bool = True,
    ) -> requests.Response:
        res = requests.request(method, self._get_url(path), json=json, params=params,␣
↪headers=self.headers)
        if raise_for_status:
            res.raise_for_status()
        return res

    def _is_a_directory(self, path: str) -> bool:
        return path == "" or path.endswith("/")

    def _to_full_key(self, path: str) -> str:
        path = path.lstrip("/")
        full_key = f"{self.workdir}/{path}"
        return full_key.lstrip("/")
```

- `_workdir = field(default='/', kw_only=True, alias='workdir')` class-attribute instance-attribute
- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']))` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')))` class-attribute instance-attribute
- `bucket_id = field(default=Factory(lambda: os.getenv('GT_CLOUD_BUCKET_ID')), kw_only=True)` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), init=False)` class-attribute instance-attribute
- `workdir` property writable

attrs_post_init()

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def __attrs_post_init__(self) -> None:
    try:
        self._call_api(method="get", path=f"/buckets/{self.bucket_id}").json()
    except requests.exceptions.HTTPError as e:
        if e.response.status_code == 404:
            raise ValueError(f"No Bucket found with ID: {self.bucket_id}") from e
        raise ValueError(f"Unexpected error when retrieving Bucket with ID: {self.
↪bucket_id}") from e
```

_call_api(method, path, json=None, params=None, *, raise_for_status=True)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def _call_api(
    self,
    method: str,
    path: str,
    json: Optional[dict] = None,
    params: Optional[dict] = None,
    *,
    raise_for_status: bool = True,
) -> requests.Response:
    res = requests.request(method, self._get_url(path), json=json, params=params,␣
↪headers=self.headers)
    if raise_for_status:
        res.raise_for_status()
    return res
```

_get_asset_url(full_key)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def _get_asset_url(self, full_key: str) -> tuple[str, dict]:
    url_response = self._call_api(
        method="post", path=f"/buckets/{self.bucket_id}/asset-urls/{full_key}", raise_
↪for_status=True
    ).json()
    return url_response["url"], url_response.get("headers", {})
```

_get_url(path)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def _get_url(self, path: str) -> str:
    path = path.lstrip("/")
    return griptape_cloud_url(self.base_url, f"api/{path}")
```

_is_a_directory(path)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def _is_a_directory(self, path: str) -> bool:
    return path == "" or path.endswith("/")
```

_to_full_key(path)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def _to_full_key(self, path: str) -> str:
    path = path.lstrip("/")
    full_key = f"{self.workdir}/{path}"
    return full_key.lstrip("/")
```

try_list_files(path, postfix='')

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def try_list_files(self, path: str, postfix: str = "") -> list[str]:
    full_key = self._to_full_key(path)

    data = {"prefix": full_key}
    if postfix:
        data["postfix"] = postfix
    list_assets_response = self._call_api(
        method="get", path=f"/buckets/{self.bucket_id}/assets", params=data, raise_
↪for_status=False
    ).json()

    return [asset["name"] for asset in list_assets_response.get("assets", [])]
```

try_load_file(path)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def try_load_file(self, path: str) -> bytes:
    full_key = self._to_full_key(path)

    if self._is_a_directory(full_key):
        raise IsADirectoryError

    try:
        sas_url, headers = self._get_asset_url(full_key)
        response = requests.get(sas_url, headers=headers)
        response.raise_for_status()
        return response.content
    except requests.exceptions.HTTPError as e:
        if e.response.status_code == 404:
            raise FileNotFoundError from e
        raise e
```

try_save_file(path, value)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def try_save_file(self, path: str, value: bytes) -> str:
    full_key = self._to_full_key(path)

    if self._is_a_directory(full_key):
        raise IsADirectoryError

    self._call_api(
        method="put",
        path=f"/buckets/{self.bucket_id}/assets",
        json={"name": full_key},
        raise_for_status=True,
    )

    sas_url, headers = self._get_asset_url(full_key)
    response = requests.put(sas_url, data=value, headers=headers)
    response.raise_for_status()

    return f"buckets/{self.bucket_id}/assets/{full_key}"
```

validate*bucket_id(*, value)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
@bucket_id.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_bucket_id(self, _: Attribute, value: Optional[str]) -> str:
    if value is None:
        raise ValueError(f"{self.__class__.__name__} requires an Bucket ID")
    return value
```

griptape_cloud_file_manager_driver

- `logger = logging.getLogger(__name__)` module-attribute

Bases: *BaseFileManagerDriver* “)

Attributes

Name	Type	Description
<a href="griptape_cloud_file_manager_driver.mdx#griptape.drivers.file_manager.griptape_cloud_file_manager_driver.GriptapeCloudFileManagerDriver.bucket_id" title="bucket_id = field(default=Factory(lambda: os.getenv('GT_CLOUD_ kw_only=True) class-attribute instance-attribute (griptape.drivers.file_manager.griptape_cloud_file_manager_driver.GriptapeCloudFileManagerDriver.bucket_id)"><cod class="language-python">	*Optional[str]*	The ID of the Bucket to list, load, and save Assets in. If not provided, the driver will attempt to retrieve the ID from the environment variable *GT_CLOUD_BUCKET_ID*.
<a href="griptape_cloud_file_manager_driver.mdx#griptape.drivers.file_manager.griptape_cloud_file_manager_driver.GriptapeCloudFileManagerDriver.workdir" title="workdir property writable (griptape.drivers.file_manager.griptape_cloud_file_manager_driver.GriptapeCloudFileManagerDriver.workdir)"><code class="language-python">	*str*	The working directory. List, load, and save operations will be performed relative to this directory.

continues on next page

Table 157 – continued from previous page

Name	Type	Description
<a href="griptape_cloud_file_manager_driver.mdx#griptape.drivers.file_manager.griptape_cloud_file_manager_driver.GriptapeCloudFileManagerDriver.base_url" title="base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_ 'https://cloud.griptape.ai'))) class-attribute instance-attribute (griptape.drivers.file_manager.griptape_cloud_file_manager_driver.GriptapeCloudFileManagerDriver.base_url)"><code class="language-python">	*str*	The base URL of the Gen AI Builder API. Defaults to the value of the environment variable *GT_CLOUD_BASE_URL* or *https://cloud.griptape.ai*.
<a href="griptape_cloud_file_manager_driver.mdx#griptape.drivers.file_manager.griptape_cloud_file_manager_driver.GriptapeCloudFileManagerDriver.api_key" title="api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_K class-attribute instance-attribute (griptape.drivers.file_manager.griptape_cloud_file_manager_driver.GriptapeCloudFileManagerDriver.api_key)"><code class="language-python">	*str*	The API key to use for authenticating with the Gen AI Builder API. If not provided, the driver will attempt to retrieve the API key from the environment variable *GT_CLOUD_API_KEY*.

Raises

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
@define
class GriptapeCloudFileManagerDriver(BaseFileManagerDriver):
    """GriptapeCloudFileManagerDriver can be used to list, load, and save files as␣
↪Assets in Gen AI Builder Buckets.

    Attributes:
        bucket_id: The ID of the Bucket to list, load, and save Assets in. If not␣
↪provided, the driver will attempt to
            retrieve the ID from the environment variable `GT_CLOUD_BUCKET_ID`.
        workdir: The working directory. List, load, and save operations will be␣
↪performed relative to this directory.
        base_url: The base URL of the Gen AI Builder API. Defaults to the value of␣
↪the environment variable
            `GT_CLOUD_BASE_URL` or `https://cloud.griptape.ai`.
        api_key: The API key to use for authenticating with the Gen AI Builder API.␣
↪If not provided, the driver will
            attempt to retrieve the API key from the environment variable `GT_CLOUD_
↪API_KEY`.

    Raises:
        ValueError: If `api_key` is not provided, if `workdir` does not start with "/"
↪", or invalid `bucket_id` and/or `bucket_name` value(s) are provided.
    """

    bucket_id: Optional[str] = field(default=Factory(lambda: os.getenv("GT_CLOUD_
↪BUCKET_ID")), kw_only=True)
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]))
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        init=False,
    )
    _workdir: str = field(default="/", kw_only=True, alias="workdir")

    @property
    def workdir(self) -> str:
        if self._workdir.startswith("/"):
            return self._workdir
        return f"/{self._workdir}"

    @workdir.setter
    def workdir(self, value: str) -> None:
        self._workdir = value

    @bucket_id.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
```

(continues on next page)

(continued from previous page)

```
    def validate_bucket_id(self, _: Attribute, value: Optional[str]) -> str:
        if value is None:
            raise ValueError(f"{self.__class__.__name__} requires an Bucket ID")
        return value

    def __attrs_post_init__(self) -> None:
        try:
            self._call_api(method="get", path=f"/buckets/{self.bucket_id}").json()
        except requests.exceptions.HTTPError as e:
            if e.response.status_code == 404:
                raise ValueError(f"No Bucket found with ID: {self.bucket_id}") from e
            raise ValueError(f"Unexpected error when retrieving Bucket with ID: {self.
↪bucket_id}") from e

    def try_list_files(self, path: str, postfix: str = "") -> list[str]:
        full_key = self._to_full_key(path)

        data = {"prefix": full_key}
        if postfix:
            data["postfix"] = postfix
        list_assets_response = self._call_api(
            method="get", path=f"/buckets/{self.bucket_id}/assets", params=data,␣
↪raise_for_status=False
        ).json()

        return [asset["name"] for asset in list_assets_response.get("assets", [])]

    def try_load_file(self, path: str) -> bytes:
        full_key = self._to_full_key(path)

        if self._is_a_directory(full_key):
            raise IsADirectoryError

        try:
            sas_url, headers = self._get_asset_url(full_key)
            response = requests.get(sas_url, headers=headers)
            response.raise_for_status()
            return response.content
        except requests.exceptions.HTTPError as e:
            if e.response.status_code == 404:
                raise FileNotFoundError from e
            raise e

    def try_save_file(self, path: str, value: bytes) -> str:
        full_key = self._to_full_key(path)

        if self._is_a_directory(full_key):
            raise IsADirectoryError

        self._call_api(
            method="put",
            path=f"/buckets/{self.bucket_id}/assets",
```

(continues on next page)

(continued from previous page)

```
            json={"name": full_key},
            raise_for_status=True,
        )

        sas_url, headers = self._get_asset_url(full_key)
        response = requests.put(sas_url, data=value, headers=headers)
        response.raise_for_status()

        return f"buckets/{self.bucket_id}/assets/{full_key}"

    def _get_asset_url(self, full_key: str) -> tuple[str, dict]:
        url_response = self._call_api(
            method="post", path=f"/buckets/{self.bucket_id}/asset-urls/{full_key}",␣
↪raise_for_status=True
        ).json()
        return url_response["url"], url_response.get("headers", {})

    def _get_url(self, path: str) -> str:
        path = path.lstrip("/")
        return griptape_cloud_url(self.base_url, f"api/{path}")

    def _call_api(
        self,
        method: str,
        path: str,
        json: Optional[dict] = None,
        params: Optional[dict] = None,
        *,
        raise_for_status: bool = True,
    ) -> requests.Response:
        res = requests.request(method, self._get_url(path), json=json, params=params,␣
↪headers=self.headers)
        if raise_for_status:
            res.raise_for_status()
        return res

    def _is_a_directory(self, path: str) -> bool:
        return path == "" or path.endswith("/")

    def _to_full_key(self, path: str) -> str:
        path = path.lstrip("/")
        full_key = f"{self.workdir}/{path}"
        return full_key.lstrip("/")
```

- `_workdir = field(default='/', kw_only=True, alias='workdir')` class-attribute instance-attribute
- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']))` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')))` class-attribute instance-attribute
- `bucket_id = field(default=Factory(lambda: os.getenv('GT_CLOUD_BUCKET_ID')),`

`kw_only=True)` class-attribute instance-attribute

- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), init=False)` class-attribute instance-attribute
- `workdir` property writable

attrs_post_init()

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def __attrs_post_init__(self) -> None:
    try:
        self._call_api(method="get", path=f"/buckets/{self.bucket_id}").json()
    except requests.exceptions.HTTPError as e:
        if e.response.status_code == 404:
            raise ValueError(f"No Bucket found with ID: {self.bucket_id}") from e
        raise ValueError(f"Unexpected error when retrieving Bucket with ID: {self.
↪bucket_id}") from e
```

_call_api(method, path, json=None, params=None, *, raise_for_status=True)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def _call_api(
    self,
    method: str,
    path: str,
    json: Optional[dict] = None,
    params: Optional[dict] = None,
    *,
    raise_for_status: bool = True,
) -> requests.Response:
    res = requests.request(method, self._get_url(path), json=json, params=params,␣
↪headers=self.headers)
    if raise_for_status:
        res.raise_for_status()
    return res
```

_get_asset_url(full_key)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def _get_asset_url(self, full_key: str) -> tuple[str, dict]:
    url_response = self._call_api(
        method="post", path=f"/buckets/{self.bucket_id}/asset-urls/{full_key}", raise_
↪for_status=True
    ).json()
    return url_response["url"], url_response.get("headers", {})
```

_get_url(path)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def _get_url(self, path: str) -> str:
    path = path.lstrip("/")
    return griptape_cloud_url(self.base_url, f"api/{path}")
```

_is_a_directory(path)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def _is_a_directory(self, path: str) -> bool:
    return path == "" or path.endswith("/")
```

_to_full_key(path)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def _to_full_key(self, path: str) -> str:
    path = path.lstrip("/")
    full_key = f"{self.workdir}/{path}"
    return full_key.lstrip("/")
```

try_list_files(path, postfix='')

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def try_list_files(self, path: str, postfix: str = "") -> list[str]:
    full_key = self._to_full_key(path)

    data = {"prefix": full_key}
    if postfix:
        data["postfix"] = postfix
    list_assets_response = self._call_api(
        method="get", path=f"/buckets/{self.bucket_id}/assets", params=data, raise_
↪for_status=False
    ).json()

    return [asset["name"] for asset in list_assets_response.get("assets", [])]
```

try_load_file(path)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def try_load_file(self, path: str) -> bytes:
    full_key = self._to_full_key(path)

    if self._is_a_directory(full_key):
        raise IsADirectoryError

    try:
```

(continues on next page)

(continued from previous page)

```
        sas_url, headers = self._get_asset_url(full_key)
        response = requests.get(sas_url, headers=headers)
        response.raise_for_status()
        return response.content
    except requests.exceptions.HTTPError as e:
        if e.response.status_code == 404:
            raise FileNotFoundError from e
        raise e
```

try_save_file(path, value)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
def try_save_file(self, path: str, value: bytes) -> str:
    full_key = self._to_full_key(path)

    if self._is_a_directory(full_key):
        raise IsADirectoryError

    self._call_api(
        method="put",
        path=f"/buckets/{self.bucket_id}/assets",
        json={"name": full_key},
        raise_for_status=True,
    )

    sas_url, headers = self._get_asset_url(full_key)
    response = requests.put(sas_url, data=value, headers=headers)
    response.raise_for_status()

    return f"buckets/{self.bucket_id}/assets/{full_key}"
```

validate*bucket_id(*, value)

Source Code in griptape/drivers/file_manager/griptape_cloud_file_manager_driver.py

```
@bucket_id.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_bucket_id(self, _: Attribute, value: Optional[str]) -> str:
    if value is None:
        raise ValueError(f"{self.__class__.__name__} requires an Bucket ID")
    return value
```

local

- `__all__ = ['LocalFileManagerDriver']` module-attribute

Bases: *BaseFileManagerDriver* “)

Attributes

Name	Type	Description
<a href="local.mdx#griptape.drivers.file_manager.local.LocalFileManagerDriver.workdir" title="workdir property writable (griptape.drivers.file_manager.local.LocalFileManagerDriver.workdir)"><code class="language-python">	*str*	The working directory as an absolute path. List, load, and save operations will be performed relative to this directory. Defaults to the current working directory. Setting this to None will disable the working directory and all paths will be treated as absolute paths.

Source Code in griptape/drivers/file_manager/local_file_manager_driver.py

```
@define
class LocalFileManagerDriver(BaseFileManagerDriver):
    """LocalFileManagerDriver can be used to list, load, and save files on the local␣
↪file system.

    Attributes:
        workdir: The working directory as an absolute path. List, load, and save␣
↪operations will be performed relative to this directory.
                 Defaults to the current working directory.
                 Setting this to None will disable the working directory and all␣
↪paths will be treated as absolute paths.
    """

    _workdir: str = field(default=Factory(lambda: os.getcwd()), kw_only=True, alias=
↪"workdir")

    @property
    def workdir(self) -> str:
        if os.path.isabs(self._workdir):
            return self._workdir
        return os.path.join(os.getcwd(), self._workdir)

    @workdir.setter
    def workdir(self, value: str) -> None:
        self._workdir = value

    def try_list_files(self, path: str) -> list[str]:
        full_path = self._full_path(path)
        return os.listdir(full_path)

    def try_load_file(self, path: str) -> bytes:
        full_path = self._full_path(path)
        if self._is_dir(full_path):
            raise IsADirectoryError
```

(continues on next page)

(continued from previous page)

```
        return Path(full_path).read_bytes()

    def try_save_file(self, path: str, value: bytes) -> str:
        full_path = self._full_path(path)
        if self._is_dir(full_path):
            raise IsADirectoryError
        os.makedirs(os.path.dirname(full_path), exist_ok=True)
        Path(full_path).write_bytes(value)
        return full_path

    def _full_path(self, path: str) -> str:
        full_path = path if os.path.isabs(path) else os.path.join(self.workdir, path.
↪lstrip("/"))
        # Need to keep the trailing slash if it was there,
        # because it means the path is a directory.
        ended_with_sep = path.endswith("/")
        full_path = os.path.normpath(full_path)
        if ended_with_sep:
            full_path = full_path.rstrip("/") + "/"
        return full_path

    def _is_dir(self, full_path: str) -> bool:
        return full_path.endswith("/") or Path(full_path).is_dir()
```

- `_workdir = field(default=Factory(lambda: os.getcwd()), kw_only=True, alias='workdir')` class-attribute instance-attribute
- `workdir` property writable

_full_path(path)

Source Code in griptape/drivers/file_manager/local_file_manager_driver.py

```
def _full_path(self, path: str) -> str:
    full_path = path if os.path.isabs(path) else os.path.join(self.workdir, path.
↪lstrip("/"))
    # Need to keep the trailing slash if it was there,
    # because it means the path is a directory.
    ended_with_sep = path.endswith("/")
    full_path = os.path.normpath(full_path)
    if ended_with_sep:
        full_path = full_path.rstrip("/") + "/"
    return full_path
```

_is_dir(full_path)

Source Code in griptape/drivers/file_manager/local_file_manager_driver.py

```
def _is_dir(self, full_path: str) -> bool:
    return full_path.endswith("/") or Path(full_path).is_dir()
```

try_list_files(path)

Source Code in griptape/drivers/file_manager/local_file_manager_driver.py

```
def try_list_files(self, path: str) -> list[str]:
    full_path = self._full_path(path)
    return os.listdir(full_path)
```

try_load_file(path)

Source Code in griptape/drivers/file_manager/local_file_manager_driver.py

```
def try_load_file(self, path: str) -> bytes:
    full_path = self._full_path(path)
    if self._is_dir(full_path):
        raise IsADirectoryError
    return Path(full_path).read_bytes()
```

try_save_file(path, value)

Source Code in griptape/drivers/file_manager/local_file_manager_driver.py

```
def try_save_file(self, path: str, value: bytes) -> str:
    full_path = self._full_path(path)
    if self._is_dir(full_path):
        raise IsADirectoryError
    os.makedirs(os.path.dirname(full_path), exist_ok=True)
    Path(full_path).write_bytes(value)
    return full_path
```

local_file_manager_driver

Bases: *BaseFileManagerDriver* “)

Attributes

Name	Type	Description
<a href="local_file_manager_driver.mdx#griptape.drivers.file_manager.local_file_manager_driver.LocalFileManagerDriver.workdir" title="workdir property writable (griptape.drivers.file_manager.local_file_manager_driver.LocalFileManagerDriver.workdir)"><code class="language-python">	*str*	The working directory as an absolute path. List, load, and save operations will be performed relative to this directory. Defaults to the current working directory. Setting this to None will disable the working directory and all paths will be treated as absolute paths.

Source Code in griptape/drivers/file_manager/local_file_manager_driver.py

```python
@define
class LocalFileManagerDriver(BaseFileManagerDriver):
    """LocalFileManagerDriver can be used to list, load, and save files on the local file system.

    Attributes:
        workdir: The working directory as an absolute path. List, load, and save operations will be performed relative to this directory.
                 Defaults to the current working directory.
                 Setting this to None will disable the working directory and all paths will be treated as absolute paths.
    """

    _workdir: str = field(default=Factory(lambda: os.getcwd()), kw_only=True, alias="workdir")

    @property
    def workdir(self) -> str:
        if os.path.isabs(self._workdir):
            return self._workdir
        return os.path.join(os.getcwd(), self._workdir)

    @workdir.setter
    def workdir(self, value: str) -> None:
        self._workdir = value

    def try_list_files(self, path: str) -> list[str]:
        full_path = self._full_path(path)
        return os.listdir(full_path)

    def try_load_file(self, path: str) -> bytes:
        full_path = self._full_path(path)
```

(continues on next page)

(continued from previous page)

```
        if self._is_dir(full_path):
            raise IsADirectoryError
        return Path(full_path).read_bytes()

    def try_save_file(self, path: str, value: bytes) -> str:
        full_path = self._full_path(path)
        if self._is_dir(full_path):
            raise IsADirectoryError
        os.makedirs(os.path.dirname(full_path), exist_ok=True)
        Path(full_path).write_bytes(value)
        return full_path

    def _full_path(self, path: str) -> str:
        full_path = path if os.path.isabs(path) else os.path.join(self.workdir, path.
↪lstrip("/"))
        # Need to keep the trailing slash if it was there,
        # because it means the path is a directory.
        ended_with_sep = path.endswith("/")
        full_path = os.path.normpath(full_path)
        if ended_with_sep:
            full_path = full_path.rstrip("/") + "/"
        return full_path

    def _is_dir(self, full_path: str) -> bool:
        return full_path.endswith("/") or Path(full_path).is_dir()
```

- `_workdir = field(default=Factory(lambda: os.getcwd()), kw_only=True, alias='workdir')` class-attribute instance-attribute
- `workdir` property writable

_full_path(path)

Source Code in griptape/drivers/file_manager/local_file_manager_driver.py

```
def _full_path(self, path: str) -> str:
    full_path = path if os.path.isabs(path) else os.path.join(self.workdir, path.
↪lstrip("/"))
    # Need to keep the trailing slash if it was there,
    # because it means the path is a directory.
    ended_with_sep = path.endswith("/")
    full_path = os.path.normpath(full_path)
    if ended_with_sep:
        full_path = full_path.rstrip("/") + "/"
    return full_path
```

_is_dir(full_path)

Source Code in griptape/drivers/file_manager/local_file_manager_driver.py

```
def _is_dir(self, full_path: str) -> bool:
    return full_path.endswith("/") or Path(full_path).is_dir()
```

try_list_files(path)

Source Code in griptape/drivers/file_manager/local_file_manager_driver.py

```
def try_list_files(self, path: str) -> list[str]:
    full_path = self._full_path(path)
    return os.listdir(full_path)
```

try_load_file(path)

Source Code in griptape/drivers/file_manager/local_file_manager_driver.py

```
def try_load_file(self, path: str) -> bytes:
    full_path = self._full_path(path)
    if self._is_dir(full_path):
        raise IsADirectoryError
    return Path(full_path).read_bytes()
```

try_save_file(path, value)

Source Code in griptape/drivers/file_manager/local_file_manager_driver.py

```
def try_save_file(self, path: str, value: bytes) -> str:
    full_path = self._full_path(path)
    if self._is_dir(full_path):
        raise IsADirectoryError
    os.makedirs(os.path.dirname(full_path), exist_ok=True)
    Path(full_path).write_bytes(value)
    return full_path
```

image_generation

- `__all__ = ['BaseImageGenerationDriver', 'BaseMultiModelImageGenerationDriver']` module-attribute

Bases: *SerializableMixin* “) , *ExponentialBackoffMixin* ”) , ABC

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
@define
class BaseImageGenerationDriver(SerializableMixin, ExponentialBackoffMixin, ABC):
    model: str = field(kw_only=True, metadata={"serialize": True})

    def before_run(self, prompts: list[str], negative_prompts: Optional[list[str]] =␣
↪None) -> None:
        EventBus.publish_event(StartImageGenerationEvent(prompts=prompts, negative_
↪prompts=negative_prompts))

    def after_run(self) -> None:
        EventBus.publish_event(FinishImageGenerationEvent())

    def run_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact:
        for attempt in self.retrying():
```

(continues on next page)

(continued from previous page)

```
            with attempt:
                self.before_run(prompts, negative_prompts)
                result = self.try_text_to_image(prompts, negative_prompts)
                self.after_run()

                return result

        raise Exception("Failed to run text to image generation")

    def run_image_variation(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        for attempt in self.retrying():
            with attempt:
                self.before_run(prompts, negative_prompts)
                result = self.try_image_variation(prompts, image, negative_prompts)
                self.after_run()

                return result

        raise Exception("Failed to generate image variations")

    def run_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        for attempt in self.retrying():
            with attempt:
                self.before_run(prompts, negative_prompts)
                result = self.try_image_inpainting(prompts, image, mask, negative_
↪prompts)
                self.after_run()

                return result

        raise Exception("Failed to run image inpainting")

    def run_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        for attempt in self.retrying():
            with attempt:
```

(continues on next page)

(continued from previous page)

```
                self.before_run(prompts, negative_prompts)
                result = self.try_image_outpainting(prompts, image, mask, negative_
↪prompts)
                self.after_run()

                return result

        raise Exception("Failed to run image outpainting")

    @abstractmethod
    def try_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact: ...

    @abstractmethod
    def try_image_variation(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact: ...

    @abstractmethod
    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact: ...

    @abstractmethod
    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact: ...
```

- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

after_run()

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
def after_run(self) -> None:
    EventBus.publish_event(FinishImageGenerationEvent())
```

before_run(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
def before_run(self, prompts: list[str], negative_prompts: Optional[list[str]] =␣
↪None) -> None:
    EventBus.publish_event(StartImageGenerationEvent(prompts=prompts, negative_
↪prompts=negative_prompts))
```

run_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
def run_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    for attempt in self.retrying():
        with attempt:
            self.before_run(prompts, negative_prompts)
            result = self.try_image_inpainting(prompts, image, mask, negative_prompts)
            self.after_run()

            return result

    raise Exception("Failed to run image inpainting")
```

run_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
def run_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    for attempt in self.retrying():
        with attempt:
            self.before_run(prompts, negative_prompts)
            result = self.try_image_outpainting(prompts, image, mask, negative_
↪prompts)
            self.after_run()

            return result

    raise Exception("Failed to run image outpainting")
```

run_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
def run_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    for attempt in self.retrying():
        with attempt:
            self.before_run(prompts, negative_prompts)
            result = self.try_image_variation(prompts, image, negative_prompts)
            self.after_run()

            return result

    raise Exception("Failed to generate image variations")
```

run_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
def run_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    for attempt in self.retrying():
        with attempt:
            self.before_run(prompts, negative_prompts)
            result = self.try_text_to_image(prompts, negative_prompts)
            self.after_run()

            return result

    raise Exception("Failed to run text to image generation")
```

try_image_inpainting(prompts, image, mask, negative_prompts=None)abstractmethod

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
@abstractmethod
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact: ...
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)abstractmethod

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
@abstractmethod
def try_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact: ...
```

try_image_variation(prompts, image, negative_prompts=None)abstractmethod

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
@abstractmethod
def try_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact: ...
```

try_text_to_image(prompts, negative_prompts=None)abstractmethod

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
@abstractmethod
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact: ...
```

BaseMultiModelImageGenerationDriver

Bases: *BaseImageGenerationDriver* “) , ABC

Attributes

Name	Type	Description
<a href="./#griptape.drivers.image_generation.BaseMultiModelImageGenerationDriver.image_generation_model_driver" title="image_generation_model_driver = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.BaseMultiModelImageGenerationDriver.image_generation_model_driver)"><co class="language-python">	<a href="../image_generation_model#griptape.drivers.image_generation_model.BaseImageGenerationModelDriver" title="BaseImageGenerationModelDriver (griptape.drivers.image_generation_model.BaseImageGenerationModelDriver)"><code class="language-python">BaseImageGenerationModel	Image Model Driver to use.

Source Code in griptape/drivers/image_generation/base_multi_model_image_generation_driver.py

```
@define
class BaseMultiModelImageGenerationDriver(BaseImageGenerationDriver, ABC):
    """Image Generation Driver for platforms like Amazon Bedrock that host many LLM␣
↪models.

    Instances of this Image Generation Driver require a Image Generation Model Driver␣
↪which is used to structure the
    image generation request in the format required by the model and to process the␣
↪output.

    Attributes:
        image_generation_model_driver: Image Model Driver to use.
    """

    image_generation_model_driver: BaseImageGenerationModelDriver = field(kw_␣
↪only=True, metadata={"serializable": True})
```

- `image_generation_model_driver = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

amazon_bedrock

- `__all__ = ['AmazonBedrockImageGenerationDriver']` module-attribute

Bases: *BaseMultiModelImageGenerationDriver* “)

Attributes

Name	Type	Description
<a href="base_image_generation_driver.mdx#griptape.drivers.image_generation.base_image_generation_driver.BaseImageGenerationDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.amazon_bedrock.AmazonBedrockImageGenerationDriver.model)"><cod class="language-python">	*str*	Bedrock model ID.
<a href="amazon_bedrock.mdx#griptape.drivers.image_generation.amazon_bedrock.AmazonBedrockImageGenerationDriver.session" title="session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True) class-attribute instance-attribute (griptape.drivers.image_generation.amazon_bedrock.AmazonBedrockImageGenerationDriver.session)"><code class="language-python">	*Session*	boto3 session.

continues on next page

Table 161 – continued from previous page

Name	Type	Description
<a href="amazon_bedrock.mdx#griptape.drivers.image_generation.amazon_bedrock.AmazonBedrockImageGenerationDriver.client" title="client() (griptape.drivers.image_generation.amazon_bedrock.AmazonBedrockImageGenerationDriver.client)"><code class="language-python">	*BedrockRuntimeClient*	Bedrock runtime client.
<a href="amazon_bedrock.mdx#griptape.drivers.image_generation.amazon_bedrock.AmazonBedrockImageGenerationDriver.image_width" title="image_width = field(default=512, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.amazon_bedrock.AmazonBedrockImageGenerationDriver.image_width)"><code class="language-python">	*int*	Width of output images. Defaults to 512 and must be a multiple of 64.

continues on next page

Table 161 – continued from previous page

Name	Type	Description
<a href="amazon_bedrock.mdx#griptape.drivers.image_generation.amazon_bedrock.AmazonBedrockImageGenerationDriver.image_height" title="image_height = field(default=512, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.amazon_bedrock.AmazonBedrockImageGenerationDriver.image_height)"><code class="language-python">	*int*	Height of output images. Defaults to 512 and must be a multiple of 64.
<a href="amazon_bedrock.mdx#griptape.drivers.image_generation.amazon_bedrock.AmazonBedrockImageGenerationDriver.seed" title="seed = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.amazon_bedrock.AmazonBedrockImageGenerationDriver.seed)"><code class="language-python">	*Optional[int]*	Optionally provide a consistent seed to generation requests, increasing consistency in output.

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
@define
class AmazonBedrockImageGenerationDriver(BaseMultiModelImageGenerationDriver):
    """Driver for image generation models provided by Amazon Bedrock.

    Attributes:
        model: Bedrock model ID.
```

(continues on next page)

(continued from previous page)

```
        session: boto3 session.
        client: Bedrock runtime client.
        image_width: Width of output images. Defaults to 512 and must be a multiple␣
↪of 64.
        image_height: Height of output images. Defaults to 512 and must be a multiple␣
↪of 64.
        seed: Optionally provide a consistent seed to generation requests, increasing␣
↪consistency in output.
    """

    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    image_width: int = field(default=512, kw_only=True, metadata={"serializable":␣
↪True})
    image_height: int = field(default=512, kw_only=True, metadata={"serializable":␣
↪True})
    seed: Optional[int] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    _client: Optional[BedrockRuntimeClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> BedrockRuntimeClient:
        return self.session.client("bedrock-runtime")

    def try_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact:
        request = self.image_generation_model_driver.text_to_image_request_parameters(
            prompts,
            self.image_width,
            self.image_height,
            negative_prompts=negative_prompts,
            seed=self.seed,
        )

        image_bytes = self._make_request(request)

        return ImageArtifact(
            value=image_bytes,
            format="png",
            width=self.image_width,
            height=self.image_height,
            meta={"prompt": ", ".join(prompts), "model": self.model},
        )

    def try_image_variation(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
```

(continues on next page)

(continued from previous page)

```
        request = self.image_generation_model_driver.image_variation_request_
↪parameters(
            prompts,
            image=image,
            negative_prompts=negative_prompts,
            seed=self.seed,
        )

        image_bytes = self._make_request(request)

        return ImageArtifact(
            value=image_bytes,
            format="png",
            width=image.width,
            height=image.height,
            meta={"prompt": ", ".join(prompts), "model": self.model},
        )

    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        request = self.image_generation_model_driver.image_inpainting_request_
↪parameters(
            prompts,
            image=image,
            mask=mask,
            negative_prompts=negative_prompts,
            seed=self.seed,
        )

        image_bytes = self._make_request(request)

        return ImageArtifact(
            value=image_bytes,
            format="png",
            width=image.width,
            height=image.height,
            meta={"prompt": ", ".join(prompts), "model": self.model},
        )

    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        request = self.image_generation_model_driver.image_outpainting_request_
```

(continues on next page)

(continued from previous page)

```
↪parameters(
            prompts,
            image=image,
            mask=mask,
            negative_prompts=negative_prompts,
            seed=self.seed,
        )

        image_bytes = self._make_request(request)

        return ImageArtifact(
            value=image_bytes,
            format="png",
            width=image.width,
            height=image.height,
            meta={"prompt": ", ".join(prompts), "model": self.model},
        )

    def _make_request(self, request: dict) -> bytes:
        response = self.client.invoke_model(
            body=json.dumps(request),
            modelId=self.model,
            accept="application/json",
            contentType="application/json",
        )

        response_body = json.loads(response.get("body").read())

        try:
            image_bytes = self.image_generation_model_driver.get_generated_
↪image(response_body)
        except Exception as e:
            raise ValueError(f"Inpainting generation failed: {e}") from e

        return image_bytes
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `image_height = field(default=512, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `image_width = field(default=512, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `seed = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute

_make_request(request)

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
def _make_request(self, request: dict) -> bytes:
    response = self.client.invoke_model(
        body=json.dumps(request),
        modelId=self.model,
        accept="application/json",
        contentType="application/json",
    )

    response_body = json.loads(response.get("body").read())

    try:
        image_bytes = self.image_generation_model_driver.get_generated_image(response_
↪body)
    except Exception as e:
        raise ValueError(f"Inpainting generation failed: {e}") from e

    return image_bytes
```

client()

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
@lazy_property()
def client(self) -> BedrockRuntimeClient:
    return self.session.client("bedrock-runtime")
```

try_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    request = self.image_generation_model_driver.image_inpainting_request_parameters(
        prompts,
        image=image,
        mask=mask,
        negative_prompts=negative_prompts,
        seed=self.seed,
    )

    image_bytes = self._make_request(request)

    return ImageArtifact(
        value=image_bytes,
```

(continues on next page)

(continued from previous page)

```
        format="png",
        width=image.width,
        height=image.height,
        meta={"prompt": ", ".join(prompts), "model": self.model},
    )
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
def try_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    request = self.image_generation_model_driver.image_outpainting_request_parameters(
        prompts,
        image=image,
        mask=mask,
        negative_prompts=negative_prompts,
        seed=self.seed,
    )

    image_bytes = self._make_request(request)

    return ImageArtifact(
        value=image_bytes,
        format="png",
        width=image.width,
        height=image.height,
        meta={"prompt": ", ".join(prompts), "model": self.model},
    )
```

try_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
def try_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    request = self.image_generation_model_driver.image_variation_request_parameters(
        prompts,
        image=image,
        negative_prompts=negative_prompts,
        seed=self.seed,
    )
```

(continues on next page)

(continued from previous page)

```
    image_bytes = self._make_request(request)

    return ImageArtifact(
        value=image_bytes,
        format="png",
        width=image.width,
        height=image.height,
        meta={"prompt": ", ".join(prompts), "model": self.model},
    )
```

try_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    request = self.image_generation_model_driver.text_to_image_request_parameters(
        prompts,
        self.image_width,
        self.image_height,
        negative_prompts=negative_prompts,
        seed=self.seed,
    )

    image_bytes = self._make_request(request)

    return ImageArtifact(
        value=image_bytes,
        format="png",
        width=self.image_width,
        height=self.image_height,
        meta={"prompt": ", ".join(prompts), "model": self.model},
    )
```

amazon_bedrock_image_generation_driver

Bases: *BaseMultiModelImageGenerationDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.image_generation.BaseImageGenerationDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.amazon_bedrock_image_generation_driver.AmazonBedrockImageGenerationDriver.model)"><cod class="language-python">	*str*	Bedrock model ID.
<a href="amazon_bedrock_image_generation_driver.mdx#griptape.drivers.image_generation.amazon_bedrock_image_generation_driver.AmazonBedrockImageGenerationDriver.session" title="session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True) class-attribute instance-attribute (griptape.drivers.image_generation.amazon_bedrock_image_generation_driver.AmazonBedrockImageGenerationDriver.session)"><code class="language-python">	*Session*	boto3 session.

continues on next page

Table 162 – continued from previous page

Name	Type	Description
<a href="amazon_bedrock_image_generation_driver.mdx#griptape.drivers.image_generation.amazon_bedrock_image_generation_driver.AmazonBedrockImageGenerationDriver.client" title="client() (griptape.drivers.image_generation.amazon_bedrock_image_generation_driver.AmazonBedrockImageGenerationDriver.client)"><code class="language-python">	*BedrockRuntimeClient*	Bedrock runtime client.
<a href="amazon_bedrock_image_generation_driver.mdx#griptape.drivers.image_generation.amazon_bedrock_image_generation_driver.AmazonBedrockImageGenerationDriver.image_width" title="image_width = field(default=512, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.amazon_bedrock_image_generation_driver.AmazonBedrockImageGenerationDriver.image_width)"><code class="language-python">	*int*	Width of output images. Defaults to 512 and must be a multiple of 64.

continues on next page

Table 162 – continued from previous page

Name	Type	Description
<a href="amazon_bedrock_image_generation_driver.mdx#griptape.drivers.image_generation.amazon_bedrock_image_generation_driver.AmazonBedrockImageGenerationDriver.image_height" title="image_height = field(default=512, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.amazon_bedrock_image_generation_driver.AmazonBedrockImageGenerationDriver.image_height)"><code class="language-python">	*int*	Height of output images. Defaults to 512 and must be a multiple of 64.

continues on next page

Table 162 – continued from previous page

Name	Type	Description
<a href="amazon_bedrock_image_generation_driver.mdx#griptape.drivers.image_generation.amazon_bedrock_image_generation_driver.AmazonBedrockImageGenerationDriver.seed" title="seed = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.amazon_bedrock_image_generation_driver.AmazonBedrockImageGenerationDriver.seed)"><code class="language-python">	*Optional[int]*	Optionally provide a consistent seed to generation requests, increasing consistency in output.

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
@define
class AmazonBedrockImageGenerationDriver(BaseMultiModelImageGenerationDriver):
    """Driver for image generation models provided by Amazon Bedrock.

    Attributes:
        model: Bedrock model ID.
        session: boto3 session.
        client: Bedrock runtime client.
        image_width: Width of output images. Defaults to 512 and must be a multiple␣
↪of 64.
        image_height: Height of output images. Defaults to 512 and must be a multiple␣
↪of 64.
        seed: Optionally provide a consistent seed to generation requests, increasing␣
↪consistency in output.
    """

    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    image_width: int = field(default=512, kw_only=True, metadata={"serializable":␣
↪True})
    image_height: int = field(default=512, kw_only=True, metadata={"serializable":␣
```

(continues on next page)

(continued from previous page)

```
↪True})
    seed: Optional[int] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    _client: Optional[BedrockRuntimeClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> BedrockRuntimeClient:
        return self.session.client("bedrock-runtime")

    def try_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact:
        request = self.image_generation_model_driver.text_to_image_request_parameters(
            prompts,
            self.image_width,
            self.image_height,
            negative_prompts=negative_prompts,
            seed=self.seed,
        )

        image_bytes = self._make_request(request)

        return ImageArtifact(
            value=image_bytes,
            format="png",
            width=self.image_width,
            height=self.image_height,
            meta={"prompt": ", ".join(prompts), "model": self.model},
        )

    def try_image_variation(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        request = self.image_generation_model_driver.image_variation_request_
↪parameters(
            prompts,
            image=image,
            negative_prompts=negative_prompts,
            seed=self.seed,
        )

        image_bytes = self._make_request(request)

        return ImageArtifact(
            value=image_bytes,
            format="png",
            width=image.width,
            height=image.height,
```

(continues on next page)

(continued from previous page)

```
            meta={"prompt": ", ".join(prompts), "model": self.model},
        )

    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        request = self.image_generation_model_driver.image_inpainting_request_
↪parameters(
            prompts,
            image=image,
            mask=mask,
            negative_prompts=negative_prompts,
            seed=self.seed,
        )

        image_bytes = self._make_request(request)

        return ImageArtifact(
            value=image_bytes,
            format="png",
            width=image.width,
            height=image.height,
            meta={"prompt": ", ".join(prompts), "model": self.model},
        )

    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        request = self.image_generation_model_driver.image_outpainting_request_
↪parameters(
            prompts,
            image=image,
            mask=mask,
            negative_prompts=negative_prompts,
            seed=self.seed,
        )

        image_bytes = self._make_request(request)

        return ImageArtifact(
            value=image_bytes,
            format="png",
            width=image.width,
            height=image.height,
```

(continues on next page)

(continued from previous page)

```
            meta={"prompt": ", ".join(prompts), "model": self.model},
        )

    def _make_request(self, request: dict) -> bytes:
        response = self.client.invoke_model(
            body=json.dumps(request),
            modelId=self.model,
            accept="application/json",
            contentType="application/json",
        )

        response_body = json.loads(response.get("body").read())

        try:
            image_bytes = self.image_generation_model_driver.get_generated_
↪image(response_body)
        except Exception as e:
            raise ValueError(f"Inpainting generation failed: {e}") from e

        return image_bytes
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `image_height = field(default=512, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `image_width = field(default=512, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `seed = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute

_make_request(request)

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
def _make_request(self, request: dict) -> bytes:
    response = self.client.invoke_model(
        body=json.dumps(request),
        modelId=self.model,
        accept="application/json",
        contentType="application/json",
    )

    response_body = json.loads(response.get("body").read())

    try:
        image_bytes = self.image_generation_model_driver.get_generated_image(response_
↪body)
    except Exception as e:
```

(continues on next page)

(continued from previous page)

```
        raise ValueError(f"Inpainting generation failed: {e}") from e

    return image_bytes
```

client()

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
@lazy_property()
def client(self) -> BedrockRuntimeClient:
    return self.session.client("bedrock-runtime")
```

try_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    request = self.image_generation_model_driver.image_inpainting_request_parameters(
        prompts,
        image=image,
        mask=mask,
        negative_prompts=negative_prompts,
        seed=self.seed,
    )

    image_bytes = self._make_request(request)

    return ImageArtifact(
        value=image_bytes,
        format="png",
        width=image.width,
        height=image.height,
        meta={"prompt": ", ".join(prompts), "model": self.model},
    )
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
def try_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
```

(continues on next page)

(continued from previous page)

```
) -> ImageArtifact:
    request = self.image_generation_model_driver.image_outpainting_request_parameters(
        prompts,
        image=image,
        mask=mask,
        negative_prompts=negative_prompts,
        seed=self.seed,
    )

    image_bytes = self._make_request(request)

    return ImageArtifact(
        value=image_bytes,
        format="png",
        width=image.width,
        height=image.height,
        meta={"prompt": ", ".join(prompts), "model": self.model},
    )
```

try_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
def try_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    request = self.image_generation_model_driver.image_variation_request_parameters(
        prompts,
        image=image,
        negative_prompts=negative_prompts,
        seed=self.seed,
    )

    image_bytes = self._make_request(request)

    return ImageArtifact(
        value=image_bytes,
        format="png",
        width=image.width,
        height=image.height,
        meta={"prompt": ", ".join(prompts), "model": self.model},
    )
```

try_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/amazon_bedrock_image_generation_driver.py

```
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]
↪= None) -> ImageArtifact:
    request = self.image_generation_model_driver.text_to_image_request_parameters(
        prompts,
        self.image_width,
        self.image_height,
        negative_prompts=negative_prompts,
        seed=self.seed,
    )

    image_bytes = self._make_request(request)

    return ImageArtifact(
        value=image_bytes,
        format="png",
        width=self.image_width,
        height=self.image_height,
        meta={"prompt": ", ".join(prompts), "model": self.model},
    )
```

azure_openai_image_generation_driver

Bases: *OpenAiImageGenerationDriver* “)

Attributes

Name	Type	Description
<a href="azure_openai_image_generation_driver.mdx#griptape.drivers.image_generation.azure_openai_image_generation_driver.AzureOpenAiImageGenerationDriver.azure_deployment" title="azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.azure_openai_image_generation_driver.AzureOpenAiImageGenerationDriver.azure_deployment)"><code class="language-python">	*str*	An optional Azure OpenAi deployment id. Defaults to the model name.

continues on next page

Table 163 – continued from previous page

Name	Type	Description
<a href="azure_openai_image_generation_driver.mdx#griptape.drivers.image_generation.azure_openai_image_generation_driver.AzureOpenAiImageGenerationDriver.azure_endpoint" title="azure_endpoint = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.azure_openai_image_generation_driver.AzureOpenAiImageGenerationDriver.azure_endpoint)"><code class="language-python">	*str*	An Azure OpenAi endpoint.

continues on next page

Table 163 – continued from previous page

Name	Type	Description
<a href="azure_openai_image_generation_driver.mdx#griptape.drivers.image_generation.azure_openai_image_generation_driver.AzureOpenAiImageGenerationDriver.azure_ad_token" title="azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.image_generation.azure_openai_image_generation_driver.AzureOpenAiImageGenerationDriver.azure_ad_token)"><code class="language-python">	*Optional[str]*	An optional Azure Active Directory token.

continues on next page

Table 163 – continued from previous page

Name	Type	Description
<a href="azure_openai_image_generation_driver.mdx#griptape.drivers.image_generation.azure_openai_image_generation_driver.AzureOpenAiImageGenerationDriver.azure_ad_token_provider" title="azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.image_generation.azure_openai_image_generation_driver.AzureOpenAiImageGenerationDriver.azure_ad_token_provider)"><code class="language-python">	*Optional[Callable[[], str]]*	An optional Azure Active Directory token provider.

continues on next page

Table 163 – continued from previous page

Name	Type	Description
<a href="azure_openai_image_generation_driver.mdx#griptape.drivers.image_generation.azure_openai_image_generation_driver.AzureOpenAiImageGenerationDriver.api_version" title="api_version = field(default='2024-02-01', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.azure_openai_image_generation_driver.AzureOpenAiImageGenerationDriver.api_version)"><code class="language-python">	*str*	An Azure OpenAi API version.
<a href="azure_openai_image_generation_driver.mdx#griptape.drivers.image_generation.azure_openai_image_generation_driver.AzureOpenAiImageGenerationDriver.client" title="client() (griptape.drivers.image_generation.azure_openai_image_generation_driver.AzureOpenAiImageGenerationDriver.client)"><code class="language-python">	*AzureOpenAI*	An *openai.AzureOpenAI* client.

Source Code in griptape/drivers/image_generation/azure_openai_image_generation_driver.py

```
@define
class AzureOpenAiImageGenerationDriver(OpenAiImageGenerationDriver):
```

(continues on next page)

(continued from previous page)

```
    """Driver for Azure-hosted OpenAI image generation API.

    Attributes:
        azure_deployment: An optional Azure OpenAi deployment id. Defaults to the␣
↪model name.
        azure_endpoint: An Azure OpenAi endpoint.
        azure_ad_token: An optional Azure Active Directory token.
        azure_ad_token_provider: An optional Azure Active Directory token provider.
        api_version: An Azure OpenAi API version.
        client: An `openai.AzureOpenAI` client.
    """

    azure_deployment: str = field(
        kw_only=True,
        default=Factory(lambda self: self.model, takes_self=True),
        metadata={"serializable": True},
    )
    azure_endpoint: str = field(kw_only=True, metadata={"serializable": True})
    azure_ad_token: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": False})
    azure_ad_token_provider: Optional[Callable[[], str]] = field(
        kw_only=True,
        default=None,
        metadata={"serializable": False},
    )
    api_version: str = field(default="2024-02-01", kw_only=True, metadata={
↪"serializable": True})
    _client: Optional[openai.AzureOpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.AzureOpenAI:
        return openai.AzureOpenAI(
            organization=self.organization,
            api_key=self.api_key,
            api_version=self.api_version,
            azure_endpoint=self.azure_endpoint,
            azure_deployment=self.azure_deployment,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_version = field(default='2024-02-01', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute

- `azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True})` class-attribute instance-attribute
- `azure_endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/image_generation/azure_openai_image_generation_driver.py

```
@lazy_property()
def client(self) -> openai.AzureOpenAI:
    return openai.AzureOpenAI(
        organization=self.organization,
        api_key=self.api_key,
        api_version=self.api_version,
        azure_endpoint=self.azure_endpoint,
        azure_deployment=self.azure_deployment,
        azure_ad_token=self.azure_ad_token,
        azure_ad_token_provider=self.azure_ad_token_provider,
    )
```

base_image_generation_driver

Bases: *SerializableMixin* “) , *ExponentialBackoffMixin* ”) , ABC

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
@define
class BaseImageGenerationDriver(SerializableMixin, ExponentialBackoffMixin, ABC):
    model: str = field(kw_only=True, metadata={"serializable": True})

    def before_run(self, prompts: list[str], negative_prompts: Optional[list[str]] =␣
↪None) -> None:
        EventBus.publish_event(StartImageGenerationEvent(prompts=prompts, negative_
↪prompts=negative_prompts))

    def after_run(self) -> None:
        EventBus.publish_event(FinishImageGenerationEvent())

    def run_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact:
        for attempt in self.retrying():
            with attempt:
                self.before_run(prompts, negative_prompts)
                result = self.try_text_to_image(prompts, negative_prompts)
                self.after_run()

                return result

        raise Exception("Failed to run text to image generation")

    def run_image_variation(
        self,
```

(continues on next page)

(continued from previous page)

```
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        for attempt in self.retrying():
            with attempt:
                self.before_run(prompts, negative_prompts)
                result = self.try_image_variation(prompts, image, negative_prompts)
                self.after_run()

                return result

        raise Exception("Failed to generate image variations")

    def run_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        for attempt in self.retrying():
            with attempt:
                self.before_run(prompts, negative_prompts)
                result = self.try_image_inpainting(prompts, image, mask, negative_
↪prompts)
                self.after_run()

                return result

        raise Exception("Failed to run image inpainting")

    def run_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        for attempt in self.retrying():
            with attempt:
                self.before_run(prompts, negative_prompts)
                result = self.try_image_outpainting(prompts, image, mask, negative_
↪prompts)
                self.after_run()

                return result

        raise Exception("Failed to run image outpainting")

    @abstractmethod
    def try_text_to_image(self, prompts: list[str], negative_prompts:␣
```

(continues on next page)

(continued from previous page)

```
↪Optional[list[str]] = None) -> ImageArtifact: ...

    @abstractmethod
    def try_image_variation(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact: ...

    @abstractmethod
    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact: ...

    @abstractmethod
    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact: ...
```

- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

after_run()

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
def after_run(self) -> None:
    EventBus.publish_event(FinishImageGenerationEvent())
```

before_run(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
def before_run(self, prompts: list[str], negative_prompts: Optional[list[str]] =␣
↪None) -> None:
    EventBus.publish_event(StartImageGenerationEvent(prompts=prompts, negative_
↪prompts=negative_prompts))
```

run_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
def run_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    for attempt in self.retrying():
        with attempt:
            self.before_run(prompts, negative_prompts)
            result = self.try_image_inpainting(prompts, image, mask, negative_prompts)
            self.after_run()

            return result

    raise Exception("Failed to run image inpainting")
```

run_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
def run_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    for attempt in self.retrying():
        with attempt:
            self.before_run(prompts, negative_prompts)
            result = self.try_image_outpainting(prompts, image, mask, negative_
↪prompts)
            self.after_run()

            return result

    raise Exception("Failed to run image outpainting")
```

run_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
def run_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    for attempt in self.retrying():
        with attempt:
            self.before_run(prompts, negative_prompts)
```

(continues on next page)

(continued from previous page)

```
            result = self.try_image_variation(prompts, image, negative_prompts)
            self.after_run()

            return result

    raise Exception("Failed to generate image variations")
```

run_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
def run_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    for attempt in self.retrying():
        with attempt:
            self.before_run(prompts, negative_prompts)
            result = self.try_text_to_image(prompts, negative_prompts)
            self.after_run()

            return result

    raise Exception("Failed to run text to image generation")
```

try_image_inpainting(prompts, image, mask, negative_prompts=None)abstractmethod

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
@abstractmethod
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact: ...
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)abstractmethod

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
@abstractmethod
def try_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact: ...
```

try_image_variation(prompts, image, negative_prompts=None)abstractmethod

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
@abstractmethod
def try_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact: ...
```

try_text_to_image(prompts, negative_prompts=None)abstractmethod

Source Code in griptape/drivers/image_generation/base_image_generation_driver.py

```
@abstractmethod
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact: ...
```

base_multi_model_image_generation_driver

Bases: *BaseImageGenerationDriver* “) , ABC

Attributes

Name	Type	Description
<a href="base_multi_model_age_generation_driver.mdx#griptape.drivers.image_generation.base_multi_model_image_generation_driver.BaseMultiModelImageGenerationDriver.image_generation_model_driver" title="image_generation_model_driver = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.base_multi_model_image_generation_driver.BaseMultiModelImageGenerationDriver.image_generation_model_driver)"><co class="language-python">	<a href="../image_generation_model#griptape.drivers.image_generation_model.BaseImageGenerationModelDriver" title="BaseImageGenerationModelDriver (griptape.drivers.image_generation_model.BaseImageGenerationModelDriver)"><code class="language-python">BaseImageGenerationModel	Image Model Driver to use.

Source Code in griptape/drivers/image_generation/base_multi_model_image_generation_driver.py

```
@define
class BaseMultiModelImageGenerationDriver(BaseImageGenerationDriver, ABC):
    """Image Generation Driver for platforms like Amazon Bedrock that host many LLM␣
↪models.

    Instances of this Image Generation Driver require a Image Generation Model Driver␣
↪which is used to structure the
    image generation request in the format required by the model and to process the␣
↪output.

    Attributes:
        image_generation_model_driver: Image Model Driver to use.
    """

    image_generation_model_driver: BaseImageGenerationModelDriver = field(kw_␣
↪only=True, metadata={"serializable": True})
```

- `image_generation_model_driver = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

dummy

- `__all__ = ['DummyImageGenerationDriver']` module-attribute

Bases: *BaseImageGenerationDriver* “)

Source Code in griptape/drivers/image_generation/dummy_image_generation_driver.py

```
@define
class DummyImageGenerationDriver(BaseImageGenerationDriver):
    model: None = field(init=False, default=None, kw_only=True)

    def try_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact:
        raise DummyError(__class__.__name__, "try_text_to_image")

    def try_image_variation(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise DummyError(__class__.__name__, "try_image_variation")

    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise DummyError(__class__.__name__, "try_image_inpainting")

    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise DummyError(__class__.__name__, "try_image_outpainting")
```

- `model = field(init=False, default=None, kw_only=True)` class-attribute instance-attribute

try_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/dummy_image_generation_driver.py

```
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
```

(continues on next page)

(continued from previous page)

```
) -> ImageArtifact:
    raise DummyError(__class__.__name__, "try_image_inpainting")
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/dummy_image_generation_driver.py

```
def try_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise DummyError(__class__.__name__, "try_image_outpainting")
```

try_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/dummy_image_generation_driver.py

```
def try_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise DummyError(__class__.__name__, "try_image_variation")
```

try_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/dummy_image_generation_driver.py

```
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    raise DummyError(__class__.__name__, "try_text_to_image")
```

dummy_image_generation_driver

Bases: *BaseImageGenerationDriver* “)

Source Code in griptape/drivers/image_generation/dummy_image_generation_driver.py

```
@define
class DummyImageGenerationDriver(BaseImageGenerationDriver):
    model: None = field(init=False, default=None, kw_only=True)

    def try_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact:
        raise DummyError(__class__.__name__, "try_text_to_image")

    def try_image_variation(
```

(continues on next page)

(continued from previous page)

```
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise DummyError(__class__.__name__, "try_image_variation")

    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise DummyError(__class__.__name__, "try_image_inpainting")

    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise DummyError(__class__.__name__, "try_image_outpainting")
```

- `model = field(init=False, default=None, kw_only=True)` class-attribute instance-attribute

try_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/dummy_image_generation_driver.py

```
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise DummyError(__class__.__name__, "try_image_inpainting")
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/dummy_image_generation_driver.py

```
def try_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise DummyError(__class__.__name__, "try_image_outpainting")
```

try_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/dummy_image_generation_driver.py

```
def try_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise DummyError(__class__.__name__, "try_image_variation")
```

try_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/dummy_image_generation_driver.py

```
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    raise DummyError(__class__.__name__, "try_text_to_image")
```

griptape_cloud

- `__all__ = ['GriptapeCloudImageGenerationDriver']` module-attribute

Bases: *BaseImageGenerationDriver* “)

Source Code in griptape/drivers/image_generation/griptape_cloud_image_generation_driver.py

```
@define
class GriptapeCloudImageGenerationDriver(BaseImageGenerationDriver):
    model: Optional[str] = field(default=None, kw_only=True)
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]))
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True), kw_only=True
    )
    style: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    quality: Literal["standard", "hd"] = field(default="standard", kw_only=True,␣
↪metadata={"serializable": True})
    image_size: Literal["1024x1024", "1024x1792", "1792x1024"] = field(
        default="1024x1024", kw_only=True, metadata={"serializable": True}
    )

    def try_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact:
        url = griptape_cloud_url(self.base_url, "api/images/generations")

        response = requests.post(
            url,
```

(continues on next page)

(continued from previous page)

```
            headers=self.headers,
            json={
                "prompts": prompts,
                "driver_configuration": {
                    "model": self.model,
                    "image_size": self.image_size,
                    "quality": self.quality,
                    "style": self.style,
                },
            },
        )
        response.raise_for_status()
        response = response.json()

        return ImageArtifact.from_dict(response["artifact"])

    def try_image_variation(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support image␣
↪variation")

    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪inpainting")

    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪outpainting")
```

- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']))` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')))` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute

- `image_size = field(default='1024x1024', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(default=None, kw_only=True)` class-attribute instance-attribute
- `quality = field(default='standard', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `style = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

try_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/griptape_cloud_image_generation_driver.py

```
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support inpainting
↪")
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/griptape_cloud_image_generation_driver.py

```
def try_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support outpainting
↪")
```

try_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/griptape_cloud_image_generation_driver.py

```
def try_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support image␣
↪variation")
```

try_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/griptape_cloud_image_generation_driver.py

```
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    url = griptape_cloud_url(self.base_url, "api/images/generations")

    response = requests.post(
        url,
        headers=self.headers,
        json={
            "prompts": prompts,
            "driver_configuration": {
                "model": self.model,
                "image_size": self.image_size,
                "quality": self.quality,
                "style": self.style,
            },
        },
    )
    response.raise_for_status()
    response = response.json()

    return ImageArtifact.from_dict(response["artifact"])
```

griptape_cloud_image_generation_driver

Bases: *BaseImageGenerationDriver* “)

Source Code in griptape/drivers/image_generation/griptape_cloud_image_generation_driver.py

```
@define
class GriptapeCloudImageGenerationDriver(BaseImageGenerationDriver):
    model: Optional[str] = field(default=None, kw_only=True)
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]))
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True), kw_only=True
    )
    style: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    quality: Literal["standard", "hd"] = field(default="standard", kw_only=True,␣
↪metadata={"serializable": True})
    image_size: Literal["1024x1024", "1024x1792", "1792x1024"] = field(
        default="1024x1024", kw_only=True, metadata={"serializable": True}
    )

    def try_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact:
```

(continues on next page)

(continued from previous page)

```
        url = griptape_cloud_url(self.base_url, "api/images/generations")

        response = requests.post(
            url,
            headers=self.headers,
            json={
                "prompts": prompts,
                "driver_configuration": {
                    "model": self.model,
                    "image_size": self.image_size,
                    "quality": self.quality,
                    "style": self.style,
                },
            },
        )
        response.raise_for_status()
        response = response.json()

        return ImageArtifact.from_dict(response["artifact"])

    def try_image_variation(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support image␣
↪variation")

    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪inpainting")

    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪outpainting")
```

- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']))`
  class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://`

`cloud.griptape.ai'))` class-attribute instance-attribute

- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `image_size = field(default='1024x1024', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(default=None, kw_only=True)` class-attribute instance-attribute
- `quality = field(default='standard', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `style = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

try_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/griptape_cloud_image_generation_driver.py

```
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support inpainting
↪")
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/griptape_cloud_image_generation_driver.py

```
def try_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support outpainting
↪")
```

try_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/griptape_cloud_image_generation_driver.py

```
def try_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support image␣
↪variation")
```

try_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/griptape_cloud_image_generation_driver.py

```
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    url = griptape_cloud_url(self.base_url, "api/images/generations")

    response = requests.post(
        url,
        headers=self.headers,
        json={
            "prompts": prompts,
            "driver_configuration": {
                "model": self.model,
                "image_size": self.image_size,
                "quality": self.quality,
                "style": self.style,
            },
        },
    )
    response.raise_for_status()
    response = response.json()

    return ImageArtifact.from_dict(response["artifact"])
```

huggingface_pipeline

- `__all__ = ['HuggingFacePipelineImageGenerationDriver']` module-attribute

Bases: *BaseImageGenerationDriver* “) , ABC

Attributes

Name	Type	Description
<a href="huggingface_pipeline.mdx#griptape.drivers.image_generation.huggingface_pipeline.HuggingFacePipelineImageGenerationDriver.pipeline_driver" title="pipeline_driver = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.huggingface_pipeline.HuggingFacePipelineImageGenerationDriver.pipeline_driver"><code class="language-python">	<a href="../image_generation_pipeline#griptape.drivers.image_generation_pipeline.BaseDiffusionImageGenerationPipelineDriver" title="BaseDiffusionImageGenerationPipelineDriver (griptape.drivers.image_generation_pipeline.BaseDiffusionImageGenerationPipelineDriver)"><code class="language-python">BaseDiffusionImageGenerat	A pipeline image generation model driver typed for the specific pipeline required by the model.
<a href="huggingface_pipeline.mdx#griptape.drivers.image_generation.huggingface_pipeline.HuggingFacePipelineImageGenerationDriver.device" title="device = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.huggingface_pipeline.HuggingFacePipelineImageGenerationDriver.device)"><code class="language-python">	*Optional[str]*	The hardware device used for inference. For example, "cpu", "cuda", or "mps".

continues on next page

Table 165 – continued from previous page

Name	Type	Description
<a href="huggingface_pipeline.mdx#griptape.drivers.image_generation.huggingface_pipeline.HuggingFacePipelineImageGenerationDriver.output_format" title="output_format = field(default='png', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.huggingface_pipeline.HuggingFacePipelineImageGenerationDriver.output_format)"><code class="language-python">	*str*	The format the generated image is returned in. Defaults to "png".

Source Code in griptape/drivers/image_generation/huggingface_pipeline_image_generation_driver.py

```
@define
class HuggingFacePipelineImageGenerationDriver(BaseImageGenerationDriver, ABC):
    """Image generation driver for models hosted by Hugging Faces Diffusion Pipeline.

    For more information, see the HuggingFace documentation for Diffusers:
            https://huggingface.co/docs/diffusers/en/index

    Attributes:
        pipeline_driver: A pipeline image generation model driver typed for the␣
↪specific pipeline required by the model.
        device: The hardware device used for inference. For example, "cpu", "cuda",␣
↪or "mps".
        output_format: The format the generated image is returned in. Defaults to "png
↪".
    """

    pipeline_driver: BaseDiffusionImageGenerationPipelineDriver = field(kw_only=True,␣
↪metadata={"serializable": True})
    device: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    output_format: str = field(default="png", kw_only=True, metadata={"serializable":␣
↪True})

    def try_text_to_image(self, prompts: list[str], negative_prompts:␣
```

(continues on next page)

(continued from previous page)

```
↪Optional[list[str]] = None) -> ImageArtifact:
        pipeline = self.pipeline_driver.prepare_pipeline(self.model, self.device)

        prompt = ", ".join(prompts)
        output_image = pipeline(
            prompt, **self.pipeline_driver.make_additional_params(negative_prompts,␣
↪self.device)
        ).images[0]

        buffer = io.BytesIO()
        output_image.save(buffer, format=self.output_format.upper())

        return ImageArtifact(
            value=buffer.getvalue(),
            format=self.output_format.lower(),
            height=output_image.height,
            width=output_image.width,
            meta={"prompt": prompt},
        )

    def try_image_variation(
        self, prompts: list[str], image: ImageArtifact, negative_prompts:␣
↪Optional[list[str]] = None
    ) -> ImageArtifact:
        pil_image = import_optional_dependency("PIL.Image")

        pipeline = self.pipeline_driver.prepare_pipeline(self.model, self.device)

        prompt = ", ".join(prompts)
        input_image = pil_image.open(io.BytesIO(image.value))
        # The size of the input image drives the size of the output image.
        # Resize the input image to the configured dimensions.
        output_width, output_height = self.pipeline_driver.output_image_dimensions
        if input_image.height != output_height or input_image.width != output_width:
            input_image = input_image.resize((output_width, output_height))

        output_image = pipeline(
            prompt,
            **self.pipeline_driver.make_image_param(input_image),
            **self.pipeline_driver.make_additional_params(negative_prompts, self.
↪device),
        ).images[0]

        buffer = io.BytesIO()
        output_image.save(buffer, format=self.output_format.upper())

        return ImageArtifact(
            value=buffer.getvalue(),
            format=self.output_format.lower(),
            height=output_image.height,
            width=output_image.width,
            meta={"prompt": prompt},
```

(continues on next page)

(continued from previous page)

```
        )

    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError("Inpainting is not supported by this driver.")

    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError("Outpainting is not supported by this driver.")
```

- `device = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output_format = field(default='png', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `pipeline_driver = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

try_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/huggingface_pipeline_image_generation_driver.py

```
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError("Inpainting is not supported by this driver.")
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/huggingface_pipeline_image_generation_driver.py

```
def try_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
```

(continues on next page)

(continued from previous page)

```
) -> ImageArtifact:
    raise NotImplementedError("Outpainting is not supported by this driver.")
```

try_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/huggingface_pipeline_image_generation_driver.py

```
def try_image_variation(
    self, prompts: list[str], image: ImageArtifact, negative_prompts:␣
↪Optional[list[str]] = None
) -> ImageArtifact:
    pil_image = import_optional_dependency("PIL.Image")

    pipeline = self.pipeline_driver.prepare_pipeline(self.model, self.device)

    prompt = ", ".join(prompts)
    input_image = pil_image.open(io.BytesIO(image.value))
    # The size of the input image drives the size of the output image.
    # Resize the input image to the configured dimensions.
    output_width, output_height = self.pipeline_driver.output_image_dimensions
    if input_image.height != output_height or input_image.width != output_width:
        input_image = input_image.resize((output_width, output_height))

    output_image = pipeline(
        prompt,
        **self.pipeline_driver.make_image_param(input_image),
        **self.pipeline_driver.make_additional_params(negative_prompts, self.device),
    ).images[0]

    buffer = io.BytesIO()
    output_image.save(buffer, format=self.output_format.upper())

    return ImageArtifact(
        value=buffer.getvalue(),
        format=self.output_format.lower(),
        height=output_image.height,
        width=output_image.width,
        meta={"prompt": prompt},
    )
```

try_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/huggingface_pipeline_image_generation_driver.py

```
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    pipeline = self.pipeline_driver.prepare_pipeline(self.model, self.device)

    prompt = ", ".join(prompts)
    output_image = pipeline(
        prompt, **self.pipeline_driver.make_additional_params(negative_prompts, self.
```

(continues on next page)

(continued from previous page)

```
↪device)
    ).images[0]

    buffer = io.BytesIO()
    output_image.save(buffer, format=self.output_format.upper())

    return ImageArtifact(
        value=buffer.getvalue(),
        format=self.output_format.lower(),
        height=output_image.height,
        width=output_image.width,
        meta={"prompt": prompt},
    )
```

huggingface_pipeline_image_generation_driver

Bases: *BaseImageGenerationDriver* “) , ABC

Attributes

Name	Type	Description
<a href="hugging-face_pipeline_im-age_genera-tion_driver.mdx#grip-tape.drivers.im-age_generation.hug-gingface_pipeline_im-age_genera-tion_driver.Hug-gingFacePipelineIm-ageGenera-tionDriver.pipeline_drive title="pipeline_driver = field(kw_only=True, metadata={'seri-alizable': True}) class-attribute instance-attribute (griptape.drivers.im-age_generation.hug-gingface_pipeline_im-age_genera-tion_driver.Hug-gingFacePipelineIm-ageGenera-tionDriver.pipeline_drive class="language-python">	<a href="../image_generation_pipeline#grip-tape.drivers.image_generation_pipeline.BaseDiffu-sionImageGenerationPipelineDriver" title="BaseD-iffusionImageGenerationPipelineDriver (grip-tape.drivers.image_generation_pipeline.BaseDif-fusionImageGenerationPipelineDriver)"><code class="language-python">BaseDiffusionImageGenerat	A pipeline image generation model driver typed for the spe-cific pipeline required by the model.

continues on next page

Table 166 – continued from previous page

Name	Type	Description
<a href="hugging-face_pipeline_im-age_genera-tion_driver.mdx#grip-tape.drivers.im-age_generation.hug-gingface_pipeline_im-age_genera-tion_driver.Hugging-FacePipelineImage-GenerationDriver.de-vice" title="device = field(default=None, kw_only=True, meta-data={'serializable': True}) class-attribute instance-attribute (griptape.drivers.im-age_generation.hug-gingface_pipeline_im-age_genera-tion_driver.Hugging-FacePipelineImage-GenerationDriver.de-vice)"><code class="language-python">	*Optional[str]*	The hardware device used for inference. For example, "cpu", "cuda", or "mps".

continues on next page

Table 166 – continued from previous page

Name	Type	Description
<a href="huggingface_pipeline_image_generation_driver.mdx#griptape.drivers.image_generation.huggingface_pipeline_image_generation_driver.HuggingFacePipelineImageGenerationDriver.output_format" title="output_format = field(default='png', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.huggingface_pipeline_image_generation_driver.HuggingFacePipelineImageGenerationDriver.output_format)"><code class="language-python">	*str*	The format the generated image is returned in. Defaults to "png".

Source Code in griptape/drivers/image_generation/huggingface_pipeline_image_generation_driver.py

```
@define
class HuggingFacePipelineImageGenerationDriver(BaseImageGenerationDriver, ABC):
    """Image generation driver for models hosted by Hugging Faces Diffusion Pipeline.

    For more information, see the HuggingFace documentation for Diffusers:
            https://huggingface.co/docs/diffusers/en/index

    Attributes:
        pipeline_driver: A pipeline image generation model driver typed for the␣
↪specific pipeline required by the model.
        device: The hardware device used for inference. For example, "cpu", "cuda",␣
↪or "mps".
        output_format: The format the generated image is returned in. Defaults to "png
↪".
    """

    pipeline_driver: BaseDiffusionImageGenerationPipelineDriver = field(kw_only=True,␣
↪metadata={"serializable": True})
    device: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
```

(continues on next page)

(continued from previous page)

```
    output_format: str = field(default="png", kw_only=True, metadata={"serializable":␣
↪True})

    def try_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact:
        pipeline = self.pipeline_driver.prepare_pipeline(self.model, self.device)

        prompt = ", ".join(prompts)
        output_image = pipeline(
            prompt, **self.pipeline_driver.make_additional_params(negative_prompts,␣
↪self.device)
        ).images[0]

        buffer = io.BytesIO()
        output_image.save(buffer, format=self.output_format.upper())

        return ImageArtifact(
            value=buffer.getvalue(),
            format=self.output_format.lower(),
            height=output_image.height,
            width=output_image.width,
            meta={"prompt": prompt},
        )

    def try_image_variation(
        self, prompts: list[str], image: ImageArtifact, negative_prompts:␣
↪Optional[list[str]] = None
    ) -> ImageArtifact:
        pil_image = import_optional_dependency("PIL.Image")

        pipeline = self.pipeline_driver.prepare_pipeline(self.model, self.device)

        prompt = ", ".join(prompts)
        input_image = pil_image.open(io.BytesIO(image.value))
        # The size of the input image drives the size of the output image.
        # Resize the input image to the configured dimensions.
        output_width, output_height = self.pipeline_driver.output_image_dimensions
        if input_image.height != output_height or input_image.width != output_width:
            input_image = input_image.resize((output_width, output_height))

        output_image = pipeline(
            prompt,
            **self.pipeline_driver.make_image_param(input_image),
            **self.pipeline_driver.make_additional_params(negative_prompts, self.
↪device),
        ).images[0]

        buffer = io.BytesIO()
        output_image.save(buffer, format=self.output_format.upper())

        return ImageArtifact(
            value=buffer.getvalue(),
```

(continues on next page)

(continued from previous page)

```
            format=self.output_format.lower(),
            height=output_image.height,
            width=output_image.width,
            meta={"prompt": prompt},
        )

    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError("Inpainting is not supported by this driver.")

    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError("Outpainting is not supported by this driver.")
```

- `device = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output_format = field(default='png', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `pipeline_driver = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

try_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/huggingface_pipeline_image_generation_driver.py

```
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError("Inpainting is not supported by this driver.")
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/huggingface_pipeline_image_generation_driver.py

```
def try_image_outpainting(
    self,
    prompts: list[str],
```

(continues on next page)

(continued from previous page)

```
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError("Outpainting is not supported by this driver.")
```

try_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/huggingface_pipeline_image_generation_driver.py

```
def try_image_variation(
    self, prompts: list[str], image: ImageArtifact, negative_prompts:␣
↪Optional[list[str]] = None
) -> ImageArtifact:
    pil_image = import_optional_dependency("PIL.Image")

    pipeline = self.pipeline_driver.prepare_pipeline(self.model, self.device)

    prompt = ", ".join(prompts)
    input_image = pil_image.open(io.BytesIO(image.value))
    # The size of the input image drives the size of the output image.
    # Resize the input image to the configured dimensions.
    output_width, output_height = self.pipeline_driver.output_image_dimensions
    if input_image.height != output_height or input_image.width != output_width:
        input_image = input_image.resize((output_width, output_height))

    output_image = pipeline(
        prompt,
        **self.pipeline_driver.make_image_param(input_image),
        **self.pipeline_driver.make_additional_params(negative_prompts, self.device),
    ).images[0]

    buffer = io.BytesIO()
    output_image.save(buffer, format=self.output_format.upper())

    return ImageArtifact(
        value=buffer.getvalue(),
        format=self.output_format.lower(),
        height=output_image.height,
        width=output_image.width,
        meta={"prompt": prompt},
    )
```

try_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/huggingface_pipeline_image_generation_driver.py

```
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    pipeline = self.pipeline_driver.prepare_pipeline(self.model, self.device)
```

(continues on next page)

(continued from previous page)

```
    prompt = ", ".join(prompts)
    output_image = pipeline(
        prompt, **self.pipeline_driver.make_additional_params(negative_prompts, self.
↪device)
    ).images[0]

    buffer = io.BytesIO()
    output_image.save(buffer, format=self.output_format.upper())

    return ImageArtifact(
        value=buffer.getvalue(),
        format=self.output_format.lower(),
        height=output_image.height,
        width=output_image.width,
        meta={"prompt": prompt},
    )
```

leonardo

- `__all__ = ['LeonardoImageGenerationDriver']` module-attribute

Bases: *BaseImageGenerationDriver* “)

Attributes

Name	Type	Description
<a href="base_image_generation_driver.mdx#griptape.drivers.image_generation.base_image_generation_driver.BaseImageGenerationDriver.model" title="model = field(kw_only=True, metadata={'serializable': True})">class-attribute instance-attribute (griptape.drivers.image_generation.leonardo.LeonardoImageGenerationDriver.model)"><cod class="language-python">	*str*	The ID of the model to use when generating images.

continues on next page

Table 167 – continued from previous page

Name	Type	Description
<a href="leonardo.mdx#griptape.drivers.image_generation.leonardo.LeonardoImageGenerationDriver.api_key" title="api_key = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.leonardo.LeonardoImageGenerationDriver.api_key)"><cc class="language-python">	*str*	The API key to use when making requests to the Leonardo API.
<a href="leonardo.mdx#griptape.drivers.image_generation.leonardo.LeonardoImageGenerationDriver.requests_session" title="requests_session = field(default=Factory(lambda: requests.Session()), kw_only=True) class-attribute instance-attribute (griptape.drivers.image_generation.leonardo.LeonardoImageGenerationDriver.requests_session)"><code class="language-python">	*Session*	The requests session to use when making requests to the Leonardo API.

continues on next page

Table 167 – continued from previous page

Name	Type	Description
<a href="leonardo.mdx#griptape.drivers.image_generation.leonardo.LeonardoImageGenerationDriver.api_base" title="api_base = 'https://cloud.leonardo.ai/api/rest/v1' class-attribute instance-attribute (griptape.drivers.image_generation.leonardo.LeonardoImageGenerationDriver.api_base)"><c class="language-python">	*str*	The base URL of the Leonardo API.
<a href="leonardo.mdx#griptape.drivers.image_generation.leonardo.LeonardoImageGenerationDriver.max_attempts" title="max_attempts = field(default=10, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.leonardo.LeonardoImageGenerationDriver.max_attempts)"><code class="language-python">	*int*	The maximum number of times to poll the Leonardo API for a completed image.

continues on next page

Table 167 – continued from previous page

Name	Type	Description
<a href="leonardo.mdx#griptape.drivers.image_generation.leonardo.LeonardoImageGenerationDriver.image_width" title="image_width = field(default=512, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.leonardo.LeonardoImageGenerationDriver.image_width)"><code class="language-python">	*int*	The width of the generated image in the range [32, 1024] and divisible by 8.
<a href="leonardo.mdx#griptape.drivers.image_generation.leonardo.LeonardoImageGenerationDriver.image_height" title="image_height = field(default=512, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.leonardo.LeonardoImageGenerationDriver.image_height)"><code class="language-python">	*int*	The height of the generated image in the range [32, 1024] and divisible by 8.

continues on next page

Table 167 – continued from previous page

Name	Type	Description
<a href="leonardo.mdx#griptape.drivers.image_generation.leonardo.LeonardoImageGenerationDriver.steps" title="steps = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.leonardo.LeonardoImageGenerationDriver.steps)"><code class="language-python">	*Optional[int]*	Optionally specify the number of inference steps to run for each image generation request, [30, 60].
<a href="leonardo.mdx#griptape.drivers.image_generation.leonardo.LeonardoImageGenerationDriver.seed" title="seed = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.leonardo.LeonardoImageGenerationDriver.seed)"><code class="language-python">	*Optional[int]*	Optionally provide a consistent seed to generation requests, increasing consistency in output.

continues on next page

Table 167 – continued from previous page

Name	Type	Description
<a href="leonardo.mdx#griptape.drivers.image_generation.leonardo.LeonardoImageGenerationDriver.init_strength" title="init_strength = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.leonardo.LeonardoImageGenerationDriver.init_strength)" class="language-python">	*Optional[float]*	Optionally specify the strength of the initial image, [0.0, 1.0].

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
@define
class LeonardoImageGenerationDriver(BaseImageGenerationDriver):
    """Driver for the Leonardo image generation API.

    Details on Leonardo image generation parameters can be found here:
    https://docs.leonardo.ai/reference/creategeneration

    Attributes:
        model: The ID of the model to use when generating images.
        api_key: The API key to use when making requests to the Leonardo API.
        requests_session: The requests session to use when making requests to the␣
↪Leonardo API.
        api_base: The base URL of the Leonardo API.
        max_attempts: The maximum number of times to poll the Leonardo API for a␣
↪completed image.
        image_width: The width of the generated image in the range [32, 1024] and␣
↪divisible by 8.
        image_height: The height of the generated image in the range [32, 1024] and␣
↪divisible by 8.
        steps: Optionally specify the number of inference steps to run for each image␣
↪generation request, [30, 60].
        seed: Optionally provide a consistent seed to generation requests, increasing␣
↪consistency in output.
        init_strength: Optionally specify the strength of the initial image, [0.0, 1.
↪0].
    """

    api_key: str = field(kw_only=True, metadata={"serializable": True})
```

(continues on next page)

(continued from previous page)

```
    requests_session: requests.Session = field(default=Factory(lambda: requests.
↪Session()), kw_only=True)
    api_base: str = "https://cloud.leonardo.ai/api/rest/v1"
    max_attempts: int = field(default=10, kw_only=True, metadata={"serializable":␣
↪True})
    image_width: int = field(default=512, kw_only=True, metadata={"serializable":␣
↪True})
    image_height: int = field(default=512, kw_only=True, metadata={"serializable":␣
↪True})
    steps: Optional[int] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    seed: Optional[int] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    init_strength: Optional[float] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    control_net: bool = field(default=False, kw_only=True, metadata={"serializable":␣
↪True})
    control_net_type: Optional[Literal["POSE", "CANNY", "DEPTH"]] = field(
        default=None,
        kw_only=True,
        metadata={"serializable": True},
    )

    def try_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact:
        if negative_prompts is None:
            negative_prompts = []

        generation_id = self._create_generation(prompts=prompts, negative_
↪prompts=negative_prompts)
        image_url = self._get_image_url(generation_id=generation_id)
        image_data = self._download_image(url=image_url)

        return ImageArtifact(
            value=image_data,
            format="png",
            width=self.image_width,
            height=self.image_height,
            meta={
                "model": self.model,
                "prompt": ", ".join(prompts),
            },
        )

    def try_image_variation(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        if negative_prompts is None:
            negative_prompts = []
```

(continues on next page)

(continued from previous page)

```
        init_image_id = self._upload_init_image(image)
        generation_id = self._create_generation(
            prompts=prompts,
            negative_prompts=negative_prompts,
            init_image_id=init_image_id,
        )
        image_url = self._get_image_url(generation_id=generation_id)
        image_data = self._download_image(url=image_url)

        return ImageArtifact(
            value=image_data,
            format="png",
            width=self.image_width,
            height=self.image_height,
            meta={
                "model": self.model,
                "prompt": ", ".join(prompts),
            },
        )

    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪outpainting")

    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪inpainting")

    def _upload_init_image(self, image: ImageArtifact) -> str:
        request = {"extension": image.mime_type.split("/")[1]}

        prep_response = self._make_api_request("/init-image", request=request)
        if prep_response is None or prep_response["uploadInitImage"] is None:
            raise Exception(f"failed to prepare init image: {prep_response}")

        fields = json.loads(prep_response["uploadInitImage"]["fields"])
        pre_signed_url = prep_response["uploadInitImage"]["url"]
        init_image_id = prep_response["uploadInitImage"]["id"]
```

(continues on next page)

(continued from previous page)

```
        files = {"file": image.value}
        upload_response = requests.post(pre_signed_url, data=fields, files=files)
        if not upload_response.ok:
            raise Exception(f"failed to upload init image: {upload_response.text}")

        return init_image_id

    def _create_generation(
        self,
        prompts: list[str],
        negative_prompts: list[str],
        init_image_id: Optional[str] = None,
    ) -> str:
        prompt = ", ".join(prompts)
        negative_prompt = ", ".join(negative_prompts)
        request = {
            "prompt": prompt,
            "negative_prompt": negative_prompt,
            "width": self.image_width,
            "height": self.image_height,
            "num_images": 1,
            "modelId": self.model,
        }

        if init_image_id is not None:
            request["init_image_id"] = init_image_id

        if self.init_strength is not None:
            request["init_strength"] = self.init_strength

        if self.steps:
            request["num_inference_steps"] = self.steps

        if self.seed is not None:
            request["seed"] = self.seed

        if self.control_net:
            request["controlNet"] = self.control_net
            request["controlNetType"] = self.control_net_type

        response = self._make_api_request("/generations", request=request)
        if response is None or response["sdGenerationJob"] is None:
            raise Exception(f"failed to create generation: {response}")

        return response["sdGenerationJob"]["generationId"]

    def _make_api_request(self, endpoint: str, request: dict, method: str = "POST") ->
↪ dict:
        url = f"{self.api_base}{endpoint}"
        headers = {"Authorization": f"Bearer {self.api_key}"}

        response = self.requests_session.request(url=url, method=method, json=request,
```

(continues on next page)

(continued from previous page)

```
↪ headers=headers)
        if not response.ok:
            raise Exception(f"failed to make API request: {response.text}")

        return response.json()

    def _get_image_url(self, generation_id: str) -> str:
        for attempt in range(self.max_attempts):
            response = self.requests_session.get(
                url=f"{self.api_base}/generations/{generation_id}",
                headers={"Authorization": f"Bearer {self.api_key}"},
            ).json()

            if response["generations_by_pk"]["status"] == "PENDING":
                time.sleep(attempt + 1)
                continue

            return response["generations_by_pk"]["generated_images"][0]["url"]
        raise Exception("image generation failed to complete")

    def _download_image(self, url: str) -> bytes:
        response = self.requests_session.get(url=url, headers={"Authorization": f
↪"Bearer {self.api_key}"})

        return response.content
```

- `api_base = 'https://cloud.leonardo.ai/api/rest/v1'` class-attribute instance-attribute
- `api_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `control_net = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `control_net_type = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `image_height = field(default=512, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `image_width = field(default=512, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `init_strength = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `max_attempts = field(default=10, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `requests_session = field(default=Factory(lambda: requests.Session()), kw_only=True)` class-attribute instance-attribute
- `seed = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `steps = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_create_generation(prompts, negative_prompts, init_image_id=None)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def _create_generation(
    self,
    prompts: list[str],
    negative_prompts: list[str],
    init_image_id: Optional[str] = None,
) -> str:
    prompt = ", ".join(prompts)
    negative_prompt = ", ".join(negative_prompts)
    request = {
        "prompt": prompt,
        "negative_prompt": negative_prompt,
        "width": self.image_width,
        "height": self.image_height,
        "num_images": 1,
        "modelId": self.model,
    }

    if init_image_id is not None:
        request["init_image_id"] = init_image_id

    if self.init_strength is not None:
        request["init_strength"] = self.init_strength

    if self.steps:
        request["num_inference_steps"] = self.steps

    if self.seed is not None:
        request["seed"] = self.seed

    if self.control_net:
        request["controlNet"] = self.control_net
        request["controlNetType"] = self.control_net_type

    response = self._make_api_request("/generations", request=request)
    if response is None or response["sdGenerationJob"] is None:
        raise Exception(f"failed to create generation: {response}")

    return response["sdGenerationJob"]["generationId"]
```

_download_image(url)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def _download_image(self, url: str) -> bytes:
    response = self.requests_session.get(url=url, headers={"Authorization": f"Bearer
↪{self.api_key}"})

    return response.content
```

_get_image_url(generation_id)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def _get_image_url(self, generation_id: str) -> str:
    for attempt in range(self.max_attempts):
        response = self.requests_session.get(
            url=f"{self.api_base}/generations/{generation_id}",
            headers={"Authorization": f"Bearer {self.api_key}"},
        ).json()

        if response["generations_by_pk"]["status"] == "PENDING":
            time.sleep(attempt + 1)
            continue

        return response["generations_by_pk"]["generated_images"][0]["url"]
    raise Exception("image generation failed to complete")
```

_make_api_request(endpoint, request, method='POST')

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def _make_api_request(self, endpoint: str, request: dict, method: str = "POST") ->␣
↪dict:
    url = f"{self.api_base}{endpoint}"
    headers = {"Authorization": f"Bearer {self.api_key}"}

    response = self.requests_session.request(url=url, method=method, json=request,␣
↪headers=headers)
    if not response.ok:
        raise Exception(f"failed to make API request: {response.text}")

    return response.json()
```

_upload_init_image(image)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def _upload_init_image(self, image: ImageArtifact) -> str:
    request = {"extension": image.mime_type.split("/")[1]}

    prep_response = self._make_api_request("/init-image", request=request)
    if prep_response is None or prep_response["uploadInitImage"] is None:
        raise Exception(f"failed to prepare init image: {prep_response}")

    fields = json.loads(prep_response["uploadInitImage"]["fields"])
    pre_signed_url = prep_response["uploadInitImage"]["url"]
    init_image_id = prep_response["uploadInitImage"]["id"]

    files = {"file": image.value}
    upload_response = requests.post(pre_signed_url, data=fields, files=files)
    if not upload_response.ok:
        raise Exception(f"failed to upload init image: {upload_response.text}")
```

(continues on next page)

(continued from previous page)

```
    return init_image_id
```

try_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support inpainting
↪")
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def try_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support outpainting
↪")
```

try_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def try_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    if negative_prompts is None:
        negative_prompts = []

    init_image_id = self._upload_init_image(image)
    generation_id = self._create_generation(
        prompts=prompts,
        negative_prompts=negative_prompts,
        init_image_id=init_image_id,
    )
    image_url = self._get_image_url(generation_id=generation_id)
```

(continues on next page)

(continued from previous page)

```
    image_data = self._download_image(url=image_url)

    return ImageArtifact(
        value=image_data,
        format="png",
        width=self.image_width,
        height=self.image_height,
        meta={
            "model": self.model,
            "prompt": ", ".join(prompts),
        },
    )
```

try_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    if negative_prompts is None:
        negative_prompts = []

    generation_id = self._create_generation(prompts=prompts, negative_
↪prompts=negative_prompts)
    image_url = self._get_image_url(generation_id=generation_id)
    image_data = self._download_image(url=image_url)

    return ImageArtifact(
        value=image_data,
        format="png",
        width=self.image_width,
        height=self.image_height,
        meta={
            "model": self.model,
            "prompt": ", ".join(prompts),
        },
    )
```

leonardo_image_generation_driver

Bases: *BaseImageGenerationDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.image_generation.BaseImageGenerationDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.leonardo_image_generation_driver.LeonardoImageGenerationDriver.model)"><cod class="language-python">	*str*	The ID of the model to use when generating images.
<a href="leonardo_image_generation_driver.mdx#griptape.drivers.image_generation.leonardo_image_generation_driver.LeonardoImageGenerationDriver.api_key" title="api_key = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.leonardo_image_generation_driver.LeonardoImageGenerationDriver.api_key)"><cc class="language-python">	*str*	The API key to use when making requests to the Leonardo API.

continues on next page

Table 168 – continued from previous page

Name	Type	Description
<a href="leonardo_image_generation_driver.mdx#griptape.drivers.image_generation.leonardo_image_generation_driver.LeonardoImageGenerationDriver.requests_session" title="requests_session = field(default=Factory(lambda: requests.Session()), kw_only=True) class-attribute instance-attribute (griptape.drivers.image_generation.leonardo_image_generation_driver.LeonardoImageGenerationDriver.requests_session)"><code class="language-python">	*Session*	The requests session to use when making requests to the Leonardo API.

continues on next page

Table 168 – continued from previous page

Name	Type	Description
<a href="leonardo_image_generation_driver.mdx#griptape.drivers.image_generation.leonardo_image_generation_driver.LeonardoImageGenerationDriver.api_base" title="api_base = 'https://cloud.leonardo.ai/api/rest/v1' class-attribute instance-attribute (griptape.drivers.image_generation.leonardo_image_generation_driver.LeonardoImageGenerationDriver.api_base)"><c class="language-python">	*str*	The base URL of the Leonardo API.
<a href="leonardo_image_generation_driver.mdx#griptape.drivers.image_generation.leonardo_image_generation_driver.LeonardoImageGenerationDriver.max_attempts" title="max_attempts = field(default=10, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.leonardo_image_generation_driver.LeonardoImageGenerationDriver.max_attempts)"><code class="language-python">	*int*	The maximum number of times to poll the Leonardo API for a completed image.

continues on next page

Table 168 – continued from previous page

Name	Type	Description
<a href="leonardo_image_generation_driver.mdx#griptape.drivers.image_generation.leonardo_image_generation_driver.LeonardoImageGenerationDriver.image_width" title="image_width = field(default=512, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.leonardo_image_generation_driver.LeonardoImageGenerationDriver.image_width)"><code class="language-python">	*int*	The width of the generated image in the range [32, 1024] and divisible by 8.

continues on next page

Table 168 – continued from previous page

Name	Type	Description
<a href="leonardo_image_generation_driver.mdx#griptape.drivers.image_generation.leonardo_image_generation_driver.LeonardoImageGenerationDriver.image_height" title="image_height = field(default=512, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.leonardo_image_generation_driver.LeonardoImageGenerationDriver.image_height)"><code class="language-python">	*int*	The height of the generated image in the range [32, 1024] and divisible by 8.
<a href="leonardo_image_generation_driver.mdx#griptape.drivers.image_generation.leonardo_image_generation_driver.LeonardoImageGenerationDriver.steps" title="steps = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.leonardo_image_generation_driver.LeonardoImageGenerationDriver.steps)"><code class="language-python">	*Optional[int]*	Optionally specify the number of inference steps to run for each image generation request, [30, 60].

continues on next page

Table 168 – continued from previous page

Name	Type	Description
<a href="leonardo_image_generation_driver.mdx#griptape.drivers.image_generation.leonardo_image_generation_driver.LeonardoImageGenerationDriver.seed" title="seed = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.leonardo_image_generation_driver.LeonardoImageGenerationDriver.seed)"><code class="language-python">	*Optional[int]*	Optionally provide a consistent seed to generation requests, increasing consistency in output.
<a href="leonardo_image_generation_driver.mdx#griptape.drivers.image_generation.leonardo_image_generation_driver.LeonardoImageGenerationDriver.init_strength" title="init_strength = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.leonardo_image_generation_driver.LeonardoImageGenerationDriver.init_strength)" class="language-python">	*Optional[float]*	Optionally specify the strength of the initial image, [0.0, 1.0].

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
@define
class LeonardoImageGenerationDriver(BaseImageGenerationDriver):
    """Driver for the Leonardo image generation API.

    Details on Leonardo image generation parameters can be found here:
    https://docs.leonardo.ai/reference/creategeneration

    Attributes:
        model: The ID of the model to use when generating images.
        api_key: The API key to use when making requests to the Leonardo API.
        requests_session: The requests session to use when making requests to the␣
↪Leonardo API.
        api_base: The base URL of the Leonardo API.
        max_attempts: The maximum number of times to poll the Leonardo API for a␣
↪completed image.
        image_width: The width of the generated image in the range [32, 1024] and␣
↪divisible by 8.
        image_height: The height of the generated image in the range [32, 1024] and␣
↪divisible by 8.
        steps: Optionally specify the number of inference steps to run for each image␣
↪generation request, [30, 60].
        seed: Optionally provide a consistent seed to generation requests, increasing␣
↪consistency in output.
        init_strength: Optionally specify the strength of the initial image, [0.0, 1.
↪0].
    """

    api_key: str = field(kw_only=True, metadata={"serializable": True})
    requests_session: requests.Session = field(default=Factory(lambda: requests.
↪Session()), kw_only=True)
    api_base: str = "https://cloud.leonardo.ai/api/rest/v1"
    max_attempts: int = field(default=10, kw_only=True, metadata={"serializable":␣
↪True})
    image_width: int = field(default=512, kw_only=True, metadata={"serializable":␣
↪True})
    image_height: int = field(default=512, kw_only=True, metadata={"serializable":␣
↪True})
    steps: Optional[int] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    seed: Optional[int] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    init_strength: Optional[float] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    control_net: bool = field(default=False, kw_only=True, metadata={"serializable":␣
↪True})
    control_net_type: Optional[Literal["POSE", "CANNY", "DEPTH"]] = field(
        default=None,
        kw_only=True,
        metadata={"serializable": True},
    )

    def try_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact:
```

(continues on next page)

(continued from previous page)

```
        if negative_prompts is None:
            negative_prompts = []

        generation_id = self._create_generation(prompts=prompts, negative_
↪prompts=negative_prompts)
        image_url = self._get_image_url(generation_id=generation_id)
        image_data = self._download_image(url=image_url)

        return ImageArtifact(
            value=image_data,
            format="png",
            width=self.image_width,
            height=self.image_height,
            meta={
                "model": self.model,
                "prompt": ", ".join(prompts),
            },
        )

    def try_image_variation(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        if negative_prompts is None:
            negative_prompts = []

        init_image_id = self._upload_init_image(image)
        generation_id = self._create_generation(
            prompts=prompts,
            negative_prompts=negative_prompts,
            init_image_id=init_image_id,
        )
        image_url = self._get_image_url(generation_id=generation_id)
        image_data = self._download_image(url=image_url)

        return ImageArtifact(
            value=image_data,
            format="png",
            width=self.image_width,
            height=self.image_height,
            meta={
                "model": self.model,
                "prompt": ", ".join(prompts),
            },
        )

    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
```

(continues on next page)

(continued from previous page)

```
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪outpainting")

    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪inpainting")

    def _upload_init_image(self, image: ImageArtifact) -> str:
        request = {"extension": image.mime_type.split("/")[1]}

        prep_response = self._make_api_request("/init-image", request=request)
        if prep_response is None or prep_response["uploadInitImage"] is None:
            raise Exception(f"failed to prepare init image: {prep_response}")

        fields = json.loads(prep_response["uploadInitImage"]["fields"])
        pre_signed_url = prep_response["uploadInitImage"]["url"]
        init_image_id = prep_response["uploadInitImage"]["id"]

        files = {"file": image.value}
        upload_response = requests.post(pre_signed_url, data=fields, files=files)
        if not upload_response.ok:
            raise Exception(f"failed to upload init image: {upload_response.text}")

        return init_image_id

    def _create_generation(
        self,
        prompts: list[str],
        negative_prompts: list[str],
        init_image_id: Optional[str] = None,
    ) -> str:
        prompt = ", ".join(prompts)
        negative_prompt = ", ".join(negative_prompts)
        request = {
            "prompt": prompt,
            "negative_prompt": negative_prompt,
            "width": self.image_width,
            "height": self.image_height,
            "num_images": 1,
            "modelId": self.model,
        }

        if init_image_id is not None:
```

(continues on next page)

(continued from previous page)

```
            request["init_image_id"] = init_image_id

        if self.init_strength is not None:
            request["init_strength"] = self.init_strength

        if self.steps:
            request["num_inference_steps"] = self.steps

        if self.seed is not None:
            request["seed"] = self.seed

        if self.control_net:
            request["controlNet"] = self.control_net
            request["controlNetType"] = self.control_net_type

        response = self._make_api_request("/generations", request=request)
        if response is None or response["sdGenerationJob"] is None:
            raise Exception(f"failed to create generation: {response}")

        return response["sdGenerationJob"]["generationId"]

    def _make_api_request(self, endpoint: str, request: dict, method: str = "POST") ->
↪ dict:
        url = f"{self.api_base}{endpoint}"
        headers = {"Authorization": f"Bearer {self.api_key}"}

        response = self.requests_session.request(url=url, method=method, json=request,
↪ headers=headers)
        if not response.ok:
            raise Exception(f"failed to make API request: {response.text}")

        return response.json()

    def _get_image_url(self, generation_id: str) -> str:
        for attempt in range(self.max_attempts):
            response = self.requests_session.get(
                url=f"{self.api_base}/generations/{generation_id}",
                headers={"Authorization": f"Bearer {self.api_key}"},
            ).json()

            if response["generations_by_pk"]["status"] == "PENDING":
                time.sleep(attempt + 1)
                continue

            return response["generations_by_pk"]["generated_images"][0]["url"]
        raise Exception("image generation failed to complete")

    def _download_image(self, url: str) -> bytes:
        response = self.requests_session.get(url=url, headers={"Authorization": f
↪"Bearer {self.api_key}"})

        return response.content
```

- `api_base = 'https://cloud.leonardo.ai/api/rest/v1'` class-attribute instance-attribute
- `api_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `control_net = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `control_net_type = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `image_height = field(default=512, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `image_width = field(default=512, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `init_strength = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `max_attempts = field(default=10, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `requests_session = field(default=Factory(lambda: requests.Session()), kw_only=True)` class-attribute instance-attribute
- `seed = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `steps = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_create_generation(prompts, negative_prompts, init_image_id=None)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def _create_generation(
    self,
    prompts: list[str],
    negative_prompts: list[str],
    init_image_id: Optional[str] = None,
) -> str:
    prompt = ", ".join(prompts)
    negative_prompt = ", ".join(negative_prompts)
    request = {
        "prompt": prompt,
        "negative_prompt": negative_prompt,
        "width": self.image_width,
        "height": self.image_height,
        "num_images": 1,
        "modelId": self.model,
    }

    if init_image_id is not None:
        request["init_image_id"] = init_image_id

    if self.init_strength is not None:
        request["init_strength"] = self.init_strength
```

(continues on next page)

(continued from previous page)

```
    if self.steps:
        request["num_inference_steps"] = self.steps

    if self.seed is not None:
        request["seed"] = self.seed

    if self.control_net:
        request["controlNet"] = self.control_net
        request["controlNetType"] = self.control_net_type

    response = self._make_api_request("/generations", request=request)
    if response is None or response["sdGenerationJob"] is None:
        raise Exception(f"failed to create generation: {response}")

    return response["sdGenerationJob"]["generationId"]
```

_download_image(url)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def _download_image(self, url: str) -> bytes:
    response = self.requests_session.get(url=url, headers={"Authorization": f"Bearer
↪{self.api_key}"})

    return response.content
```

_get_image_url(generation_id)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def _get_image_url(self, generation_id: str) -> str:
    for attempt in range(self.max_attempts):
        response = self.requests_session.get(
            url=f"{self.api_base}/generations/{generation_id}",
            headers={"Authorization": f"Bearer {self.api_key}"},
        ).json()

        if response["generations_by_pk"]["status"] == "PENDING":
            time.sleep(attempt + 1)
            continue

        return response["generations_by_pk"]["generated_images"][0]["url"]
    raise Exception("image generation failed to complete")
```

_make_api_request(endpoint, request, method='POST')

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def _make_api_request(self, endpoint: str, request: dict, method: str = "POST") ->␣
↪dict:
```

(continues on next page)

(continued from previous page)

```
    url = f"{self.api_base}{endpoint}"
    headers = {"Authorization": f"Bearer {self.api_key}"}

    response = self.requests_session.request(url=url, method=method, json=request,␣
↪headers=headers)
    if not response.ok:
        raise Exception(f"failed to make API request: {response.text}")

    return response.json()
```

_upload_init_image(image)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def _upload_init_image(self, image: ImageArtifact) -> str:
    request = {"extension": image.mime_type.split("/")[1]}

    prep_response = self._make_api_request("/init-image", request=request)
    if prep_response is None or prep_response["uploadInitImage"] is None:
        raise Exception(f"failed to prepare init image: {prep_response}")

    fields = json.loads(prep_response["uploadInitImage"]["fields"])
    pre_signed_url = prep_response["uploadInitImage"]["url"]
    init_image_id = prep_response["uploadInitImage"]["id"]

    files = {"file": image.value}
    upload_response = requests.post(pre_signed_url, data=fields, files=files)
    if not upload_response.ok:
        raise Exception(f"failed to upload init image: {upload_response.text}")

    return init_image_id
```

try_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support inpainting
↪")
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def try_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support outpainting
↪")
```

try_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def try_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    if negative_prompts is None:
        negative_prompts = []

    init_image_id = self._upload_init_image(image)
    generation_id = self._create_generation(
        prompts=prompts,
        negative_prompts=negative_prompts,
        init_image_id=init_image_id,
    )
    image_url = self._get_image_url(generation_id=generation_id)
    image_data = self._download_image(url=image_url)

    return ImageArtifact(
        value=image_data,
        format="png",
        width=self.image_width,
        height=self.image_height,
        meta={
            "model": self.model,
            "prompt": ", ".join(prompts),
        },
    )
```

try_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/leonardo_image_generation_driver.py

```
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    if negative_prompts is None:
        negative_prompts = []
```

(continues on next page)

(continued from previous page)

```
    generation_id = self._create_generation(prompts=prompts, negative_
↪prompts=negative_prompts)
    image_url = self._get_image_url(generation_id=generation_id)
    image_data = self._download_image(url=image_url)

    return ImageArtifact(
        value=image_data,
        format="png",
        width=self.image_width,
        height=self.image_height,
        meta={
            "model": self.model,
            "prompt": ", ".join(prompts),
        },
    )
```

openai

- `__all__ = ['AzureOpenAiImageGenerationDriver', 'OpenAiImageGenerationDriver']` module-attribute

Bases: *OpenAiImageGenerationDriver* “)

Attributes

Name	Type	Description
<a href="ope-nai.mdx#grip-tape.drivers.im-age_generation.ope-nai.AzureOpe-nAiImageGenera-tionDriver.azure_de-ployment" ti-tle="azure_deployment = field(kw_only=True, default=Fac-tory(lambda self: self.model, takes_self=True), meta-data={'serializable': True}) class-attribute instance-attribute (griptape.drivers.im-age_generation.ope-nai.AzureOpe-nAiImageGenera-tionDriver.azure_de-ployment)"><code class="language-python">	*str*	An optional Azure OpenAi de-ployment id. Defaults to the model name.

continues on next page

Table 169 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.image_generation.openai.AzureOpenAiImageGenerationDriver.azure_endpoint" title="azure_endpoint = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.openai.AzureOpenAiImageGenerationDriver.azure_endpoint)"><code class="language-python">	*str*	An Azure OpenAi endpoint.
<a href="openai.mdx#griptape.drivers.image_generation.openai.AzureOpenAiImageGenerationDriver.azure_ad_token" title="azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.image_generation.openai.AzureOpenAiImageGenerationDriver.azure_ad_token)"><code class="language-python">	*Optional[str]*	An optional Azure Active Directory token.

continues on next page

Table 169 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.image_generation.openai.AzureOpenAiImageGenerationDriver.azure_ad_token_provider" title="azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.image_generation.openai.AzureOpenAiImageGenerationDriver.azure_ad_token_provider)"><code class="language-python">	*Optional[Callable[[], str]]*	An optional Azure Active Directory token provider.
<a href="openai.mdx#griptape.drivers.image_generation.openai.AzureOpenAiImageGenerationDriver.api_version" title="api_version = field(default='2024-02-01', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.openai.AzureOpenAiImageGenerationDriver.api_version)"><code class="language-python">	*str*	An Azure OpenAi API version.

continues on next page

Table 169 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.image_generation.openai.AzureOpenAiImageGenerationDriver.client" title="client() (griptape.drivers.image_generation.openai.AzureOpenAiImageGenerationDriver.client)"><code class="language-python">	*AzureOpenAI*	An *openai.AzureOpenAI* client.

Source Code in griptape/drivers/image_generation/azure_openai_image_generation_driver.py

```
@define
class AzureOpenAiImageGenerationDriver(OpenAiImageGenerationDriver):
    """Driver for Azure-hosted OpenAI image generation API.

    Attributes:
        azure_deployment: An optional Azure OpenAi deployment id. Defaults to the
↪model name.
        azure_endpoint: An Azure OpenAi endpoint.
        azure_ad_token: An optional Azure Active Directory token.
        azure_ad_token_provider: An optional Azure Active Directory token provider.
        api_version: An Azure OpenAi API version.
        client: An `openai.AzureOpenAI` client.
    """

    azure_deployment: str = field(
        kw_only=True,
        default=Factory(lambda self: self.model, takes_self=True),
        metadata={"serializable": True},
    )
    azure_endpoint: str = field(kw_only=True, metadata={"serializable": True})
    azure_ad_token: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": False})
    azure_ad_token_provider: Optional[Callable[[], str]] = field(
        kw_only=True,
        default=None,
        metadata={"serializable": False},
    )
    api_version: str = field(default="2024-02-01", kw_only=True, metadata={
↪"serializable": True})
    _client: Optional[openai.AzureOpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )
```

(continues on next page)

(continued from previous page)

```
    @lazy_property()
    def client(self) -> openai.AzureOpenAI:
        return openai.AzureOpenAI(
            organization=self.organization,
            api_key=self.api_key,
            api_version=self.api_version,
            azure_endpoint=self.azure_endpoint,
            azure_deployment=self.azure_deployment,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_version = field(default='2024-02-01', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True})` class-attribute instance-attribute
- `azure_endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/image_generation/azure_openai_image_generation_driver.py

```
@lazy_property()
def client(self) -> openai.AzureOpenAI:
    return openai.AzureOpenAI(
        organization=self.organization,
        api_key=self.api_key,
        api_version=self.api_version,
        azure_endpoint=self.azure_endpoint,
        azure_deployment=self.azure_deployment,
        azure_ad_token=self.azure_ad_token,
        azure_ad_token_provider=self.azure_ad_token_provider,
    )
```

OpenAiImageGenerationDriver

Bases: *BaseImageGenerationDriver* “)

Attributes

Name	Type	Description
<a href="base_image_generation_driver.mdx#griptape.drivers.image_generation.base_image_generation_driver.BaseImageGenerationDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.model)"><cod class="language-python">	*str*	OpenAI model, for example 'dall-e-2' or 'dall-e-3'.
<a href="openai.mdx#griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.api_type" title="api_type = field(default=openai.api_type, kw_only=True) class-attribute instance-attribute (griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.api_type)"><c class="language-python">	*Optional[str]*	OpenAI API type, for example 'open_ai' or 'azure'.

continues on next page

Table 170 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.api_version" title="api_version = field(default=openai.api_version, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.api_version)"><code class="language-python">	*Optional[str]*	API version.
<a href="openai.mdx#griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.base_url" title="base_url = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.base_url)"><c class="language-python">	*Optional[str]*	API URL.

continues on next page

Table 170 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.api_key" title="api_key = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.api_key)"><co class="language-python">	*Optional[str]*	OpenAI API key.
<a href="openai.mdx#griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.organization" title="organization = field(default=openai.organization, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.organization)"><code class="language-python">	*Optional[str]*	OpenAI organization ID.

continues on next page

Table 170 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.style" title="style = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['dall-e-3']}) class-attribute instance-attribute (griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.style)"><code class="language-python">	*Optional[Literal['vivid', 'natural']]*	Optional and only supported for dall-e-3, can be either 'vivid' or 'natural'.
<a href="openai.mdx#griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.quality" title="quality = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.quality)"><code class="language-python">	*Optional[Literal['standard', 'hd', 'low', 'medium', 'high', 'auto']]*	Optional and only supported for dall-e-3. Accepts 'standard', 'hd'.

continues on next page

Table 170 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.image_size" title="image_size = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.image_size)"><code class="language-python">	*Optional[Literal['256x256', '512x512', '1024x1024', '1024x1792', '1792x1024']]*	Size of the generated image. Must be one of the following, depending on the requested model: dall-e-2: [256x256, 512x512, 1024x1024] dall-e-3: [1024x1024, 1024x1792, 1792x1024] gpt-image-1: [1024x1024, 1536x1024, 1024x1536, auto]
<a href="openai.mdx#griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.response_format" title="response_format = field(default='b64_json', kw_only=True, metadata={'serializable': True, 'model_denylist': ['gpt-image-1']}) class-attribute instance-attribute (griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.response_format)"><code class="language-python">	*Literal['b64_json']*	The response format. Currently only supports 'b64_json' which will return a base64 encoded image in a JSON object.

continues on next page

Table 170 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.background" title="background = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']}) class-attribute instance-attribute (griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.background)"><code class="language-python">	*Optional[Literal['transparent', 'opaque', 'auto']]*	Optional and only supported for gpt-image-1. Can be either 'transparent', 'opaque', or 'auto'.
<a href="openai.mdx#griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.moderation" title="moderation = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']}) class-attribute instance-attribute (griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.moderation)"><code class="language-python">	*Optional[Literal['low', 'auto']]*	Optional and only supported for gpt-image-1. Can be either 'low' or 'auto'.

continues on next page

Table 170 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.output_compression" title="output_compression = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']}) class-attribute instance-attribute (griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.output_compression)"><code class="language-python">	*Optional[int]*	Optional and only supported for gpt-image-1. Can be an integer between 0 and 100.
<a href="openai.mdx#griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.output_format" title="output_format = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']}) class-attribute instance-attribute (griptape.drivers.image_generation.openai.OpenAiImageGenerationDriver.output_format)"><code class="language-python">	*Optional[Literal['png', 'jpeg']]*	Optional and only supported for gpt-image-1. Can be either 'png' or 'jpeg'.

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
@define
class OpenAiImageGenerationDriver(BaseImageGenerationDriver):
    """Driver for the OpenAI image generation API.
```

(continues on next page)

(continued from previous page)

```
    Attributes:
        model: OpenAI model, for example dall-e-2 or dall-e-3.
        api_type: OpenAI API type, for example open_ai or azure.
        api_version: API version.
        base_url: API URL.
        api_key: OpenAI API key.
        organization: OpenAI organization ID.
        style: Optional and only supported for dall-e-3, can be either vivid or␣
↪natural.
        quality: Optional and only supported for dall-e-3. Accepts standard, hd.
        image_size: Size of the generated image. Must be one of the following,␣
↪depending on the requested model:
            dall-e-2: [256x256, 512x512, 1024x1024]
            dall-e-3: [1024x1024, 1024x1792, 1792x1024]
            gpt-image-1: [1024x1024, 1536x1024, 1024x1536, auto]
        response_format: The response format. Currently only supports b64_json which␣
↪will return
            a base64 encoded image in a JSON object.
        background: Optional and only supported for gpt-image-1. Can be either␣
↪transparent, opaque, or auto.
        moderation: Optional and only supported for gpt-image-1. Can be either low or␣
↪auto.
        output_compression: Optional and only supported for gpt-image-1. Can be an␣
↪integer between 0 and 100.
        output_format: Optional and only supported for gpt-image-1. Can be either png␣
↪or jpeg.
    """

    api_type: Optional[str] = field(default=openai.api_type, kw_only=True)
    api_version: Optional[str] = field(default=openai.api_version, kw_only=True,␣
↪metadata={"serializable": True})
    base_url: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    organization: Optional[str] = field(default=openai.organization, kw_only=True,␣
↪metadata={"serializable": True})
    style: Optional[Literal["vivid", "natural"]] = field(
        default=None, kw_only=True, metadata={"serializable": True, "model_allowlist
↪": ["dall-e-3"]}
    )
    quality: Optional[Literal["standard", "hd", "low", "medium", "high", "auto"]] =␣
↪field(
        default=None,
        kw_only=True,
        metadata={"serializable": True},
    )
    image_size: Optional[Literal["256x256", "512x512", "1024x1024", "1024x1792",
↪"1792x1024"]] = field(
        default=None,
        kw_only=True,
```

(continues on next page)

(continued from previous page)

```
        metadata={"serializable": True},
    )
    response_format: Literal["b64_json"] = field(
        default="b64_json",
        kw_only=True,
        metadata={"serializable": True, "model_denylist": ["gpt-image-1"]},
    )
    background: Optional[Literal["transparent", "opaque", "auto"]] = field(
        default=None,
        kw_only=True,
        metadata={"serializable": True, "model_allowlist": ["gpt-image-1"]},
    )
    moderation: Optional[Literal["low", "auto"]] = field(
        default=None,
        kw_only=True,
        metadata={"serializable": True, "model_allowlist": ["gpt-image-1"]},
    )
    output_compression: Optional[int] = field(
        default=None,
        kw_only=True,
        metadata={"serializable": True, "model_allowlist": ["gpt-image-1"]},
    )
    output_format: Optional[Literal["png", "jpeg"]] = field(
        default=None,
        kw_only=True,
        metadata={"serializable": True, "model_allowlist": ["gpt-image-1"]},
    )
    _client: Optional[openai.OpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @image_size.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_image_size(self, attribute: str, value: str | None) -> None:
        """Validates the image size based on the model.

        Must be one of `1024x1024`, `1536x1024` (landscape), `1024x1536` (portrait),␣
↪or `auto` (default value) for
        `gpt-image-1`, one of `256x256`, `512x512`, or `1024x1024` for `dall-e-2`, and
        one of `1024x1024`, `1792x1024`, or `1024x1792` for `dall-e-3`.

        """
        if value is None:
            return

        if self.model.startswith("gpt-image"):
            allowed_sizes = ("1024x1024", "1536x1024", "1024x1536", "auto")
        elif self.model == "dall-e-2":
            allowed_sizes = ("256x256", "512x512", "1024x1024")
        elif self.model == "dall-e-3":
            allowed_sizes = ("1024x1024", "1792x1024", "1024x1792")
        else:
```

(continues on next page)

(continued from previous page)

```
            raise NotImplementedError(f"Image size validation not implemented for␣
↪model {self.model}")

        if value is not None and value not in allowed_sizes:
            raise ValueError(f"Image size, {value}, must be one of the following:
↪{allowed_sizes}")

    @lazy_property()
    def client(self) -> openai.OpenAI:
        return openai.OpenAI(api_key=self.api_key, base_url=self.base_url,␣
↪organization=self.organization)

    def try_text_to_image(self, prompts: list[str], negative_prompts:␣
↪Optional[list[str]] = None) -> ImageArtifact:
        prompt = ", ".join(prompts)

        response = self.client.images.generate(
            model=self.model,
            prompt=prompt,
            n=1,
            **self._build_model_params(
                {
                    "size": "image_size",
                    "quality": "quality",
                    "style": "style",
                    "response_format": "response_format",
                    "background": "background",
                    "moderation": "moderation",
                    "output_compression": "output_compression",
                    "output_format": "output_format",
                }
            ),
        )

        return self._parse_image_response(response, prompt)

    def try_image_variation(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        """Creates a variation of an image.

        Only supported by for dall-e-2. Requires image size to be one of the␣
↪following:
            [256x256, 512x512, 1024x1024]
        """
        if self.model != "dall-e-2":
            raise NotImplementedError("Image variation only supports dall-e-2")
        response = self.client.images.create_variation(
            image=image.value,
```

(continues on next page)

(continued from previous page)

```
            n=1,
            response_format=self.response_format,
            size=self.image_size,  # pyright: ignore[reportArgumentType]
        )

        return self._parse_image_response(response, "")

    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        prompt = ", ".join(prompts)
        response = self.client.images.edit(
            prompt=prompt,
            image=image.value,
            mask=mask.value,
            **self._build_model_params(
                {
                    "size": "image_size",
                    "response_format": "response_format",
                }
            ),
        )

        return self._parse_image_response(response, prompt)

    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪outpainting")

    def _parse_image_response(self, response: ImagesResponse, prompt: str) ->␣
↪ImageArtifact:
        from griptape.loaders.image_loader import ImageLoader

        if response.data is None or response.data[0] is None or response.data[0].b64_
↪json is None:
            raise Exception("Failed to generate image")

        image_data = base64.b64decode(response.data[0].b64_json)

        image_artifact = ImageLoader().parse(image_data)

        image_artifact.meta["prompt"] = prompt
```

(continues on next page)

(continued from previous page)

```
        image_artifact.meta["model"] = self.model

        return image_artifact

    def _build_model_params(self, values: dict) -> dict:
        """Builds parameters while considering field metadata and None values.

        Args:
            values: A dictionary mapping parameter names to field names.

        Field will be added to the params dictionary if all conditions are met:
            - The field value is not None
            - The model_allowlist is None or the model is in the allowlist
            - The model_denylist is None or the model is not in the denylist
        """
        params = {}

        fields = fields_dict(self.__class__)
        for param_name, field_name in values.items():
            metadata = fields[field_name].metadata
            model_allowlist = metadata.get("model_allowlist")
            model_denylist = metadata.get("model_denylist")

            field_value = getattr(self, field_name, None)

            allowlist_condition = model_allowlist is None or self.model in model_
↪allowlist
            denylist_condition = model_denylist is None or self.model not in model_
↪denylist

            if field_value is not None and allowlist_condition and denylist_condition:
                params[param_name] = field_value
        return params
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `api_type = field(default=openai.api_type, kw_only=True)` class-attribute instance-attribute
- `api_version = field(default=openai.api_version, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `background = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']})` class-attribute instance-attribute
- `base_url = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `image_size = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `moderation = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']})` class-attribute instance-attribute

- `organization = field(default=openai.organization, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output_compression = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']})` class-attribute instance-attribute
- `output_format = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']})` class-attribute instance-attribute
- `quality = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `response_format = field(default='b64_json', kw_only=True, metadata={'serializable': True, 'model_denylist': ['gpt-image-1']})` class-attribute instance-attribute
- `style = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['dall-e-3']})` class-attribute instance-attribute

_build_model_params(values)

Builds parameters while considering field metadata and None values.

Parameters

Name	Type	Description	Default
values	*dict*	A dictionary mapping parameter names to field names. 	*required*

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
def _build_model_params(self, values: dict) -> dict:
    """Builds parameters while considering field metadata and None values.

    Args:
        values: A dictionary mapping parameter names to field names.

    Field will be added to the params dictionary if all conditions are met:
        - The field value is not None
        - The model_allowlist is None or the model is in the allowlist
        - The model_denylist is None or the model is not in the denylist
    """
    params = {}

    fields = fields_dict(self.__class__)
    for param_name, field_name in values.items():
        metadata = fields[field_name].metadata
        model_allowlist = metadata.get("model_allowlist")
        model_denylist = metadata.get("model_denylist")

        field_value = getattr(self, field_name, None)

        allowlist_condition = model_allowlist is None or self.model in model_allowlist
```

(continues on next page)

(continued from previous page)

```
        denylist_condition = model_denylist is None or self.model not in model_
↪denylist

        if field_value is not None and allowlist_condition and denylist_condition:
            params[param_name] = field_value
    return params
```

_parse_image_response(response, prompt)

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
def _parse_image_response(self, response: ImagesResponse, prompt: str) ->␣
↪ImageArtifact:
    from griptape.loaders.image_loader import ImageLoader

    if response.data is None or response.data[0] is None or response.data[0].b64_json␣
↪is None:
        raise Exception("Failed to generate image")

    image_data = base64.b64decode(response.data[0].b64_json)

    image_artifact = ImageLoader().parse(image_data)

    image_artifact.meta["prompt"] = prompt
    image_artifact.meta["model"] = self.model

    return image_artifact
```

client()

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
@lazy_property()
def client(self) -> openai.OpenAI:
    return openai.OpenAI(api_key=self.api_key, base_url=self.base_url,␣
↪organization=self.organization)
```

try_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    prompt = ", ".join(prompts)
    response = self.client.images.edit(
        prompt=prompt,
```

(continues on next page)

(continued from previous page)

```
        image=image.value,
        mask=mask.value,
        **self._build_model_params(
            {
                "size": "image_size",
                "response_format": "response_format",
            }
        ),
    )

    return self._parse_image_response(response, prompt)
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
def try_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support outpainting
↪")
```

try_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
def try_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    """Creates a variation of an image.

    Only supported by for dall-e-2. Requires image size to be one of the following:
        [256x256, 512x512, 1024x1024]
    """
    if self.model != "dall-e-2":
        raise NotImplementedError("Image variation only supports dall-e-2")
    response = self.client.images.create_variation(
        image=image.value,
        n=1,
        response_format=self.response_format,
        size=self.image_size,  # pyright: ignore[reportArgumentType]
    )

    return self._parse_image_response(response, "")
```

try_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    prompt = ", ".join(prompts)

    response = self.client.images.generate(
        model=self.model,
        prompt=prompt,
        n=1,
        **self._build_model_params(
            {
                "size": "image_size",
                "quality": "quality",
                "style": "style",
                "response_format": "response_format",
                "background": "background",
                "moderation": "moderation",
                "output_compression": "output_compression",
                "output_format": "output_format",
            }
        ),
    )

    return self._parse_image_response(response, prompt)
```

validate_image_size(attribute, value)

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
@image_size.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_image_size(self, attribute: str, value: str | None) -> None:
    """Validates the image size based on the model.

    Must be one of `1024x1024`, `1536x1024` (landscape), `1024x1536` (portrait), or␣
↪`auto` (default value) for
    `gpt-image-1`, one of `256x256`, `512x512`, or `1024x1024` for `dall-e-2`, and
    one of `1024x1024`, `1792x1024`, or `1024x1792` for `dall-e-3`.

    """
    if value is None:
        return

    if self.model.startswith("gpt-image"):
        allowed_sizes = ("1024x1024", "1536x1024", "1024x1536", "auto")
    elif self.model == "dall-e-2":
        allowed_sizes = ("256x256", "512x512", "1024x1024")
    elif self.model == "dall-e-3":
        allowed_sizes = ("1024x1024", "1792x1024", "1024x1792")
    else:
```

(continues on next page)

(continued from previous page)

```
        raise NotImplementedError(f"Image size validation not implemented for model
↪{self.model}")

    if value is not None and value not in allowed_sizes:
        raise ValueError(f"Image size, {value}, must be one of the following:
↪{allowed_sizes}")
```

openai_image_generation_driver

Bases: *BaseImageGenerationDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.image_generation.BaseImageGenerationDriver.model" title="model = field(kw_only=True, metadata={'serializable': True})">class-attribute instance-attribute (griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.model)"><cod class="language-python">	*str*	OpenAI model, for example 'dall-e-2' or 'dall-e-3'.

continues on next page

Table 172 – continued from previous page

Name	Type	Description
<a href="openai_image_generation_driver.mdx#griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.api_type" title="api_type = field(default=openai.api_type, kw_only=True) class-attribute instance-attribute (griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.api_type)"><c class="language-python">	*Optional[str]*	OpenAI API type, for example 'open_ai' or 'azure'.
<a href="openai_image_generation_driver.mdx#griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.api_version" title="api_version = field(default=openai.api_version, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.api_version)"><code class="language-python">	*Optional[str]*	API version.

continues on next page

Table 172 – continued from previous page

Name	Type	Description
<a href="openai_image_generation_driver.mdx#griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.base_url" title="base_url = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.base_url)"><cc class="language-python">	*Optional[str]*	API URL.
<a href="openai_image_generation_driver.mdx#griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.api_key" title="api_key = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.api_key)"><cc class="language-python">	*Optional[str]*	OpenAI API key.

continues on next page

Table 172 – continued from previous page

Name	Type	Description
<a href="openai_image_generation_driver.mdx#griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.organization" title="organization = field(default=openai.organization, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.organization)"><code class="language-python">	*Optional[str]*	OpenAI organization ID.
<a href="openai_image_generation_driver.mdx#griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.style" title="style = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['dall-e-3']}) class-attribute instance-attribute (griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.style)"><code class="language-python">	*Optional[Literal['vivid', 'natural']]*	Optional and only supported for dall-e-3, can be either 'vivid' or 'natural'.

continues on next page

Table 172 – continued from previous page

Name	Type	Description
<a href="openai_image_generation_driver.mdx#griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.quality" title="quality = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.quality)"><code class="language-python">	*Optional[Literal['standard', 'hd', 'low', 'medium', 'high', 'auto']]*	Optional and only supported for dall-e-3. Accepts 'standard', 'hd'.
<a href="openai_image_generation_driver.mdx#griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.image_size" title="image_size = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.image_size)"><code class="language-python">	*Optional[Literal['256x256', '512x512', '1024x1024', '1024x1792', '1792x1024']]*	Size of the generated image. Must be one of the following, depending on the requested model: dall-e-2: [256x256, 512x512, 1024x1024] dall-e-3: [1024x1024, 1024x1792, 1792x1024] gpt-image-1: [1024x1024, 1536x1024, 1024x1536, auto]

continues on next page

Table 172 – continued from previous page

Name	Type	Description
<a href="openai_image_generation_driver.mdx#griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.response_format" title="response_format = field(default='b64_json', kw_only=True, metadata={'serializable': True, 'model_denylist': ['gpt-image-1']}) class-attribute instance-attribute (griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.response_format)"><code class="language-python">	*Literal['b64_json']*	The response format. Currently only supports 'b64_json' which will return a base64 encoded image in a JSON object.

continues on next page

Table 172 – continued from previous page

Name	Type	Description
<a href="openai_image_generation_driver.mdx#griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.background" title="background = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']}) class-attribute instance-attribute (griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.background)"><code class="language-python">	*Optional[Literal['transparent', 'opaque', 'auto']]*	Optional and only supported for gpt-image-1. Can be either 'transparent', 'opaque', or 'auto'.
<a href="openai_image_generation_driver.mdx#griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.moderation" title="moderation = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']}) class-attribute instance-attribute (griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.moderation)"><code class="language-python">	*Optional[Literal['low', 'auto']]*	Optional and only supported for gpt-image-1. Can be either 'low' or 'auto'.

continues on next page

Table 172 – continued from previous page

Name	Type	Description
<a href="openai_image_generation_driver.mdx#griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.output_compression" title="output_compression = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']}) class-attribute instance-attribute (griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.output_compression)"><code class="language-python">	*Optional[int]*	Optional and only supported for gpt-image-1. Can be an integer between 0 and 100.

continues on next page

Table 172 – continued from previous page

Name	Type	Description
<a href="openai_image_generation_driver.mdx#griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.output_format" title="output_format = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']}) class-attribute instance-attribute (griptape.drivers.image_generation.openai_image_generation_driver.OpenAiImageGenerationDriver.output_format)"><code class="language-python">	*Optional[Literal['png', 'jpeg']]*	Optional and only supported for gpt-image-1. Can be either 'png' or 'jpeg'.

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
@define
class OpenAiImageGenerationDriver(BaseImageGenerationDriver):
    """Driver for the OpenAI image generation API.

    Attributes:
        model: OpenAI model, for example dall-e-2 or dall-e-3.
        api_type: OpenAI API type, for example open_ai or azure.
        api_version: API version.
        base_url: API URL.
        api_key: OpenAI API key.
        organization: OpenAI organization ID.
        style: Optional and only supported for dall-e-3, can be either vivid or␣
↪natural.
        quality: Optional and only supported for dall-e-3. Accepts standard, hd.
        image_size: Size of the generated image. Must be one of the following,␣
↪depending on the requested model:
            dall-e-2: [256x256, 512x512, 1024x1024]
            dall-e-3: [1024x1024, 1024x1792, 1792x1024]
            gpt-image-1: [1024x1024, 1536x1024, 1024x1536, auto]
        response_format: The response format. Currently only supports b64_json which␣
↪will return
```

(continues on next page)

(continued from previous page)

```
            a base64 encoded image in a JSON object.
        background: Optional and only supported for gpt-image-1. Can be either␣
↪transparent, opaque, or auto.
        moderation: Optional and only supported for gpt-image-1. Can be either low or␣
↪auto.
        output_compression: Optional and only supported for gpt-image-1. Can be an␣
↪integer between 0 and 100.
        output_format: Optional and only supported for gpt-image-1. Can be either png␣
↪or jpeg.
    """

    api_type: Optional[str] = field(default=openai.api_type, kw_only=True)
    api_version: Optional[str] = field(default=openai.api_version, kw_only=True,␣
↪metadata={"serializable": True})
    base_url: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    organization: Optional[str] = field(default=openai.organization, kw_only=True,␣
↪metadata={"serializable": True})
    style: Optional[Literal["vivid", "natural"]] = field(
        default=None, kw_only=True, metadata={"serializable": True, "model_allowlist
↪": ["dall-e-3"]}
    )
    quality: Optional[Literal["standard", "hd", "low", "medium", "high", "auto"]] =␣
↪field(
        default=None,
        kw_only=True,
        metadata={"serializable": True},
    )
    image_size: Optional[Literal["256x256", "512x512", "1024x1024", "1024x1792",
↪"1792x1024"]] = field(
        default=None,
        kw_only=True,
        metadata={"serializable": True},
    )
    response_format: Literal["b64_json"] = field(
        default="b64_json",
        kw_only=True,
        metadata={"serializable": True, "model_denylist": ["gpt-image-1"]},
    )
    background: Optional[Literal["transparent", "opaque", "auto"]] = field(
        default=None,
        kw_only=True,
        metadata={"serializable": True, "model_allowlist": ["gpt-image-1"]},
    )
    moderation: Optional[Literal["low", "auto"]] = field(
        default=None,
        kw_only=True,
        metadata={"serializable": True, "model_allowlist": ["gpt-image-1"]},
    )
    output_compression: Optional[int] = field(
```

(continues on next page)

(continued from previous page)

```
        default=None,
        kw_only=True,
        metadata={"serializable": True, "model_allowlist": ["gpt-image-1"]},
    )
    output_format: Optional[Literal["png", "jpeg"]] = field(
        default=None,
        kw_only=True,
        metadata={"serializable": True, "model_allowlist": ["gpt-image-1"]},
    )
    _client: Optional[openai.OpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @image_size.validator  # pyright: ignore[reportAttributeAccessIssue,
↪reportOptionalMemberAccess]
    def validate_image_size(self, attribute: str, value: str | None) -> None:
        """Validates the image size based on the model.

        Must be one of `1024x1024`, `1536x1024` (landscape), `1024x1536` (portrait),
↪or `auto` (default value) for
        `gpt-image-1`, one of `256x256`, `512x512`, or `1024x1024` for `dall-e-2`, and
        one of `1024x1024`, `1792x1024`, or `1024x1792` for `dall-e-3`.

        """
        if value is None:
            return

        if self.model.startswith("gpt-image"):
            allowed_sizes = ("1024x1024", "1536x1024", "1024x1536", "auto")
        elif self.model == "dall-e-2":
            allowed_sizes = ("256x256", "512x512", "1024x1024")
        elif self.model == "dall-e-3":
            allowed_sizes = ("1024x1024", "1792x1024", "1024x1792")
        else:
            raise NotImplementedError(f"Image size validation not implemented for
↪model {self.model}")

        if value is not None and value not in allowed_sizes:
            raise ValueError(f"Image size, {value}, must be one of the following:
↪{allowed_sizes}")

    @lazy_property()
    def client(self) -> openai.OpenAI:
        return openai.OpenAI(api_key=self.api_key, base_url=self.base_url,
↪organization=self.organization)

    def try_text_to_image(self, prompts: list[str], negative_prompts:
↪Optional[list[str]] = None) -> ImageArtifact:
        prompt = ", ".join(prompts)

        response = self.client.images.generate(
            model=self.model,
```

(continues on next page)

(continued from previous page)

```
            prompt=prompt,
            n=1,
            **self._build_model_params(
                {
                    "size": "image_size",
                    "quality": "quality",
                    "style": "style",
                    "response_format": "response_format",
                    "background": "background",
                    "moderation": "moderation",
                    "output_compression": "output_compression",
                    "output_format": "output_format",
                }
            ),
        )

        return self._parse_image_response(response, prompt)

    def try_image_variation(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        """Creates a variation of an image.

        Only supported by for dall-e-2. Requires image size to be one of the␣
↪following:
            [256x256, 512x512, 1024x1024]
        """
        if self.model != "dall-e-2":
            raise NotImplementedError("Image variation only supports dall-e-2")
        response = self.client.images.create_variation(
            image=image.value,
            n=1,
            response_format=self.response_format,
            size=self.image_size,  # pyright: ignore[reportArgumentType]
        )

        return self._parse_image_response(response, "")

    def try_image_inpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        prompt = ", ".join(prompts)
        response = self.client.images.edit(
            prompt=prompt,
            image=image.value,
```

(continues on next page)

(continued from previous page)

```
            mask=mask.value,
            **self._build_model_params(
                {
                    "size": "image_size",
                    "response_format": "response_format",
                }
            ),
        )

        return self._parse_image_response(response, prompt)

    def try_image_outpainting(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
    ) -> ImageArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪outpainting")

    def _parse_image_response(self, response: ImagesResponse, prompt: str) ->␣
↪ImageArtifact:
        from griptape.loaders.image_loader import ImageLoader

        if response.data is None or response.data[0] is None or response.data[0].b64_
↪json is None:
            raise Exception("Failed to generate image")

        image_data = base64.b64decode(response.data[0].b64_json)

        image_artifact = ImageLoader().parse(image_data)

        image_artifact.meta["prompt"] = prompt
        image_artifact.meta["model"] = self.model

        return image_artifact

    def _build_model_params(self, values: dict) -> dict:
        """Builds parameters while considering field metadata and None values.

        Args:
            values: A dictionary mapping parameter names to field names.

        Field will be added to the params dictionary if all conditions are met:
            - The field value is not None
            - The model_allowlist is None or the model is in the allowlist
            - The model_denylist is None or the model is not in the denylist
        """
        params = {}

        fields = fields_dict(self.__class__)
```

(continues on next page)

(continued from previous page)

```
        for param_name, field_name in values.items():
            metadata = fields[field_name].metadata
            model_allowlist = metadata.get("model_allowlist")
            model_denylist = metadata.get("model_denylist")

            field_value = getattr(self, field_name, None)

            allowlist_condition = model_allowlist is None or self.model in model_
↪allowlist
            denylist_condition = model_denylist is None or self.model not in model_
↪denylist

            if field_value is not None and allowlist_condition and denylist_condition:
                params[param_name] = field_value
        return params
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `api_type = field(default=openai.api_type, kw_only=True)` class-attribute instance-attribute
- `api_version = field(default=openai.api_version, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `background = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']})` class-attribute instance-attribute
- `base_url = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `image_size = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `moderation = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']})` class-attribute instance-attribute
- `organization = field(default=openai.organization, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output_compression = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']})` class-attribute instance-attribute
- `output_format = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['gpt-image-1']})` class-attribute instance-attribute
- `quality = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `response_format = field(default='b64_json', kw_only=True, metadata={'serializable': True, 'model_denylist': ['gpt-image-1']})` class-attribute instance-attribute
- `style = field(default=None, kw_only=True, metadata={'serializable': True, 'model_allowlist': ['dall-e-3']})` class-attribute instance-attribute

_build_model_params(values)

Builds parameters while considering field metadata and None values.

Parameters

Name	Type	Description	Default
values	*dict*	A dictionary mapping parameter names to field names. 	*required*

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
def _build_model_params(self, values: dict) -> dict:
    """Builds parameters while considering field metadata and None values.

    Args:
        values: A dictionary mapping parameter names to field names.

    Field will be added to the params dictionary if all conditions are met:
        - The field value is not None
        - The model_allowlist is None or the model is in the allowlist
        - The model_denylist is None or the model is not in the denylist
    """
    params = {}

    fields = fields_dict(self.__class__)
    for param_name, field_name in values.items():
        metadata = fields[field_name].metadata
        model_allowlist = metadata.get("model_allowlist")
        model_denylist = metadata.get("model_denylist")

        field_value = getattr(self, field_name, None)

        allowlist_condition = model_allowlist is None or self.model in model_allowlist
        denylist_condition = model_denylist is None or self.model not in model_
↪denylist

        if field_value is not None and allowlist_condition and denylist_condition:
            params[param_name] = field_value
    return params
```

_parse_image_response(response, prompt)

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
def _parse_image_response(self, response: ImagesResponse, prompt: str) ->␣
↪ImageArtifact:
    from griptape.loaders.image_loader import ImageLoader
```

(continues on next page)

(continued from previous page)

```
    if response.data is None or response.data[0] is None or response.data[0].b64_json␣
↪is None:
        raise Exception("Failed to generate image")

    image_data = base64.b64decode(response.data[0].b64_json)

    image_artifact = ImageLoader().parse(image_data)

    image_artifact.meta["prompt"] = prompt
    image_artifact.meta["model"] = self.model

    return image_artifact
```

client()

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
@lazy_property()
def client(self) -> openai.OpenAI:
    return openai.OpenAI(api_key=self.api_key, base_url=self.base_url,␣
↪organization=self.organization)
```

try_image_inpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
def try_image_inpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    prompt = ", ".join(prompts)
    response = self.client.images.edit(
        prompt=prompt,
        image=image.value,
        mask=mask.value,
        **self._build_model_params(
            {
                "size": "image_size",
                "response_format": "response_format",
            }
        ),
    )

    return self._parse_image_response(response, prompt)
```

try_image_outpainting(prompts, image, mask, negative_prompts=None)

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
def try_image_outpainting(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support outpainting
↪")
```

try_image_variation(prompts, image, negative_prompts=None)

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
def try_image_variation(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
) -> ImageArtifact:
    """Creates a variation of an image.

    Only supported by for dall-e-2. Requires image size to be one of the following:
        [256x256, 512x512, 1024x1024]
    """
    if self.model != "dall-e-2":
        raise NotImplementedError("Image variation only supports dall-e-2")
    response = self.client.images.create_variation(
        image=image.value,
        n=1,
        response_format=self.response_format,
        size=self.image_size,  # pyright: ignore[reportArgumentType]
    )

    return self._parse_image_response(response, "")
```

try_text_to_image(prompts, negative_prompts=None)

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
def try_text_to_image(self, prompts: list[str], negative_prompts: Optional[list[str]]␣
↪= None) -> ImageArtifact:
    prompt = ", ".join(prompts)

    response = self.client.images.generate(
        model=self.model,
        prompt=prompt,
        n=1,
        **self._build_model_params(
```

(continues on next page)

(continued from previous page)

```
            {
                "size": "image_size",
                "quality": "quality",
                "style": "style",
                "response_format": "response_format",
                "background": "background",
                "moderation": "moderation",
                "output_compression": "output_compression",
                "output_format": "output_format",
            }
        ),
    )

    return self._parse_image_response(response, prompt)
```

validate_image_size(attribute, value)

Source Code in griptape/drivers/image_generation/openai_image_generation_driver.py

```
@image_size.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_image_size(self, attribute: str, value: str | None) -> None:
    """Validates the image size based on the model.

    Must be one of `1024x1024`, `1536x1024` (landscape), `1024x1536` (portrait), or␣
↪`auto` (default value) for
    `gpt-image-1`, one of `256x256`, `512x512`, or `1024x1024` for `dall-e-2`, and
    one of `1024x1024`, `1792x1024`, or `1024x1792` for `dall-e-3`.

    """
    if value is None:
        return

    if self.model.startswith("gpt-image"):
        allowed_sizes = ("1024x1024", "1536x1024", "1024x1536", "auto")
    elif self.model == "dall-e-2":
        allowed_sizes = ("256x256", "512x512", "1024x1024")
    elif self.model == "dall-e-3":
        allowed_sizes = ("1024x1024", "1792x1024", "1024x1792")
    else:
        raise NotImplementedError(f"Image size validation not implemented for model
↪{self.model}")

    if value is not None and value not in allowed_sizes:
        raise ValueError(f"Image size, {value}, must be one of the following:
↪{allowed_sizes}")
```

image_generation_model

- `__all__ = ['BaseImageGenerationModelDriver']` module-attribute

Bases: *SerializableMixin* “) , ABC

Source Code in griptape/drivers/image_generation_model/base_image_generation_model_driver.py

```python
@define
class BaseImageGenerationModelDriver(SerializableMixin, ABC):
    @abstractmethod
    def get_generated_image(self, response: dict) -> bytes: ...

    @abstractmethod
    def text_to_image_request_parameters(
        self,
        prompts: list[str],
        image_width: int,
        image_height: int,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict[str, Any]: ...

    @abstractmethod
    def image_variation_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict[str, Any]: ...

    @abstractmethod
    def image_inpainting_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict[str, Any]: ...

    @abstractmethod
    def image_outpainting_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict[str, Any]: ...
```

get_generated_image(response)abstractmethod

Source Code in griptape/drivers/image_generation_model/base_image_generation_model_driver.py

```
@abstractmethod
def get_generated_image(self, response: dict) -> bytes: ...
```

image_inpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)abstractmethod

Source Code in griptape/drivers/image_generation_model/base_image_generation_model_driver.py

```
@abstractmethod
def image_inpainting_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict[str, Any]: ...
```

image_outpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)abstractmethod

Source Code in griptape/drivers/image_generation_model/base_image_generation_model_driver.py

```
@abstractmethod
def image_outpainting_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict[str, Any]: ...
```

image_variation_request_parameters(prompts, image, negative_prompts=None, seed=None)abstractmethod

Source Code in griptape/drivers/image_generation_model/base_image_generation_model_driver.py

```
@abstractmethod
def image_variation_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict[str, Any]: ...
```

text_to_image_request_parameters(prompts, image_width, image_height, negative_prompts=None, seed=None)abstractmethod

Source Code in griptape/drivers/image_generation_model/base_image_generation_model_driver.py

```
@abstractmethod
def text_to_image_request_parameters(
    self,
    prompts: list[str],
    image_width: int,
    image_height: int,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict[str, Any]: ...
```

base_image_generation_model_driver

Bases: *SerializableMixin* “) , ABC

Source Code in griptape/drivers/image_generation_model/base_image_generation_model_driver.py

```
@define
class BaseImageGenerationModelDriver(SerializableMixin, ABC):
    @abstractmethod
    def get_generated_image(self, response: dict) -> bytes: ...

    @abstractmethod
    def text_to_image_request_parameters(
        self,
        prompts: list[str],
        image_width: int,
        image_height: int,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict[str, Any]: ...

    @abstractmethod
    def image_variation_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict[str, Any]: ...

    @abstractmethod
    def image_inpainting_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict[str, Any]: ...
```

(continues on next page)

(continued from previous page)

```
    @abstractmethod
    def image_outpainting_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict[str, Any]: ...
```

get_generated_image(response)abstractmethod

Source Code in griptape/drivers/image_generation_model/base_image_generation_model_driver.py

```
@abstractmethod
def get_generated_image(self, response: dict) -> bytes: ...
```

image_inpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)abstractmethod

Source Code in griptape/drivers/image_generation_model/base_image_generation_model_driver.py

```
@abstractmethod
def image_inpainting_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict[str, Any]: ...
```

image_outpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)abstractmethod

Source Code in griptape/drivers/image_generation_model/base_image_generation_model_driver.py

```
@abstractmethod
def image_outpainting_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict[str, Any]: ...
```

image_variation_request_parameters(prompts, image, negative_prompts=None, seed=None)abstractmethod

Source Code in griptape/drivers/image_generation_model/base_image_generation_model_driver.py

```
@abstractmethod
def image_variation_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict[str, Any]: ...
```

text_to_image_request_parameters(prompts, image_width, image_height, negative_prompts=None, seed=None)abstractmethod

Source Code in griptape/drivers/image_generation_model/base_image_generation_model_driver.py

```
@abstractmethod
def text_to_image_request_parameters(
    self,
    prompts: list[str],
    image_width: int,
    image_height: int,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict[str, Any]: ...
```

bedrock_stable_diffusion

- `__all__ = ['BedrockStableDiffusionImageGenerationModelDriver']` module-attribute

Bases: *BaseImageGenerationModelDriver* “)

Attributes

Name	Type	Description
<a href="bedrock_stable_diffusion.mdx#griptape.drivers.image_generation_model.bedrock_stable_diffusion.BedrockStableDiffusionImageGenerationModelDriver.cfg_scale" title="cfg_scale = field(default=7, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_model.bedrock_stable_diffusion.BedrockStableDiffusionImageGenerationModelDriver.cfg_scale)"><co class="language-python">	*int*	Specifies how strictly image generation follows the provided prompt. Defaults to 7.
mask_source	*int*	Specifies mask image configuration for image-to-image generations. Defaults to "MASK_IMAGE_BLACK".

continues on next page

Table 174 – continued from previous page

Name	Type	Description
<a href="bedrock_stable_diffusion.mdx#griptape.drivers.image_generation_model.bedrock_stable_diffusion.BedrockStableDiffusionImageGenerationModelDriver.style_preset" title="style_preset = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_model.bedrock_stable_diffusion.BedrockStableDiffusionImageGenerationModelDriver.style_preset)"><code class="language-python">	*Optional[str]*	If provided, specifies a specific image generation style preset.

continues on next page

Table 174 – continued from previous page

Name	Type	Description
<a href="bedrock_stable_diffusion.mdx#griptape.drivers.image_generation_model.bedrock_stable_diffusion.BedrockStableDiffusionImageGenerationModelDriver.clip_guidance_preset" title="clip_guidance_preset = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_model.bedrock_stable_diffusion.BedrockStableDiffusionImageGenerationModelDriver.clip_guidance_preset)"><code class="language-python">	*Optional[str]*	If provided, requests a specific clip guidance preset to be used in the diffusion process.

continues on next page

Table 174 – continued from previous page

Name	Type	Description
<a href="bedrock_stable_diffusion.mdx#griptape.drivers.image_generation_model.bedrock_stable_diffusion.BedrockStableDiffusionImageGenerationModelDriver.sampler" title="sampler = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_model.bedrock_stable_diffusion.BedrockStableDiffusionImageGenerationModelDriver.sampler)"><code class="language-python">	*Optional[str]*	If provided, requests a specific sampler to be used in the diffusion process.
<a href="bedrock_stable_diffusion.mdx#griptape.drivers.image_generation_model.bedrock_stable_diffusion.BedrockStableDiffusionImageGenerationModelDriver.steps" title="steps = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_model.bedrock_stable_diffusion.BedrockStableDiffusionImageGenerationModelDriver.steps)"><code class="language-python">	*Optional[int]*	If provided, specifies the number of diffusion steps to use in the image generation.

continues on next page

Table 174 – continued from previous page

Name	Type	Description
<a href="bedrock_stable_diffusion.mdx#griptape.drivers.image_generation_model.bedrock_stable_diffusion.BedrockStableDiffusionImageGenerationModelDriver.start_schedule" title="start_schedule = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_model.bedrock_stable_diffusion.BedrockStableDiffusionImageGenerationModelDriver.start_schedule)"><code class="language-python">	*Optional[float]*	If provided, specifies the start_schedule parameter used to determine the influence of the input image in image-to-image generation.

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
@define
class␣
↪BedrockStableDiffusionImageGenerationModelDriver(BaseImageGenerationModelDriver):
    """Image generation model driver for Stable Diffusion models on Amazon Bedrock.

    For more information on all supported parameters, see the Stable Diffusion␣
↪documentation:
        https://platform.stability.ai/docs/api-reference#tag/v1generation

    Attributes:
        cfg_scale: Specifies how strictly image generation follows the provided␣
↪prompt. Defaults to 7.
        mask_source: Specifies mask image configuration for image-to-image␣
↪generations. Defaults to "MASK_IMAGE_BLACK".
        style_preset: If provided, specifies a specific image generation style preset.
        clip_guidance_preset: If provided, requests a specific clip guidance preset␣
↪to be used in the diffusion process.
        sampler: If provided, requests a specific sampler to be used in the diffusion␣
↪process.
        steps: If provided, specifies the number of diffusion steps to use in the␣
```

(continues on next page)

(continued from previous page)

```
↪image generation.
        start_schedule: If provided, specifies the start_schedule parameter used to␣
↪determine the influence of the input
            image in image-to-image generation.
    """

    cfg_scale: int = field(default=7, kw_only=True, metadata={"serializable": True})
    style_preset: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    clip_guidance_preset: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    sampler: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    steps: Optional[int] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    start_schedule: Optional[float] = field(default=None, kw_only=True, metadata={
↪"serializable": True})

    def text_to_image_request_parameters(
        self,
        prompts: list[str],
        image_width: int,
        image_height: int,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        return self._request_parameters(
            prompts,
            width=image_width,
            height=image_height,
            negative_prompts=negative_prompts,
            seed=seed,
        )

    def image_variation_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        return self._request_parameters(prompts, image=image, negative_
↪prompts=negative_prompts, seed=seed)

    def image_inpainting_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
```

(continues on next page)

(continued from previous page)

```
        return self._request_parameters(
            prompts,
            image=image,
            mask=mask,
            mask_source="MASK_IMAGE_BLACK",
            negative_prompts=negative_prompts,
            seed=seed,
        )

    def image_outpainting_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        return self._request_parameters(
            prompts,
            image=image,
            mask=mask,
            mask_source="MASK_IMAGE_WHITE",
            negative_prompts=negative_prompts,
            seed=seed,
        )

    def _request_parameters(
        self,
        prompts: list[str],
        width: Optional[int] = None,
        height: Optional[int] = None,
        image: Optional[ImageArtifact] = None,
        mask: Optional[ImageArtifact] = None,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
        mask_source: Optional[str] = None,
    ) -> dict:
        if negative_prompts is None:
            negative_prompts = []

        text_prompts = [{"text": prompt, "weight": 1.0} for prompt in prompts]
        text_prompts += [{"text": negative_prompt, "weight": -1.0} for negative_
↪prompt in negative_prompts]

        request = {
            "text_prompts": text_prompts,
            "cfg_scale": self.cfg_scale,
            "style_preset": self.style_preset,
            "clip_guidance_preset": self.clip_guidance_preset,
            "sampler": self.sampler,
            "steps": self.steps,
            "seed": seed,
```

(continues on next page)

(continued from previous page)

```
            "start_schedule": self.start_schedule,
        }

        if image is not None:
            request["init_image"] = image.base64
            request["width"] = image.width
            request["height"] = image.height
        else:
            request["width"] = width
            request["height"] = height

        if mask is not None:
            if not mask_source:
                raise ValueError("mask_source must be provided when mask is provided")

            request["mask_source"] = mask_source
            request["mask_image"] = mask.base64

        return {k: v for k, v in request.items() if v is not None}

    def get_generated_image(self, response: dict) -> bytes:
        image_response = response["artifacts"][0]

        # finishReason may be SUCCESS, CONTENT_FILTERED, or ERROR.
        if image_response.get("finishReason") == "ERROR":
            raise Exception(f"Image generation failed: {image_response.
↪get(finishReason)}")
        if image_response.get("finishReason") == "CONTENT_FILTERED":
            logging.warning("Image generation triggered content filter and may be␣
↪blurred")

        return base64.decodebytes(bytes(image_response.get("base64"), "utf-8"))
```

- `cfg_scale = field(default=7, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `clip_guidance_preset = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `sampler = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `start_schedule = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `steps = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `style_preset = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_request_parameters(prompts, width=None, height=None, image=None, mask=None, negative_prompts=None, seed=None, mask_source=None)

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
def _request_parameters(
    self,
    prompts: list[str],
    width: Optional[int] = None,
    height: Optional[int] = None,
    image: Optional[ImageArtifact] = None,
    mask: Optional[ImageArtifact] = None,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
    mask_source: Optional[str] = None,
) -> dict:
    if negative_prompts is None:
        negative_prompts = []

    text_prompts = [{"text": prompt, "weight": 1.0} for prompt in prompts]
    text_prompts += [{"text": negative_prompt, "weight": -1.0} for negative_prompt in
↪negative_prompts]

    request = {
        "text_prompts": text_prompts,
        "cfg_scale": self.cfg_scale,
        "style_preset": self.style_preset,
        "clip_guidance_preset": self.clip_guidance_preset,
        "sampler": self.sampler,
        "steps": self.steps,
        "seed": seed,
        "start_schedule": self.start_schedule,
    }

    if image is not None:
        request["init_image"] = image.base64
        request["width"] = image.width
        request["height"] = image.height
    else:
        request["width"] = width
        request["height"] = height

    if mask is not None:
        if not mask_source:
            raise ValueError("mask_source must be provided when mask is provided")

        request["mask_source"] = mask_source
        request["mask_image"] = mask.base64

    return {k: v for k, v in request.items() if v is not None}
```

get_generated_image(response)

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
def get_generated_image(self, response: dict) -> bytes:
    image_response = response["artifacts"][0]

    # finishReason may be SUCCESS, CONTENT_FILTERED, or ERROR.
    if image_response.get("finishReason") == "ERROR":
        raise Exception(f"Image generation failed: {image_response.get(finishReason)}
↪")
    if image_response.get("finishReason") == "CONTENT_FILTERED":
        logging.warning("Image generation triggered content filter and may be blurred
↪")

    return base64.decodebytes(bytes(image_response.get("base64"), "utf-8"))
```

image_inpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
def image_inpainting_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    return self._request_parameters(
        prompts,
        image=image,
        mask=mask,
        mask_source="MASK_IMAGE_BLACK",
        negative_prompts=negative_prompts,
        seed=seed,
    )
```

image_outpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
def image_outpainting_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
```

(continues on next page)

(continued from previous page)

```
    seed: Optional[int] = None,
) -> dict:
    return self._request_parameters(
        prompts,
        image=image,
        mask=mask,
        mask_source="MASK_IMAGE_WHITE",
        negative_prompts=negative_prompts,
        seed=seed,
    )
```

image_variation_request_parameters(prompts, image, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
def image_variation_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    return self._request_parameters(prompts, image=image, negative_prompts=negative_
↪prompts, seed=seed)
```

text_to_image_request_parameters(prompts, image_width, image_height, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
def text_to_image_request_parameters(
    self,
    prompts: list[str],
    image_width: int,
    image_height: int,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    return self._request_parameters(
        prompts,
        width=image_width,
        height=image_height,
        negative_prompts=negative_prompts,
        seed=seed,
    )
```

bedrock_stable_diffusion_image_generation_model_driver

Bases: *BaseImageGenerationModelDriver* “)

Attributes

Name	Type	Description
<a href="bedrock_stable_diffusion_image_generation_model_driver.mdx#griptape.drivers.image_generation_model.bedrock_stable_diffusion_image_generation_model_driver.BedrockStableDiffusionImageGenerationModelDriver.cfg_scale" title="cfg_scale = field(default=7, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_model.bedrock_stable_diffusion_image_generation_model_driver.BedrockStableDiffusionImageGenerationModelDriver.cfg_scale)"><co class="language-python">	*int*	Specifies how strictly image generation follows the provided prompt. Defaults to 7.
mask_source	*int*	Specifies mask image configuration for image-to-image generations. Defaults to "MASK_IMAGE_BLACK".

continues on next page

Table 175 – continued from previous page

Name	Type	Description
<a href="bedrock_stable_diffusion_image_generation_model_driver.mdx#griptape.drivers.image_generation_model.bedrock_stable_diffusion_image_generation_model_driver.BedrockStableDiffusionImageGenerationModelDriver.style_preset" title="style_preset = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_model.bedrock_stable_diffusion_image_generation_model_driver.BedrockStableDiffusionImageGenerationModelDriver.style_preset)"><code class="language-python">	*Optional[str]*	If provided, specifies a specific image generation style preset.

continues on next page

Table 175 – continued from previous page

Name	Type	Description
<a href="bedrock_stable_diffusion_image_generation_model_driver.mdx#griptape.drivers.image_generation_model.bedrock_stable_diffusion_image_generation_model_driver.BedrockStableDiffusionImageGenerationModelDriver.clip_guidance_preset" title="clip_guidance_preset = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_model.bedrock_stable_diffusion_image_generation_model_driver.BedrockStableDiffusionImageGenerationModelDriver.clip_guidance_preset)"><code class="language-python">	*Optional[str]*	If provided, requests a specific clip guidance preset to be used in the diffusion process.

continues on next page

Table 175 – continued from previous page

Name	Type	Description
<a href="bedrock_stable_diffusion_image_generation_model_driver.mdx#griptape.drivers.image_generation_model.bedrock_stable_diffusion_image_generation_model_driver.BedrockStableDiffusionImageGenerationModelDriver.sampler" title="sampler = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_model.bedrock_stable_diffusion_image_generation_model_driver.BedrockStableDiffusionImageGenerationModelDriver.sampler)"><code class="language-python">	*Optional[str]*	If provided, requests a specific sampler to be used in the diffusion process.

continues on next page

Table 175 – continued from previous page

Name	Type	Description
<a href="bedrock_stable_diffusion_image_generation_model_driver.mdx#griptape.drivers.image_generation_model.bedrock_stable_diffusion_image_generation_model_driver.BedrockStableDiffusionImageGenerationModelDriver.steps" title="steps = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_model.bedrock_stable_diffusion_image_generation_model_driver.BedrockStableDiffusionImageGenerationModelDriver.steps)"><code class="language-python">	*Optional[int]*	If provided, specifies the number of diffusion steps to use in the image generation.

continues on next page

Table 175 – continued from previous page

Name	Type	Description
<a href="bedrock_stable_diffusion_image_generation_model_driver.mdx#griptape.drivers.image_generation_model.bedrock_stable_diffusion_image_generation_model_driver.BedrockStableDiffusionImageGenerationModelDriver.start_schedule" title="start_schedule = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_model.bedrock_stable_diffusion_image_generation_model_driver.BedrockStableDiffusionImageGenerationModelDriver.start_schedule)"><code class="language-python">	*Optional[float]*	If provided, specifies the start_schedule parameter used to determine the influence of the input image in image-to-image generation.

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
@define
class␣
↪BedrockStableDiffusionImageGenerationModelDriver(BaseImageGenerationModelDriver):
    """Image generation model driver for Stable Diffusion models on Amazon Bedrock.

    For more information on all supported parameters, see the Stable Diffusion␣
↪documentation:
        https://platform.stability.ai/docs/api-reference#tag/v1generation

    Attributes:
        cfg_scale: Specifies how strictly image generation follows the provided␣
↪prompt. Defaults to 7.
        mask_source: Specifies mask image configuration for image-to-image␣
↪generations. Defaults to "MASK_IMAGE_BLACK".
        style_preset: If provided, specifies a specific image generation style preset.
```

(continues on next page)

(continued from previous page)

```
        clip_guidance_preset: If provided, requests a specific clip guidance preset␣
↪to be used in the diffusion process.
        sampler: If provided, requests a specific sampler to be used in the diffusion␣
↪process.
        steps: If provided, specifies the number of diffusion steps to use in the␣
↪image generation.
        start_schedule: If provided, specifies the start_schedule parameter used to␣
↪determine the influence of the input
            image in image-to-image generation.
    """

    cfg_scale: int = field(default=7, kw_only=True, metadata={"serializable": True})
    style_preset: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    clip_guidance_preset: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    sampler: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    steps: Optional[int] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    start_schedule: Optional[float] = field(default=None, kw_only=True, metadata={
↪"serializable": True})

    def text_to_image_request_parameters(
        self,
        prompts: list[str],
        image_width: int,
        image_height: int,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        return self._request_parameters(
            prompts,
            width=image_width,
            height=image_height,
            negative_prompts=negative_prompts,
            seed=seed,
        )

    def image_variation_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        return self._request_parameters(prompts, image=image, negative_
↪prompts=negative_prompts, seed=seed)

    def image_inpainting_request_parameters(
        self,
        prompts: list[str],
```

(continues on next page)

(continued from previous page)

```
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        return self._request_parameters(
            prompts,
            image=image,
            mask=mask,
            mask_source="MASK_IMAGE_BLACK",
            negative_prompts=negative_prompts,
            seed=seed,
        )

    def image_outpainting_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        return self._request_parameters(
            prompts,
            image=image,
            mask=mask,
            mask_source="MASK_IMAGE_WHITE",
            negative_prompts=negative_prompts,
            seed=seed,
        )

    def _request_parameters(
        self,
        prompts: list[str],
        width: Optional[int] = None,
        height: Optional[int] = None,
        image: Optional[ImageArtifact] = None,
        mask: Optional[ImageArtifact] = None,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
        mask_source: Optional[str] = None,
    ) -> dict:
        if negative_prompts is None:
            negative_prompts = []

        text_prompts = [{"text": prompt, "weight": 1.0} for prompt in prompts]
        text_prompts += [{"text": negative_prompt, "weight": -1.0} for negative_
↪prompt in negative_prompts]

        request = {
            "text_prompts": text_prompts,
            "cfg_scale": self.cfg_scale,
```

(continues on next page)

(continued from previous page)

```
            "style_preset": self.style_preset,
            "clip_guidance_preset": self.clip_guidance_preset,
            "sampler": self.sampler,
            "steps": self.steps,
            "seed": seed,
            "start_schedule": self.start_schedule,
        }

        if image is not None:
            request["init_image"] = image.base64
            request["width"] = image.width
            request["height"] = image.height
        else:
            request["width"] = width
            request["height"] = height

        if mask is not None:
            if not mask_source:
                raise ValueError("mask_source must be provided when mask is provided")

            request["mask_source"] = mask_source
            request["mask_image"] = mask.base64

        return {k: v for k, v in request.items() if v is not None}

    def get_generated_image(self, response: dict) -> bytes:
        image_response = response["artifacts"][0]

        # finishReason may be SUCCESS, CONTENT_FILTERED, or ERROR.
        if image_response.get("finishReason") == "ERROR":
            raise Exception(f"Image generation failed: {image_response.
↪get(finishReason)}")
        if image_response.get("finishReason") == "CONTENT_FILTERED":
            logging.warning("Image generation triggered content filter and may be␣
↪blurred")

        return base64.decodebytes(bytes(image_response.get("base64"), "utf-8"))
```

- `cfg_scale = field(default=7, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `clip_guidance_preset = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `sampler = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `start_schedule = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `steps = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `style_preset = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_request_parameters(prompts, width=None, height=None, image=None, mask=None, negative_prompts=None, seed=None, mask_source=None)

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
def _request_parameters(
    self,
    prompts: list[str],
    width: Optional[int] = None,
    height: Optional[int] = None,
    image: Optional[ImageArtifact] = None,
    mask: Optional[ImageArtifact] = None,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
    mask_source: Optional[str] = None,
) -> dict:
    if negative_prompts is None:
        negative_prompts = []

    text_prompts = [{"text": prompt, "weight": 1.0} for prompt in prompts]
    text_prompts += [{"text": negative_prompt, "weight": -1.0} for negative_prompt in␣
↪negative_prompts]

    request = {
        "text_prompts": text_prompts,
        "cfg_scale": self.cfg_scale,
        "style_preset": self.style_preset,
        "clip_guidance_preset": self.clip_guidance_preset,
        "sampler": self.sampler,
        "steps": self.steps,
        "seed": seed,
        "start_schedule": self.start_schedule,
    }

    if image is not None:
        request["init_image"] = image.base64
        request["width"] = image.width
        request["height"] = image.height
    else:
        request["width"] = width
        request["height"] = height

    if mask is not None:
        if not mask_source:
            raise ValueError("mask_source must be provided when mask is provided")

        request["mask_source"] = mask_source
        request["mask_image"] = mask.base64

    return {k: v for k, v in request.items() if v is not None}
```

get_generated_image(response)

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
def get_generated_image(self, response: dict) -> bytes:
    image_response = response["artifacts"][0]

    # finishReason may be SUCCESS, CONTENT_FILTERED, or ERROR.
    if image_response.get("finishReason") == "ERROR":
        raise Exception(f"Image generation failed: {image_response.get(finishReason)}
↪")
    if image_response.get("finishReason") == "CONTENT_FILTERED":
        logging.warning("Image generation triggered content filter and may be blurred
↪")

    return base64.decodebytes(bytes(image_response.get("base64"), "utf-8"))
```

image_inpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
def image_inpainting_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    return self._request_parameters(
        prompts,
        image=image,
        mask=mask,
        mask_source="MASK_IMAGE_BLACK",
        negative_prompts=negative_prompts,
        seed=seed,
    )
```

image_outpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
def image_outpainting_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
```

(continues on next page)

(continued from previous page)

```
    seed: Optional[int] = None,
) -> dict:
    return self._request_parameters(
        prompts,
        image=image,
        mask=mask,
        mask_source="MASK_IMAGE_WHITE",
        negative_prompts=negative_prompts,
        seed=seed,
    )
```

image_variation_request_parameters(prompts, image, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
def image_variation_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    return self._request_parameters(prompts, image=image, negative_prompts=negative_
↪prompts, seed=seed)
```

text_to_image_request_parameters(prompts, image_width, image_height, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_stable_diffusion_image_generation_model_driver.py

```
def text_to_image_request_parameters(
    self,
    prompts: list[str],
    image_width: int,
    image_height: int,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    return self._request_parameters(
        prompts,
        width=image_width,
        height=image_height,
        negative_prompts=negative_prompts,
        seed=seed,
    )
```

bedrock_titan

- `__all__ = ['BedrockTitanImageGenerationModelDriver']` module-attribute

Bases: *BaseImageGenerationModelDriver* “)

Attributes

Name	Type	Description
<a href="bedrock_titan.mdx#griptape.drivers.image_generation_model.bedrock_titan.BedrockTitanImageGenerationModelDriver.quality" title="quality = field(default='standard', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_model.bedrock_titan.BedrockTitanImageGenerationModelDriver.quality)"><code class="language-python">	*str*	The quality of the generated image, defaults to standard.
<a href="bedrock_titan.mdx#griptape.drivers.image_generation_model.bedrock_titan.BedrockTitanImageGenerationModelDriver.cfg_scale" title="cfg_scale = field(default=7, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_model.bedrock_titan.BedrockTitanImageGenerationModelDriver.cfg_scale)"><co class="language-python">	*int*	Specifies how strictly image generation follows the provided prompt. Defaults to 7, (1.0 to 10.0].

continues on next page

Table 176 – continued from previous page

Name	Type	Description
<a href="bedrock_titan.mdx#griptape.drivers.image_generation_model.bedrock_titan.BedrockTitanImageGenerationModelDriver.outpainting_mode" title="outpainting_mode = field(default='PRECISE', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_model.bedrock_titan.BedrockTitanImageGenerationModelDriver.outpainting_mode)"><code class="language-python">	*str*	Specifies the outpainting mode, defaults to PRECISE.

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
@define
class BedrockTitanImageGenerationModelDriver(BaseImageGenerationModelDriver):
    """Image Generation Model Driver for Amazon Bedrock Titan Image Generator.

    Attributes:
        quality: The quality of the generated image, defaults to standard.
        cfg_scale: Specifies how strictly image generation follows the provided␣
↪prompt. Defaults to 7, (1.0 to 10.0].
        outpainting_mode: Specifies the outpainting mode, defaults to PRECISE.
    """

    quality: str = field(default="standard", kw_only=True, metadata={"serializable":␣
↪True})
    cfg_scale: int = field(default=7, kw_only=True, metadata={"serializable": True})
    outpainting_mode: str = field(default="PRECISE", kw_only=True, metadata={
↪"serializable": True})

    def text_to_image_request_parameters(
        self,
        prompts: list[str],
        image_width: int,
        image_height: int,
        negative_prompts: Optional[list[str]] = None,
```

(continues on next page)

(continued from previous page)

```
        seed: Optional[int] = None,
    ) -> dict:
        prompt = ", ".join(prompts)

        request = {
            "taskType": "TEXT_IMAGE",
            "textToImageParams": {"text": prompt},
            "imageGenerationConfig": {
                "numberOfImages": 1,
                "quality": self.quality,
                "width": image_width,
                "height": image_height,
                "cfgScale": self.cfg_scale,
            },
        }

        if negative_prompts:
            request["textToImageParams"]["negativeText"] = ", ".join(negative_prompts)

        if seed:
            request["imageGenerationConfig"]["seed"] = seed

        return self._add_common_params(request, image_width, image_height, seed=seed)

    def image_variation_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        prompt = ", ".join(prompts)

        request = {
            "taskType": "IMAGE_VARIATION",
            "imageVariationParams": {"text": prompt, "images": [image.base64]},
            "imageGenerationConfig": {
                "numberOfImages": 1,
                "quality": self.quality,
                "width": image.width,
                "height": image.height,
                "cfgScale": self.cfg_scale,
            },
        }

        if negative_prompts:
            request["imageVariationParams"]["negativeText"] = ", ".join(negative_
↪prompts)

        if seed:
            request["imageGenerationConfig"]["seed"] = seed
```

(continues on next page)

(continued from previous page)

```
        return self._add_common_params(request, image.width, image.height, seed=seed)

    def image_inpainting_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        prompt = ", ".join(prompts)

        request = {
            "taskType": "INPAINTING",
            "inPaintingParams": {"text": prompt, "image": image.base64, "maskImage":␣
↪mask.base64},
        }

        if negative_prompts:
            request["inPaintingParams"]["negativeText"] = ", ".join(negative_prompts)

        return self._add_common_params(request, image.width, image.height, seed=seed)

    def image_outpainting_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        prompt = ", ".join(prompts)

        request = {
            "taskType": "OUTPAINTING",
            "outPaintingParams": {
                "text": prompt,
                "image": image.base64,
                "maskImage": mask.base64,
                "outPaintingMode": self.outpainting_mode,
            },
        }

        if negative_prompts:
            request["outPaintingParams"]["negativeText"] = ", ".join(negative_prompts)

        return self._add_common_params(request, image.width, image.height, seed=seed)

    def get_generated_image(self, response: dict) -> bytes:
        b64_image_data = response["images"][0]

        return base64.decodebytes(bytes(b64_image_data, "utf-8"))
```

(continues on next page)

(continued from previous page)

```
    def _add_common_params(self, request: dict[str, Any], width: int, height: int,␣
↪seed: Optional[int] = None) -> dict:
        request["imageGenerationConfig"] = {
            "numberOfImages": 1,
            "quality": self.quality,
            "width": width,
            "height": height,
            "cfgScale": self.cfg_scale,
        }

        if seed:
            request["imageGenerationConfig"]["seed"] = seed

        return request
```

- `cfg_scale = field(default=7, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `outpainting_mode = field(default='PRECISE', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `quality = field(default='standard', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_add_common_params(request, width, height, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
def _add_common_params(self, request: dict[str, Any], width: int, height: int, seed:␣
↪Optional[int] = None) -> dict:
    request["imageGenerationConfig"] = {
        "numberOfImages": 1,
        "quality": self.quality,
        "width": width,
        "height": height,
        "cfgScale": self.cfg_scale,
    }

    if seed:
        request["imageGenerationConfig"]["seed"] = seed

    return request
```

get_generated_image(response)

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
def get_generated_image(self, response: dict) -> bytes:
    b64_image_data = response["images"][0]

    return base64.decodebytes(bytes(b64_image_data, "utf-8"))
```

image_inpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
def image_inpainting_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    prompt = ", ".join(prompts)

    request = {
        "taskType": "INPAINTING",
        "inPaintingParams": {"text": prompt, "image": image.base64, "maskImage": mask.
↪base64},
    }

    if negative_prompts:
        request["inPaintingParams"]["negativeText"] = ", ".join(negative_prompts)

    return self._add_common_params(request, image.width, image.height, seed=seed)
```

image_outpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
def image_outpainting_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    prompt = ", ".join(prompts)

    request = {
        "taskType": "OUTPAINTING",
        "outPaintingParams": {
            "text": prompt,
            "image": image.base64,
            "maskImage": mask.base64,
            "outPaintingMode": self.outpainting_mode,
        },
    }

    if negative_prompts:
        request["outPaintingParams"]["negativeText"] = ", ".join(negative_prompts)
```

(continues on next page)

(continued from previous page)

```
    return self._add_common_params(request, image.width, image.height, seed=seed)
```

image_variation_request_parameters(prompts, image, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
def image_variation_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    prompt = ", ".join(prompts)

    request = {
        "taskType": "IMAGE_VARIATION",
        "imageVariationParams": {"text": prompt, "images": [image.base64]},
        "imageGenerationConfig": {
            "numberOfImages": 1,
            "quality": self.quality,
            "width": image.width,
            "height": image.height,
            "cfgScale": self.cfg_scale,
        },
    }

    if negative_prompts:
        request["imageVariationParams"]["negativeText"] = ", ".join(negative_prompts)

    if seed:
        request["imageGenerationConfig"]["seed"] = seed

    return self._add_common_params(request, image.width, image.height, seed=seed)
```

text_to_image_request_parameters(prompts, image_width, image_height, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
def text_to_image_request_parameters(
    self,
    prompts: list[str],
    image_width: int,
    image_height: int,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    prompt = ", ".join(prompts)
```

(continues on next page)

(continued from previous page)

```
    request = {
        "taskType": "TEXT_IMAGE",
        "textToImageParams": {"text": prompt},
        "imageGenerationConfig": {
            "numberOfImages": 1,
            "quality": self.quality,
            "width": image_width,
            "height": image_height,
            "cfgScale": self.cfg_scale,
        },
    }

    if negative_prompts:
        request["textToImageParams"]["negativeText"] = ", ".join(negative_prompts)

    if seed:
        request["imageGenerationConfig"]["seed"] = seed

    return self._add_common_params(request, image_width, image_height, seed=seed)
```

bedrock_titan_image_generation_model_driver

Bases: *BaseImageGenerationModelDriver* “)

Attributes

Name	Type	Description
<a href="bedrock_titan_image_generation_model_driver.mdx#griptape.drivers.image_generation_model.bedrock_titan_image_generation_model_driver.BedrockTitanImageGenerationModelDriver.quality" title="quality = field(default='standard', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_model.bedrock_titan_image_generation_model_driver.BedrockTitanImageGenerationModelDriver.quality)"><code class="language-python">	*str*	The quality of the generated image, defaults to standard.
<a href="bedrock_titan_image_generation_model_driver.mdx#griptape.drivers.image_generation_model.bedrock_titan_image_generation_model_driver.BedrockTitanImageGenerationModelDriver.cfg_scale" title="cfg_scale = field(default=7, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_model.bedrock_titan_image_generation_model_driver.BedrockTitanImageGenerationModelDriver.cfg_scale)"><co class="language-python">	*int*	Specifies how strictly image generation follows the provided prompt. Defaults to 7, (1.0 to 10.0].

continues on next page

Table 177 – continued from previous page

Name	Type	Description
<a href="bedrock_titan_image_generation_model_driver.mdx#griptape.drivers.image_generation_model.bedrock_titan_image_generation_model_driver.BedrockTitanImageGenerationModelDriver.outpainting_mode" title="outpainting_mode = field(default='PRECISE', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_model.bedrock_titan_image_generation_model_driver.BedrockTitanImageGenerationModelDriver.outpainting_mode)"><code class="language-python">	*str*	Specifies the outpainting mode, defaults to PRECISE.

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
@define
class BedrockTitanImageGenerationModelDriver(BaseImageGenerationModelDriver):
    """Image Generation Model Driver for Amazon Bedrock Titan Image Generator.

    Attributes:
        quality: The quality of the generated image, defaults to standard.
        cfg_scale: Specifies how strictly image generation follows the provided␣
↪prompt. Defaults to 7, (1.0 to 10.0].
        outpainting_mode: Specifies the outpainting mode, defaults to PRECISE.
    """

    quality: str = field(default="standard", kw_only=True, metadata={"serializable":␣
↪True})
    cfg_scale: int = field(default=7, kw_only=True, metadata={"serializable": True})
    outpainting_mode: str = field(default="PRECISE", kw_only=True, metadata={
↪"serializable": True})

    def text_to_image_request_parameters(
        self,
        prompts: list[str],
```

(continues on next page)

(continued from previous page)

```
        image_width: int,
        image_height: int,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        prompt = ", ".join(prompts)

        request = {
            "taskType": "TEXT_IMAGE",
            "textToImageParams": {"text": prompt},
            "imageGenerationConfig": {
                "numberOfImages": 1,
                "quality": self.quality,
                "width": image_width,
                "height": image_height,
                "cfgScale": self.cfg_scale,
            },
        }

        if negative_prompts:
            request["textToImageParams"]["negativeText"] = ", ".join(negative_prompts)

        if seed:
            request["imageGenerationConfig"]["seed"] = seed

        return self._add_common_params(request, image_width, image_height, seed=seed)

    def image_variation_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        prompt = ", ".join(prompts)

        request = {
            "taskType": "IMAGE_VARIATION",
            "imageVariationParams": {"text": prompt, "images": [image.base64]},
            "imageGenerationConfig": {
                "numberOfImages": 1,
                "quality": self.quality,
                "width": image.width,
                "height": image.height,
                "cfgScale": self.cfg_scale,
            },
        }

        if negative_prompts:
            request["imageVariationParams"]["negativeText"] = ", ".join(negative_
↪prompts)
```

(continues on next page)

(continued from previous page)

```
        if seed:
            request["imageGenerationConfig"]["seed"] = seed

        return self._add_common_params(request, image.width, image.height, seed=seed)

    def image_inpainting_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        prompt = ", ".join(prompts)

        request = {
            "taskType": "INPAINTING",
            "inPaintingParams": {"text": prompt, "image": image.base64, "maskImage":␣
↪mask.base64},
        }

        if negative_prompts:
            request["inPaintingParams"]["negativeText"] = ", ".join(negative_prompts)

        return self._add_common_params(request, image.width, image.height, seed=seed)

    def image_outpainting_request_parameters(
        self,
        prompts: list[str],
        image: ImageArtifact,
        mask: ImageArtifact,
        negative_prompts: Optional[list[str]] = None,
        seed: Optional[int] = None,
    ) -> dict:
        prompt = ", ".join(prompts)

        request = {
            "taskType": "OUTPAINTING",
            "outPaintingParams": {
                "text": prompt,
                "image": image.base64,
                "maskImage": mask.base64,
                "outPaintingMode": self.outpainting_mode,
            },
        }

        if negative_prompts:
            request["outPaintingParams"]["negativeText"] = ", ".join(negative_prompts)

        return self._add_common_params(request, image.width, image.height, seed=seed)

    def get_generated_image(self, response: dict) -> bytes:
```

(continues on next page)

(continued from previous page)

```
        b64_image_data = response["images"][0]

        return base64.decodebytes(bytes(b64_image_data, "utf-8"))

    def _add_common_params(self, request: dict[str, Any], width: int, height: int,␣
↪seed: Optional[int] = None) -> dict:
        request["imageGenerationConfig"] = {
            "numberOfImages": 1,
            "quality": self.quality,
            "width": width,
            "height": height,
            "cfgScale": self.cfg_scale,
        }

        if seed:
            request["imageGenerationConfig"]["seed"] = seed

        return request
```

- `cfg_scale = field(default=7, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `outpainting_mode = field(default='PRECISE', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `quality = field(default='standard', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_add_common_params(request, width, height, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
def _add_common_params(self, request: dict[str, Any], width: int, height: int, seed:␣
↪Optional[int] = None) -> dict:
    request["imageGenerationConfig"] = {
        "numberOfImages": 1,
        "quality": self.quality,
        "width": width,
        "height": height,
        "cfgScale": self.cfg_scale,
    }

    if seed:
        request["imageGenerationConfig"]["seed"] = seed

    return request
```

get_generated_image(response)

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
def get_generated_image(self, response: dict) -> bytes:
    b64_image_data = response["images"][0]
```

(continues on next page)

(continued from previous page)

```
    return base64.decodebytes(bytes(b64_image_data, "utf-8"))
```

image_inpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
def image_inpainting_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    prompt = ", ".join(prompts)

    request = {
        "taskType": "INPAINTING",
        "inPaintingParams": {"text": prompt, "image": image.base64, "maskImage": mask.
↪base64},
    }

    if negative_prompts:
        request["inPaintingParams"]["negativeText"] = ", ".join(negative_prompts)

    return self._add_common_params(request, image.width, image.height, seed=seed)
```

image_outpainting_request_parameters(prompts, image, mask, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
def image_outpainting_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    mask: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    prompt = ", ".join(prompts)

    request = {
        "taskType": "OUTPAINTING",
        "outPaintingParams": {
            "text": prompt,
            "image": image.base64,
            "maskImage": mask.base64,
            "outPaintingMode": self.outpainting_mode,
```

(continues on next page)

(continued from previous page)

```
        },
    }

    if negative_prompts:
        request["outPaintingParams"]["negativeText"] = ", ".join(negative_prompts)

    return self._add_common_params(request, image.width, image.height, seed=seed)
```

image_variation_request_parameters(prompts, image, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
def image_variation_request_parameters(
    self,
    prompts: list[str],
    image: ImageArtifact,
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    prompt = ", ".join(prompts)

    request = {
        "taskType": "IMAGE_VARIATION",
        "imageVariationParams": {"text": prompt, "images": [image.base64]},
        "imageGenerationConfig": {
            "numberOfImages": 1,
            "quality": self.quality,
            "width": image.width,
            "height": image.height,
            "cfgScale": self.cfg_scale,
        },
    }

    if negative_prompts:
        request["imageVariationParams"]["negativeText"] = ", ".join(negative_prompts)

    if seed:
        request["imageGenerationConfig"]["seed"] = seed

    return self._add_common_params(request, image.width, image.height, seed=seed)
```

text_to_image_request_parameters(prompts, image_width, image_height, negative_prompts=None, seed=None)

Source Code in griptape/drivers/image_generation_model/bedrock_titan_image_generation_model_driver.py

```
def text_to_image_request_parameters(
    self,
    prompts: list[str],
    image_width: int,
    image_height: int,
```

(continues on next page)

(continued from previous page)

```
    negative_prompts: Optional[list[str]] = None,
    seed: Optional[int] = None,
) -> dict:
    prompt = ", ".join(prompts)

    request = {
        "taskType": "TEXT_IMAGE",
        "textToImageParams": {"text": prompt},
        "imageGenerationConfig": {
            "numberOfImages": 1,
            "quality": self.quality,
            "width": image_width,
            "height": image_height,
            "cfgScale": self.cfg_scale,
        },
    }

    if negative_prompts:
        request["textToImageParams"]["negativeText"] = ", ".join(negative_prompts)

    if seed:
        request["imageGenerationConfig"]["seed"] = seed

    return self._add_common_params(request, image_width, image_height, seed=seed)
```

image_generation_pipeline

- `__all__ = ['BaseDiffusionImageGenerationPipelineDriver']` module-attribute

Bases:

ABC

Source Code in griptape/drivers/image_generation_pipeline/base_image_generation_pipeline_driver.py

```
@define
class BaseDiffusionImageGenerationPipelineDriver(ABC):
    @abstractmethod
    def prepare_pipeline(self, model: str, device: Optional[str]) -> Any: ...

    @abstractmethod
    def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:␣
↪...

    @abstractmethod
    def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict: ...

    @property
    @abstractmethod
    def output_image_dimensions(self) -> tuple[int, int]: ...
```

- `output_image_dimensions` abstractmethod property

make_additional_params(negative_prompts, device)abstractmethod

Source Code in griptape/drivers/image_generation_pipeline/base_image_generation_pipeline_driver.py

```
@abstractmethod
def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict: ...
```

make_image_param(image)abstractmethod

Source Code in griptape/drivers/image_generation_pipeline/base_image_generation_pipeline_driver.py

```
@abstractmethod
def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]: ...
```

prepare_pipeline(model, device)abstractmethod

Source Code in griptape/drivers/image_generation_pipeline/base_image_generation_pipeline_driver.py

```
@abstractmethod
def prepare_pipeline(self, model: str, device: Optional[str]) -> Any: ...
```

base_image_generation_pipeline_driver

Bases:

ABC

Source Code in griptape/drivers/image_generation_pipeline/base_image_generation_pipeline_driver.py

```
@define
class BaseDiffusionImageGenerationPipelineDriver(ABC):
    @abstractmethod
    def prepare_pipeline(self, model: str, device: Optional[str]) -> Any: ...

    @abstractmethod
    def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:␣
↪...

    @abstractmethod
    def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict: ...

    @property
    @abstractmethod
    def output_image_dimensions(self) -> tuple[int, int]: ...
```

- `output_image_dimensions` abstractmethod property

make_additional_params(negative_prompts, device)abstractmethod

Source Code in griptape/drivers/image_generation_pipeline/base_image_generation_pipeline_driver.py

```
@abstractmethod
def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict: ...
```

make_image_param(image)abstractmethod

Source Code in griptape/drivers/image_generation_pipeline/base_image_generation_pipeline_driver.py

```
@abstractmethod
def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]: ...
```

prepare_pipeline(model, device)abstractmethod

Source Code in griptape/drivers/image_generation_pipeline/base_image_generation_pipeline_driver.py

```
@abstractmethod
def prepare_pipeline(self, model: str, device: Optional[str]) -> Any: ...
```

stable_diffusion_3

- `__all__ = ['StableDiffusion3ImageGenerationPipelineDriver']` module-attribute

Bases: *BaseDiffusionImageGenerationPipelineDriver* “)

Attributes

Name	Type	Description
<a href="stable_diffusion_3.mdx#griptape.drivers.image_generation_pipeline.stable_diffusion_3.StableDiffusion3ImageGenerationPipelineDriver.width" title="width = field(default=1024, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_pipeline.stable_diffusion_3.StableDiffusion3ImageGenerationPipelineDriver.width) class="language-python">	*int*	The width of the generated image. Defaults to 1024. Must be a multiple of 64.

continues on next page

Table 178 – continued from previous page

Name	Type	Description
<a href="stable_diffusion_3.mdx#griptape.drivers.image_generation_pipeline.stable_diffusion_3.StableDiffusion3ImageGenerationPipelineDriver.height" title="height = field(default=1024, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_pipeline.stable_diffusion_3.StableDiffusion3ImageGenerationPipelineDriver.height" class="language-python">	*int*	The height of the generated image. Defaults to 1024. Must be a multiple of 64.
<a href="stable_diffusion_3.mdx#griptape.drivers.image_generation_pipeline.stable_diffusion_3.StableDiffusion3ImageGenerationPipelineDriver.seed" title="seed = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_pipeline.stable_diffusion_3.StableDiffusion3ImageGenerationPipelineDriver.seed)" class="language-python">	*Optional[int]*	The random seed to use for image generation. If not provided, a random seed will be used.

continues on next page

Table 178 – continued from previous page

Name	Type	Description
<a href="stable_diffusion_3.mdx#griptape.drivers.image_generation_pipeline.stable_diffusion_3.StableDiffusion3ImageGenerationPipelineDriver.guidance_scale" title="guidance_scale = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_pipeline.stable_diffusion_3.StableDiffusion3ImageGenerationPipelineDriver.guidance_scale)"><code class="language-python">	*Optional[float]*	The strength of the guidance loss. If not provided, the default value will be used.
<a href="stable_diffusion_3.mdx#griptape.drivers.image_generation_pipeline.stable_diffusion_3.StableDiffusion3ImageGenerationPipelineDriver.steps" title="steps = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_pipeline.stable_diffusion_3.StableDiffusion3ImageGenerationPipelineDriver.steps)" class="language-python">	*Optional[int]*	The number of inference steps to use in image generation. If not provided, the default value will be used.

continues on next page

Table 178 – continued from previous page

Name	Type	Description
<a href="stable_diffusion_3.mdx#griptape.drivers.image_generation_pipeline.stable_diffusion_3.StableDiffusion3ImageGenerationPipelineDriver.torch_ title="torch_dtype = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_pipeline.stable_diffusion_3.StableDiffusion3ImageGenerationPipelineDriver.torch_ class="language-python">	*Optional[dtype]*	The torch data type to use for image generation. If not provided, the default value will be used.

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_image_generation_pipeline_driver.py

```
@define
class StableDiffusion3ImageGenerationPipelineDriver(BaseDiffusionImageGenerationPipelineDriver):
    """Image generation model driver for Stable Diffusion 3 models.

    For more information, see the HuggingFace documentation for the StableDiffusion3Pipeline:
        https://huggingface.co/docs/diffusers/main/en/api/pipelines/stable_diffusion/stable_diffusion_3

    Attributes:
        width: The width of the generated image. Defaults to 1024. Must be a multiple of 64.
        height: The height of the generated image. Defaults to 1024. Must be a multiple of 64.
        seed: The random seed to use for image generation. If not provided, a random seed will be used.
        guidance_scale: The strength of the guidance loss. If not provided, the default value will be used.
        steps: The number of inference steps to use in image generation. If not provided, the default value will be used.
        torch_dtype: The torch data type to use for image generation. If not provided, the default value will be used.
    """
```

(continues on next page)

(continued from previous page)

```
    width: int = field(default=1024, kw_only=True, metadata={"serializable": True})
    height: int = field(default=1024, kw_only=True, metadata={"serializable": True})
    seed: Optional[int] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    guidance_scale: Optional[float] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    steps: Optional[int] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    torch_dtype: Optional[torch.dtype] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    enable_model_cpu_offload: bool = field(default=False, kw_only=True, metadata={
↪"serializable": True})
    drop_t5_encoder: bool = field(default=False, kw_only=True, metadata={"serializable
↪": True})

    def prepare_pipeline(self, model: str, device: Optional[str]) -> Any:
        sd3_pipeline = import_optional_dependency(
            "diffusers.pipelines.stable_diffusion_3.pipeline_stable_diffusion_3"
        ).StableDiffusion3Pipeline

        pipeline_params = {}
        if self.torch_dtype is not None:
            pipeline_params["torch_dtype"] = self.torch_dtype

        if self.drop_t5_encoder:
            pipeline_params["text_encoder_3"] = None
            pipeline_params["tokenizer_3"] = None

        # A model can be provided either as a path to a local file
        # or as a HuggingFace model repo name.
        if os.path.isfile(model):
            # If the model provided is a local file (not a directory),
            # we load it using the from_single_file method.
            pipeline = sd3_pipeline.from_single_file(model, **pipeline_params)
        else:
            # If the model is a local directory or hosted on HuggingFace,
            # we load it using the from_pretrained method.
            pipeline = sd3_pipeline.from_pretrained(model, **pipeline_params)

        if self.enable_model_cpu_offload:
            pipeline.enable_model_cpu_offload()

        # Move inference to particular device if requested.
        if device is not None:
            pipeline.to(device)

        return pipeline

    def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:
        return None
```

(continues on next page)

(continued from previous page)

```
    def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict[str, Any]:
        torch_generator = import_optional_dependency("torch").Generator

        additional_params = {}
        if negative_prompts:
            additional_params["negative_prompt"] = ", ".join(negative_prompts)

        if self.width is not None:
            additional_params["width"] = self.width

        if self.height is not None:
            additional_params["height"] = self.height

        if self.seed is not None:
            additional_params["generator"] = [torch_generator(device=device).manual_
↪seed(self.seed)]

        if self.guidance_scale is not None:
            additional_params["guidance_scale"] = self.guidance_scale

        if self.steps is not None:
            additional_params["num_inference_steps"] = self.steps

        return additional_params

    @property
    def output_image_dimensions(self) -> tuple[int, int]:
        return self.width, self.height
```

- `drop_t5_encoder = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `enable_model_cpu_offload = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `guidance_scale = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `height = field(default=1024, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output_image_dimensions` property
- `seed = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `steps = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `torch_dtype = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `width = field(default=1024, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

make_additional_params(negative_prompts, device)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_image_generation_pipeline_driver.py

```
def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict[str, Any]:
    torch_generator = import_optional_dependency("torch").Generator

    additional_params = {}
    if negative_prompts:
        additional_params["negative_prompt"] = ", ".join(negative_prompts)

    if self.width is not None:
        additional_params["width"] = self.width

    if self.height is not None:
        additional_params["height"] = self.height

    if self.seed is not None:
        additional_params["generator"] = [torch_generator(device=device).manual_
↪seed(self.seed)]

    if self.guidance_scale is not None:
        additional_params["guidance_scale"] = self.guidance_scale

    if self.steps is not None:
        additional_params["num_inference_steps"] = self.steps

    return additional_params
```

make_image_param(image)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_image_generation_pipeline_driver.py

```
def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:
    return None
```

prepare_pipeline(model, device)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_image_generation_pipeline_driver.py

```
def prepare_pipeline(self, model: str, device: Optional[str]) -> Any:
    sd3_pipeline = import_optional_dependency(
        "diffusers.pipelines.stable_diffusion_3.pipeline_stable_diffusion_3"
    ).StableDiffusion3Pipeline

    pipeline_params = {}
    if self.torch_dtype is not None:
        pipeline_params["torch_dtype"] = self.torch_dtype

    if self.drop_t5_encoder:
        pipeline_params["text_encoder_3"] = None
        pipeline_params["tokenizer_3"] = None
```

(continues on next page)

(continued from previous page)

```
    # A model can be provided either as a path to a local file
    # or as a HuggingFace model repo name.
    if os.path.isfile(model):
        # If the model provided is a local file (not a directory),
        # we load it using the from_single_file method.
        pipeline = sd3_pipeline.from_single_file(model, **pipeline_params)
    else:
        # If the model is a local directory or hosted on HuggingFace,
        # we load it using the from_pretrained method.
        pipeline = sd3_pipeline.from_pretrained(model, **pipeline_params)

    if self.enable_model_cpu_offload:
        pipeline.enable_model_cpu_offload()

    # Move inference to particular device if requested.
    if device is not None:
        pipeline.to(device)

    return pipeline
```

stable_diffusion_3_controlnet

- `__all__ = ['StableDiffusion3ControlNetImageGenerationPipelineDriver']` module-attribute

Bases: *StableDiffusion3ImageGenerationPipelineDriver* “)

Attributes

Name	Type	Description
<a href="stable_diffusion_3_controlnet.mdx#griptape.drivers.image_generation_pipeline.stable_diffusion_3_controlnet.StableDiffusion3ControlNetImageGenerationPipelineDriver.controlnet_model" title="controlnet_model = field(kw_only=True) class-attribute instance-attribute (griptape.drivers.image_generation_pipeline.stable_diffusion_3_controlnet.StableDiffusion3ControlNetImageGenerationPipelineDriver.controlnet_model)"><code class="language-python">	*str*	The ControlNet model to use for image generation.

continues on next page

Table 179 – continued from previous page

Name	Type	Description
<a href="stable_diffusion_3_controlnet.mdx#griptape.drivers.image_generation_pipeline.stable_diffusion_3_controlnet.StableDiffusion3ControlNetImageGenerationPipelineDriver.controlnet_conditioning_scale" title="controlnet_conditioning_scale = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_pipeline.stable_diffusion_3_controlnet.StableDiffusion3ControlNetImageGenerationPipelineDriver.controlnet_conditioning_scale)"><code class="language-python">	*Optional[float]*	The conditioning scale for the ControlNet model. Defaults to None.

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_controlnet_image_generation_pipeline_driver.py

```
@define
class␣
↪StableDiffusion3ControlNetImageGenerationPipelineDriver(StableDiffusion3ImageGenerationPipelineDriv
    """Image generation model driver for Stable Diffusion 3 models with ControlNet.

    For more information, see the HuggingFace documentation for the␣
↪StableDiffusion3ControlNetPipeline:
        https://huggingface.co/docs/diffusers/en/api/pipelines/controlnet_sd3

    Attributes:
        controlnet_model: The ControlNet model to use for image generation.
        controlnet_conditioning_scale: The conditioning scale for the ControlNet␣
↪model. Defaults to None.
    """
```

(continues on next page)

(continued from previous page)

```
    controlnet_model: str = field(kw_only=True)
    controlnet_conditioning_scale: Optional[float] = field(default=None, kw_only=True,
↪ metadata={"serializable": True})

    def prepare_pipeline(self, model: str, device: Optional[str]) -> Any:
        sd3_controlnet_model = import_optional_dependency("diffusers.models.
↪controlnet_sd3").SD3ControlNetModel
        sd3_controlnet_pipeline = import_optional_dependency(
            "diffusers.pipelines.controlnet_sd3.pipeline_stable_diffusion_3_controlnet
↪"
        ).StableDiffusion3ControlNetPipeline

        pipeline_params = {}
        controlnet_pipeline_params = {}
        if self.torch_dtype is not None:
            pipeline_params["torch_dtype"] = self.torch_dtype
            controlnet_pipeline_params["torch_dtype"] = self.torch_dtype

        if self.drop_t5_encoder:
            pipeline_params["text_encoder_3"] = None
            pipeline_params["tokenizer_3"] = None

        # For both Stable Diffusion and ControlNet, models can be provided either
        # as a path to a local file or as a HuggingFace model repo name.
        # We use the from_single_file method if the model is a local file and the
        # from_pretrained method if the model is a local directory or hosted on␣
↪HuggingFace.
        if os.path.isfile(self.controlnet_model):
            pipeline_params["controlnet"] = sd3_controlnet_model.from_single_file(
                self.controlnet_model, **controlnet_pipeline_params
            )
        else:
            pipeline_params["controlnet"] = sd3_controlnet_model.from_pretrained(
                self.controlnet_model, **controlnet_pipeline_params
            )

        if os.path.isfile(model):
            pipeline = sd3_controlnet_pipeline.from_single_file(model, **pipeline_
↪params)
        else:
            pipeline = sd3_controlnet_pipeline.from_pretrained(model, **pipeline_
↪params)

        if self.enable_model_cpu_offload:
            pipeline.enable_model_cpu_offload()

        if device is not None:
            pipeline.to(device)

        return pipeline

    def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:
```

(continues on next page)

(continued from previous page)

```
        if image is None:
            raise ValueError("Input image is required for ControlNet pipelines.")

        return {"control_image": image}

    def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict[str, Any]:
        additional_params = super().make_additional_params(negative_prompts, device)

        del additional_params["height"]
        del additional_params["width"]

        if self.controlnet_conditioning_scale is not None:
            additional_params["controlnet_conditioning_scale"] = self.controlnet_
↪conditioning_scale

        return additional_params
```

- `controlnet_conditioning_scale = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `controlnet_model = field(kw_only=True)` class-attribute instance-attribute

make_additional_params(negative_prompts, device)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_controlnet_image_generation_pipeline_driver.py

```
def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict[str, Any]:
    additional_params = super().make_additional_params(negative_prompts, device)

    del additional_params["height"]
    del additional_params["width"]

    if self.controlnet_conditioning_scale is not None:
        additional_params["controlnet_conditioning_scale"] = self.controlnet_
↪conditioning_scale

    return additional_params
```

make_image_param(image)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_controlnet_image_generation_pipeline_driver.py

```
def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:
    if image is None:
        raise ValueError("Input image is required for ControlNet pipelines.")

    return {"control_image": image}
```

prepare_pipeline(model, device)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_controlnet_image_generation_pipeline_driver.py

```
def prepare_pipeline(self, model: str, device: Optional[str]) -> Any:
    sd3_controlnet_model = import_optional_dependency("diffusers.models.controlnet_sd3
↪").SD3ControlNetModel
    sd3_controlnet_pipeline = import_optional_dependency(
        "diffusers.pipelines.controlnet_sd3.pipeline_stable_diffusion_3_controlnet"
    ).StableDiffusion3ControlNetPipeline

    pipeline_params = {}
    controlnet_pipeline_params = {}
    if self.torch_dtype is not None:
        pipeline_params["torch_dtype"] = self.torch_dtype
        controlnet_pipeline_params["torch_dtype"] = self.torch_dtype

    if self.drop_t5_encoder:
        pipeline_params["text_encoder_3"] = None
        pipeline_params["tokenizer_3"] = None

    # For both Stable Diffusion and ControlNet, models can be provided either
    # as a path to a local file or as a HuggingFace model repo name.
    # We use the from_single_file method if the model is a local file and the
    # from_pretrained method if the model is a local directory or hosted on␣
↪HuggingFace.
    if os.path.isfile(self.controlnet_model):
        pipeline_params["controlnet"] = sd3_controlnet_model.from_single_file(
            self.controlnet_model, **controlnet_pipeline_params
        )
    else:
        pipeline_params["controlnet"] = sd3_controlnet_model.from_pretrained(
            self.controlnet_model, **controlnet_pipeline_params
        )

    if os.path.isfile(model):
        pipeline = sd3_controlnet_pipeline.from_single_file(model, **pipeline_params)
    else:
        pipeline = sd3_controlnet_pipeline.from_pretrained(model, **pipeline_params)

    if self.enable_model_cpu_offload:
        pipeline.enable_model_cpu_offload()

    if device is not None:
        pipeline.to(device)

    return pipeline
```

stable_diffusion_3_controlnet_image_generation_pipeline_driver

Bases: *StableDiffusion3ImageGenerationPipelineDriver* “)

Attributes

Name	Type	Description
<a href="stable_diffusion_3_controlnet_image_generation_pipeline_driver.mdx#griptape.drivers.image_generation_pipeline.stable_diffusion_3_controlnet_image_generation_pipeline_driver.StableDiffusion3ControlNetImageGenerationPipelineDriver.controlnet_model" title="controlnet_model = field(kw_only=True) class-attribute instance-attribute (griptape.drivers.image_generation_pipeline.stable_diffusion_3_controlnet_image_generation_pipeline_driver.StableDiffusion3ControlNetImageGenerationPipelineDriver.controlnet_model)"><code class="language-python">	*str*	The ControlNet model to use for image generation.

continues on next page

Table 180 – continued from previous page

Name	Type	Description
<a href="stable_diffusion_3_controlnet_image_generation_pipeline_driver.mdx#tape.drivers.image_generation_pipeline.stable_diffusion_3_controlnet_image_generation_pipeline_driver.StableDiffusion3ControlNetImageGenerationPipelineDriver.controlnet_conditioning_scale" title="controlnet_conditioning_scale = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_pipeline.stable_diffusion_3_controlnet_image_generation_pipeline_driver.StableDiffusion3ControlNetImageGenerationPipelineDriver.controlnet_conditioning_scale)"><code class="language-python">	*Optional[float]*	The conditioning scale for the ControlNet model. Defaults to None.

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_controlnet_image_generation_pipeline_driver.py

```
@define
class␣
↪StableDiffusion3ControlNetImageGenerationPipelineDriver(StableDiffusion3ImageGenerationPipelineDriv
    """Image generation model driver for Stable Diffusion 3 models with ControlNet.

    For more information, see the HuggingFace documentation for the␣
↪StableDiffusion3ControlNetPipeline:
        https://huggingface.co/docs/diffusers/en/api/pipelines/controlnet_sd3

    Attributes:
        controlnet_model: The ControlNet model to use for image generation.
        controlnet_conditioning_scale: The conditioning scale for the ControlNet␣
```

(continues on next page)

(continued from previous page)

```
↪model. Defaults to None.
    """

    controlnet_model: str = field(kw_only=True)
    controlnet_conditioning_scale: Optional[float] = field(default=None, kw_only=True,
↪ metadata={"serializable": True})

    def prepare_pipeline(self, model: str, device: Optional[str]) -> Any:
        sd3_controlnet_model = import_optional_dependency("diffusers.models.
↪controlnet_sd3").SD3ControlNetModel
        sd3_controlnet_pipeline = import_optional_dependency(
            "diffusers.pipelines.controlnet_sd3.pipeline_stable_diffusion_3_controlnet
↪"
        ).StableDiffusion3ControlNetPipeline

        pipeline_params = {}
        controlnet_pipeline_params = {}
        if self.torch_dtype is not None:
            pipeline_params["torch_dtype"] = self.torch_dtype
            controlnet_pipeline_params["torch_dtype"] = self.torch_dtype

        if self.drop_t5_encoder:
            pipeline_params["text_encoder_3"] = None
            pipeline_params["tokenizer_3"] = None

        # For both Stable Diffusion and ControlNet, models can be provided either
        # as a path to a local file or as a HuggingFace model repo name.
        # We use the from_single_file method if the model is a local file and the
        # from_pretrained method if the model is a local directory or hosted on␣
↪HuggingFace.
        if os.path.isfile(self.controlnet_model):
            pipeline_params["controlnet"] = sd3_controlnet_model.from_single_file(
                self.controlnet_model, **controlnet_pipeline_params
            )
        else:
            pipeline_params["controlnet"] = sd3_controlnet_model.from_pretrained(
                self.controlnet_model, **controlnet_pipeline_params
            )

        if os.path.isfile(model):
            pipeline = sd3_controlnet_pipeline.from_single_file(model, **pipeline_
↪params)
        else:
            pipeline = sd3_controlnet_pipeline.from_pretrained(model, **pipeline_
↪params)

        if self.enable_model_cpu_offload:
            pipeline.enable_model_cpu_offload()

        if device is not None:
            pipeline.to(device)
```

(continues on next page)

(continued from previous page)

```
        return pipeline

    def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:
        if image is None:
            raise ValueError("Input image is required for ControlNet pipelines.")

        return {"control_image": image}

    def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict[str, Any]:
        additional_params = super().make_additional_params(negative_prompts, device)

        del additional_params["height"]
        del additional_params["width"]

        if self.controlnet_conditioning_scale is not None:
            additional_params["controlnet_conditioning_scale"] = self.controlnet_
↪conditioning_scale

        return additional_params
```

- `controlnet_conditioning_scale = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `controlnet_model = field(kw_only=True)` class-attribute instance-attribute

make_additional_params(negative_prompts, device)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_controlnet_image_generation_pipeline_driver.py

```
def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict[str, Any]:
    additional_params = super().make_additional_params(negative_prompts, device)

    del additional_params["height"]
    del additional_params["width"]

    if self.controlnet_conditioning_scale is not None:
        additional_params["controlnet_conditioning_scale"] = self.controlnet_
↪conditioning_scale

    return additional_params
```

make_image_param(image)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_controlnet_image_generation_pipeline_driver.py

```
def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:
    if image is None:
        raise ValueError("Input image is required for ControlNet pipelines.")
```

(continues on next page)

(continued from previous page)

```
    return {"control_image": image}
```

prepare_pipeline(model, device)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_controlnet_image_generation_pipeline_driver.py

```
def prepare_pipeline(self, model: str, device: Optional[str]) -> Any:
    sd3_controlnet_model = import_optional_dependency("diffusers.models.controlnet_sd3
↪").SD3ControlNetModel
    sd3_controlnet_pipeline = import_optional_dependency(
        "diffusers.pipelines.controlnet_sd3.pipeline_stable_diffusion_3_controlnet"
    ).StableDiffusion3ControlNetPipeline

    pipeline_params = {}
    controlnet_pipeline_params = {}
    if self.torch_dtype is not None:
        pipeline_params["torch_dtype"] = self.torch_dtype
        controlnet_pipeline_params["torch_dtype"] = self.torch_dtype

    if self.drop_t5_encoder:
        pipeline_params["text_encoder_3"] = None
        pipeline_params["tokenizer_3"] = None

    # For both Stable Diffusion and ControlNet, models can be provided either
    # as a path to a local file or as a HuggingFace model repo name.
    # We use the from_single_file method if the model is a local file and the
    # from_pretrained method if the model is a local directory or hosted on␣
↪HuggingFace.
    if os.path.isfile(self.controlnet_model):
        pipeline_params["controlnet"] = sd3_controlnet_model.from_single_file(
            self.controlnet_model, **controlnet_pipeline_params
        )
    else:
        pipeline_params["controlnet"] = sd3_controlnet_model.from_pretrained(
            self.controlnet_model, **controlnet_pipeline_params
        )

    if os.path.isfile(model):
        pipeline = sd3_controlnet_pipeline.from_single_file(model, **pipeline_params)
    else:
        pipeline = sd3_controlnet_pipeline.from_pretrained(model, **pipeline_params)

    if self.enable_model_cpu_offload:
        pipeline.enable_model_cpu_offload()

    if device is not None:
        pipeline.to(device)

    return pipeline
```

stable_diffusion_3_image_generation_pipeline_driver

Bases: *BaseDiffusionImageGenerationPipelineDriver* “)

Attributes

Name	Type	Description
<a href="stable_diffusion_3_image_generation_pipeline_driver.mdx#griptape.drivers.image_generation_pipeline.stable_diffusion_3_image_generation_pipeline_driver.StableDiffusion3ImageGenerationPipelineDriver.width" title="width = field(default=1024, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_pipeline.stable_diffusion_3_image_generation_pipeline_driver.StableDiffusion3ImageGenerationPipelineDriver.width) class="language-python">	*int*	The width of the generated image. Defaults to 1024. Must be a multiple of 64.

continues on next page

Table 181 – continued from previous page

Name	Type	Description
<a href="stable_diffusion_3_image_generation_pipeline_driver.mdx#griptape.drivers.image_generation_pipeline.stable_diffusion_3_image_generation_pipeline_driver.StableDiffusion3ImageGenerationPipelineDriver.height" title="height = field(default=1024, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_pipeline.stable_diffusion_3_image_generation_pipeline_driver.StableDiffusion3ImageGenerationPipelineDriver.height" class="language-python">	*int*	The height of the generated image. Defaults to 1024. Must be a multiple of 64.

continues on next page

Table 181 – continued from previous page

Name	Type	Description
<a href="stable_diffusion_3_image_generation_pipeline_driver.mdx#griptape.drivers.image_generation_pipeline.stable_diffusion_3_image_generation_pipeline_driver.StableDiffusion3ImageGenerationPipelineDriver.seed" title="seed = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_pipeline.stable_diffusion_3_image_generation_pipeline_driver.StableDiffusion3ImageGenerationPipelineDriver.seed)" class="language-python">	*Optional[int]*	The random seed to use for image generation. If not provided, a random seed will be used.

continues on next page

Table 181 – continued from previous page

Name	Type	Description
<a href="stable_diffusion_3_image_generation_pipeline_driver.mdx#griptape.drivers.image_generation_pipeline.stable_diffusion_3_image_generation_pipeline_driver.StableDiffusion3ImageGenerationPipelineDriver.guidance_scale" title="guidance_scale = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_pipeline.stable_diffusion_3_image_generation_pipeline_driver.StableDiffusion3ImageGenerationPipelineDriver.guidance_scale)"><code class="language-python">	*Optional[float]*	The strength of the guidance loss. If not provided, the default value will be used.

continues on next page

Table 181 – continued from previous page

Name	Type	Description
<a href="stable_diffusion_3_image_generation_pipeline_driver.mdx#griptape.drivers.image_generation_pipeline.stable_diffusion_3_image_generation_pipeline_driver.StableDiffusion3ImageGenerationPipelineDriver.steps" title="steps = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_pipeline.stable_diffusion_3_image_generation_pipeline_driver.StableDiffusion3ImageGenerationPipelineDriver.steps)" class="language-python">	*Optional[int]*	The number of inference steps to use in image generation. If not provided, the default value will be used.

continues on next page

Table 181 – continued from previous page

Name	Type	Description
<a href="stable_diffusion_3_image_generation_pipeline_driver.mdx#griptape.drivers.image_generation_pipeline.stable_diffusion_3_image_generation_pipeline_driver.StableDiffusion3ImageGenerationPipelineDriver.torch_dtype" title="torch_dtype = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_pipeline.stable_diffusion_3_image_generation_pipeline_driver.StableDiffusion3ImageGenerationPipelineDriver.torch_dtype)" class="language-python">	*Optional[dtype]*	The torch data type to use for image generation. If not provided, the default value will be used.

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_image_generation_pipeline_driver.py

```
@define
class StableDiffusion3ImageGenerationPipelineDriver(BaseDiffusionImageGenerationPipelineDriver):
    """Image generation model driver for Stable Diffusion 3 models.

    For more information, see the HuggingFace documentation for the StableDiffusion3Pipeline:
        https://huggingface.co/docs/diffusers/main/en/api/pipelines/stable_diffusion/stable_diffusion_3

    Attributes:
        width: The width of the generated image. Defaults to 1024. Must be a multiple of 64.
        height: The height of the generated image. Defaults to 1024. Must be a multiple of 64.
        seed: The random seed to use for image generation. If not provided, a random seed will be used.
        guidance_scale: The strength of the guidance loss. If not provided, the default value will be used.
```

(continues on next page)

(continued from previous page)

```
        steps: The number of inference steps to use in image generation. If not␣
↪provided, the default value will be used.
        torch_dtype: The torch data type to use for image generation. If not provided,
↪ the default value will be used.
    """

    width: int = field(default=1024, kw_only=True, metadata={"serializable": True})
    height: int = field(default=1024, kw_only=True, metadata={"serializable": True})
    seed: Optional[int] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    guidance_scale: Optional[float] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    steps: Optional[int] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    torch_dtype: Optional[torch.dtype] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    enable_model_cpu_offload: bool = field(default=False, kw_only=True, metadata={
↪"serializable": True})
    drop_t5_encoder: bool = field(default=False, kw_only=True, metadata={"serializable
↪": True})

    def prepare_pipeline(self, model: str, device: Optional[str]) -> Any:
        sd3_pipeline = import_optional_dependency(
            "diffusers.pipelines.stable_diffusion_3.pipeline_stable_diffusion_3"
        ).StableDiffusion3Pipeline

        pipeline_params = {}
        if self.torch_dtype is not None:
            pipeline_params["torch_dtype"] = self.torch_dtype

        if self.drop_t5_encoder:
            pipeline_params["text_encoder_3"] = None
            pipeline_params["tokenizer_3"] = None

        # A model can be provided either as a path to a local file
        # or as a HuggingFace model repo name.
        if os.path.isfile(model):
            # If the model provided is a local file (not a directory),
            # we load it using the from_single_file method.
            pipeline = sd3_pipeline.from_single_file(model, **pipeline_params)
        else:
            # If the model is a local directory or hosted on HuggingFace,
            # we load it using the from_pretrained method.
            pipeline = sd3_pipeline.from_pretrained(model, **pipeline_params)

        if self.enable_model_cpu_offload:
            pipeline.enable_model_cpu_offload()

        # Move inference to particular device if requested.
        if device is not None:
            pipeline.to(device)
```

(continues on next page)

(continued from previous page)

```
        return pipeline

    def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:
        return None

    def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict[str, Any]:
        torch_generator = import_optional_dependency("torch").Generator

        additional_params = {}
        if negative_prompts:
            additional_params["negative_prompt"] = ", ".join(negative_prompts)

        if self.width is not None:
            additional_params["width"] = self.width

        if self.height is not None:
            additional_params["height"] = self.height

        if self.seed is not None:
            additional_params["generator"] = [torch_generator(device=device).manual_
↪seed(self.seed)]

        if self.guidance_scale is not None:
            additional_params["guidance_scale"] = self.guidance_scale

        if self.steps is not None:
            additional_params["num_inference_steps"] = self.steps

        return additional_params

    @property
    def output_image_dimensions(self) -> tuple[int, int]:
        return self.width, self.height
```

- `drop_t5_encoder = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `enable_model_cpu_offload = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `guidance_scale = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `height = field(default=1024, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output_image_dimensions` property
- `seed = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `steps = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `torch_dtype = field(default=None, kw_only=True, metadata={'serializable': True})`

class-attribute instance-attribute

- `width = field(default=1024, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute

make_additional_params(negative_prompts, device)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_image_generation_pipeline_driver.py

```
def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict[str, Any]:
    torch_generator = import_optional_dependency("torch").Generator

    additional_params = {}
    if negative_prompts:
        additional_params["negative_prompt"] = ", ".join(negative_prompts)

    if self.width is not None:
        additional_params["width"] = self.width

    if self.height is not None:
        additional_params["height"] = self.height

    if self.seed is not None:
        additional_params["generator"] = [torch_generator(device=device).manual_
↪seed(self.seed)]

    if self.guidance_scale is not None:
        additional_params["guidance_scale"] = self.guidance_scale

    if self.steps is not None:
        additional_params["num_inference_steps"] = self.steps

    return additional_params
```

make_image_param(image)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_image_generation_pipeline_driver.py

```
def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:
    return None
```

prepare_pipeline(model, device)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_image_generation_pipeline_driver.py

```
def prepare_pipeline(self, model: str, device: Optional[str]) -> Any:
    sd3_pipeline = import_optional_dependency(
        "diffusers.pipelines.stable_diffusion_3.pipeline_stable_diffusion_3"
    ).StableDiffusion3Pipeline

    pipeline_params = {}
    if self.torch_dtype is not None:
```

(continues on next page)

(continued from previous page)

```
        pipeline_params["torch_dtype"] = self.torch_dtype

    if self.drop_t5_encoder:
        pipeline_params["text_encoder_3"] = None
        pipeline_params["tokenizer_3"] = None

    # A model can be provided either as a path to a local file
    # or as a HuggingFace model repo name.
    if os.path.isfile(model):
        # If the model provided is a local file (not a directory),
        # we load it using the from_single_file method.
        pipeline = sd3_pipeline.from_single_file(model, **pipeline_params)
    else:
        # If the model is a local directory or hosted on HuggingFace,
        # we load it using the from_pretrained method.
        pipeline = sd3_pipeline.from_pretrained(model, **pipeline_params)

    if self.enable_model_cpu_offload:
        pipeline.enable_model_cpu_offload()

    # Move inference to particular device if requested.
    if device is not None:
        pipeline.to(device)

    return pipeline
```

stable_diffusion_3_img_2_img

- `__all__ = ['StableDiffusion3Img2ImgImageGenerationPipelineDriver']` module-attribute

Bases: *StableDiffusion3ImageGenerationPipelineDriver* “)

Attributes

Name	Type	Description
<a href="stable_diffusion_3_img_2_img.mdx#tape.drivers.image_generation_pipeline.stable_diffusion_3_img_2_img.StableDiffusion3Img2ImgImageGenerationPipelineDriver.strengt title="strength = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_pipeline.stable_diffusion_3_img_2_img.StableDiffusion3Img2ImgImageGenerationPipelineDriver.strengt class="language-python">	*Optional[float]*	A value [0.0, 1.0] that determines the strength of the initial image in the output.

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_img_2_img_image_generation_pipeline_driver.py

```
@define
class StableDiffusion3Img2ImgImageGenerationPipelineDriver(StableDiffusion3ImageGenerationPipelineDriver):
    """Image generation model driver for Stable Diffusion 3 model image to image pipelines.

    For more information, see the HuggingFace documentation for the StableDiffusion3Img2ImgPipeline:
        https://github.com/huggingface/diffusers/blob/main/src/diffusers/pipelines/stable_diffusion_3/pipeline_stable_diffusion_3_img2img.py

    Attributes:
        strength: A value [0.0, 1.0] that determines the strength of the initial image in the output.
    """

    strength: Optional[float] = field(default=None, kw_only=True, metadata={"serializable": True})

    def prepare_pipeline(self, model: str, device: Optional[str]) -> Any:
```

(continues on next page)

(continued from previous page)

```
        sd3_img2img_pipeline = import_optional_dependency(
            "diffusers.pipelines.stable_diffusion_3.pipeline_stable_diffusion_3_
↪img2img"
        ).StableDiffusion3Img2ImgPipeline

        pipeline_params = {}
        if self.torch_dtype is not None:
            pipeline_params["torch_dtype"] = self.torch_dtype

        if self.drop_t5_encoder:
            pipeline_params["text_encoder_3"] = None
            pipeline_params["tokenizer_3"] = None

        # A model can be provided either as a path to a local file
        # or as a HuggingFace model repo name.
        if os.path.isfile(model):
            # If the model provided is a local file (not a directory),
            # we load it using the from_single_file method.
            pipeline = sd3_img2img_pipeline.from_single_file(model, **pipeline_params)
        else:
            # If the model is a local directory or hosted on HuggingFace,
            # we load it using the from_pretrained method.
            pipeline = sd3_img2img_pipeline.from_pretrained(model, **pipeline_params)

        if self.enable_model_cpu_offload:
            pipeline.enable_model_cpu_offload()

        # Move inference to particular device if requested.
        if device is not None:
            pipeline.to(device)

        return pipeline

    def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:
        if image is None:
            raise ValueError("Input image is required for image to image pipelines.")

        return {"image": image}

    def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict[str, Any]:
        additional_params = super().make_additional_params(negative_prompts, device)

        # Explicit height and width params are not supported, but
        # are instead inferred from input image.
        del additional_params["height"]
        del additional_params["width"]

        if self.strength is not None:
            additional_params["strength"] = self.strength

        return additional_params
```

- `strength = field(default=None, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute

make_additional_params(negative_prompts, device)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_img_2_img_image_generation_pipeline_driver.py

```
def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict[str, Any]:
    additional_params = super().make_additional_params(negative_prompts, device)

    # Explicit height and width params are not supported, but
    # are instead inferred from input image.
    del additional_params["height"]
    del additional_params["width"]

    if self.strength is not None:
        additional_params["strength"] = self.strength

    return additional_params
```

make_image_param(image)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_img_2_img_image_generation_pipeline_driver.py

```
def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:
    if image is None:
        raise ValueError("Input image is required for image to image pipelines.")

    return {"image": image}
```

prepare_pipeline(model, device)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_img_2_img_image_generation_pipeline_driver.py

```
def prepare_pipeline(self, model: str, device: Optional[str]) -> Any:
    sd3_img2img_pipeline = import_optional_dependency(
        "diffusers.pipelines.stable_diffusion_3.pipeline_stable_diffusion_3_img2img"
    ).StableDiffusion3Img2ImgPipeline

    pipeline_params = {}
    if self.torch_dtype is not None:
        pipeline_params["torch_dtype"] = self.torch_dtype

    if self.drop_t5_encoder:
        pipeline_params["text_encoder_3"] = None
        pipeline_params["tokenizer_3"] = None

    # A model can be provided either as a path to a local file
    # or as a HuggingFace model repo name.
```

(continues on next page)

(continued from previous page)

```
    if os.path.isfile(model):
        # If the model provided is a local file (not a directory),
        # we load it using the from_single_file method.
        pipeline = sd3_img2img_pipeline.from_single_file(model, **pipeline_params)
    else:
        # If the model is a local directory or hosted on HuggingFace,
        # we load it using the from_pretrained method.
        pipeline = sd3_img2img_pipeline.from_pretrained(model, **pipeline_params)

    if self.enable_model_cpu_offload:
        pipeline.enable_model_cpu_offload()

    # Move inference to particular device if requested.
    if device is not None:
        pipeline.to(device)

    return pipeline
```

stable_diffusion_3_img_2_img_image_generation_pipeline_driver

Bases: *StableDiffusion3ImageGenerationPipelineDriver* “)

Attributes

Name	Type	Description
<a href="stable_diffusion_3_img_2_img_image_generation_pipeline_driver.mdx#griptape.drivers.image_generation_pipeline.stable_diffusion_3_img_2_img_image_generation_pipeline_driver.StableDiffusion3Img2ImgImageGenerationPipelineDriver.strength" title="strength = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.image_generation_pipeline.stable_diffusion_3_img_2_img_image_generation_pipeline_driver.StableDiffusion3Img2ImgImageGenerationPipelineDriver.strength" class="language-python">	*Optional[float]*	A value [0.0, 1.0] that determines the strength of the initial image in the output.

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_img_2_img_image_generation_pipeline_driver.py

```
@define
class␣
↪StableDiffusion3Img2ImgImageGenerationPipelineDriver(StableDiffusion3ImageGenerationPipelineDriver):
    """Image generation model driver for Stable Diffusion 3 model image to image␣
↪pipelines.

    For more information, see the HuggingFace documentation for the␣
↪StableDiffusion3Img2ImgPipeline:
        https://github.com/huggingface/diffusers/blob/main/src/diffusers/pipelines/
↪stable_diffusion_3/pipeline_stable_diffusion_3_img2img.py

    Attributes:
        strength: A value [0.0, 1.0] that determines the strength of the initial␣
↪image in the output.
```

(continues on next page)

(continued from previous page)

```
    """

    strength: Optional[float] = field(default=None, kw_only=True, metadata={
↪"serializable": True})

    def prepare_pipeline(self, model: str, device: Optional[str]) -> Any:
        sd3_img2img_pipeline = import_optional_dependency(
            "diffusers.pipelines.stable_diffusion_3.pipeline_stable_diffusion_3_
↪img2img"
        ).StableDiffusion3Img2ImgPipeline

        pipeline_params = {}
        if self.torch_dtype is not None:
            pipeline_params["torch_dtype"] = self.torch_dtype

        if self.drop_t5_encoder:
            pipeline_params["text_encoder_3"] = None
            pipeline_params["tokenizer_3"] = None

        # A model can be provided either as a path to a local file
        # or as a HuggingFace model repo name.
        if os.path.isfile(model):
            # If the model provided is a local file (not a directory),
            # we load it using the from_single_file method.
            pipeline = sd3_img2img_pipeline.from_single_file(model, **pipeline_params)
        else:
            # If the model is a local directory or hosted on HuggingFace,
            # we load it using the from_pretrained method.
            pipeline = sd3_img2img_pipeline.from_pretrained(model, **pipeline_params)

        if self.enable_model_cpu_offload:
            pipeline.enable_model_cpu_offload()

        # Move inference to particular device if requested.
        if device is not None:
            pipeline.to(device)

        return pipeline

    def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:
        if image is None:
            raise ValueError("Input image is required for image to image pipelines.")

        return {"image": image}

    def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict[str, Any]:
        additional_params = super().make_additional_params(negative_prompts, device)

        # Explicit height and width params are not supported, but
        # are instead inferred from input image.
        del additional_params["height"]
```

(continues on next page)

(continued from previous page)

```
        del additional_params["width"]

        if self.strength is not None:
            additional_params["strength"] = self.strength

        return additional_params
```

- `strength = field(default=None, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute

make_additional_params(negative_prompts, device)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_img_2_img_image_generation_pipeline_driver.py

```
def make_additional_params(self, negative_prompts: Optional[list[str]], device:␣
↪Optional[str]) -> dict[str, Any]:
    additional_params = super().make_additional_params(negative_prompts, device)

    # Explicit height and width params are not supported, but
    # are instead inferred from input image.
    del additional_params["height"]
    del additional_params["width"]

    if self.strength is not None:
        additional_params["strength"] = self.strength

    return additional_params
```

make_image_param(image)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_img_2_img_image_generation_pipeline_driver.py

```
def make_image_param(self, image: Optional[Image]) -> Optional[dict[str, Image]]:
    if image is None:
        raise ValueError("Input image is required for image to image pipelines.")

    return {"image": image}
```

prepare_pipeline(model, device)

Source Code in griptape/drivers/image_generation_pipeline/stable_diffusion_3_img_2_img_image_generation_pipeline_driver.py

```
def prepare_pipeline(self, model: str, device: Optional[str]) -> Any:
    sd3_img2img_pipeline = import_optional_dependency(
        "diffusers.pipelines.stable_diffusion_3.pipeline_stable_diffusion_3_img2img"
    ).StableDiffusion3Img2ImgPipeline

    pipeline_params = {}
```

(continues on next page)

(continued from previous page)

```
    if self.torch_dtype is not None:
        pipeline_params["torch_dtype"] = self.torch_dtype

    if self.drop_t5_encoder:
        pipeline_params["text_encoder_3"] = None
        pipeline_params["tokenizer_3"] = None

    # A model can be provided either as a path to a local file
    # or as a HuggingFace model repo name.
    if os.path.isfile(model):
        # If the model provided is a local file (not a directory),
        # we load it using the from_single_file method.
        pipeline = sd3_img2img_pipeline.from_single_file(model, **pipeline_params)
    else:
        # If the model is a local directory or hosted on HuggingFace,
        # we load it using the from_pretrained method.
        pipeline = sd3_img2img_pipeline.from_pretrained(model, **pipeline_params)

    if self.enable_model_cpu_offload:
        pipeline.enable_model_cpu_offload()

    # Move inference to particular device if requested.
    if device is not None:
        pipeline.to(device)

    return pipeline
```

memory

conversation

- `__all__ = ['BaseConversationMemoryDriver']` module-attribute

Bases: *SerializableMixin* “) , ABC

Source Code in griptape/drivers/memory/conversation/base_conversation_memory_driver.py

```
class BaseConversationMemoryDriver(SerializableMixin, ABC):
    @abstractmethod
    def store(self, runs: list[Run], metadata: dict[str, Any]) -> None: ...

    @abstractmethod
    def load(self) -> tuple[list[Run], dict[str, Any]]: ...

    def _to_params_dict(self, runs: list[Run], metadata: dict[str, Any]) -> dict:
        return {"runs": [run.to_dict() for run in runs], "metadata": metadata}

    def _from_params_dict(self, params_dict: dict[str, Any]) -> tuple[list[Run],␣
↪dict[str, Any]]:
        from griptape.memory.structure import Run

        return [Run.from_dict(run) for run in params_dict.get("runs", [])], params_
↪dict.get("metadata", {})
```

_from_params_dict(params_dict)

Source Code in griptape/drivers/memory/conversation/base_conversation_memory_driver.py

```
def _from_params_dict(self, params_dict: dict[str, Any]) -> tuple[list[Run], dict[str,
↪ Any]]:
    from griptape.memory.structure import Run

    return [Run.from_dict(run) for run in params_dict.get("runs", [])], params_dict.
↪get("metadata", {})
```

_to_params_dict(runs, metadata)

Source Code in griptape/drivers/memory/conversation/base_conversation_memory_driver.py

```
def _to_params_dict(self, runs: list[Run], metadata: dict[str, Any]) -> dict:
    return {"runs": [run.to_dict() for run in runs], "metadata": metadata}
```

load()abstractmethod

Source Code in griptape/drivers/memory/conversation/base_conversation_memory_driver.py

```
@abstractmethod
def load(self) -> tuple[list[Run], dict[str, Any]]: ...
```

store(runs, metadata)abstractmethod

Source Code in griptape/drivers/memory/conversation/base_conversation_memory_driver.py

```
@abstractmethod
def store(self, runs: list[Run], metadata: dict[str, Any]) -> None: ...
```

amazon_dynamodb

- `__all__ = ['AmazonDynamoDbConversationMemoryDriver']` module-attribute

Bases: *BaseConversationMemoryDriver* “)

Source Code in griptape/drivers/memory/conversation/amazon_dynamodb_conversation_memory_driver.py

```
@define
class AmazonDynamoDbConversationMemoryDriver(BaseConversationMemoryDriver):
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    table_name: str = field(kw_only=True, metadata={"serializable": True})
    partition_key: str = field(kw_only=True, metadata={"serializable": True})
    value_attribute_key: str = field(kw_only=True, metadata={"serializable": True})
    partition_key_value: str = field(kw_only=True, metadata={"serializable": True})
    sort_key: Optional[str] = field(default=None, metadata={"serializable": True})
    sort_key_value: Optional[str | int] = field(default=None, metadata={"serializable
↪": True})
    _table: Optional[Table] = field(default=None, kw_only=True, alias="table",␣
↪metadata={"serializable": False})
```

(continues on next page)

(continued from previous page)

```
    @lazy_property()
    def table(self) -> Table:
        return self.session.resource("dynamodb").Table(self.table_name)

    def store(self, runs: list[Run], metadata: dict) -> None:
        self.table.update_item(
            Key=self._get_key(),
            UpdateExpression="set #attr = :value",
            ExpressionAttributeNames={"#attr": self.value_attribute_key},
            ExpressionAttributeValues={
                ":value": json.dumps(self._to_params_dict(runs, metadata)),
            },
        )

    def load(self) -> tuple[list[Run], dict[str, Any]]:
        response = self.table.get_item(Key=self._get_key())

        if "Item" in response and self.value_attribute_key in response["Item"]:
            memory_dict = json.loads(str(response["Item"][self.value_attribute_key]))
            return self._from_params_dict(memory_dict)
        return [], {}

    def _get_key(self) -> dict[str, str | int]:
        key: dict[str, str | int] = {self.partition_key: self.partition_key_value}

        if self.sort_key is not None and self.sort_key_value is not None:
            key[self.sort_key] = self.sort_key_value

        return key
```

- `_table = field(default=None, kw_only=True, alias='table', metadata={'serializable': False})` class-attribute instance-attribute
- `partition_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `partition_key_value = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `sort_key = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `sort_key_value = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `table_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `value_attribute_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_get_key()

Source Code in griptape/drivers/memory/conversation/amazon_dynamodb_conversation_memory_driver.py

```
def _get_key(self) -> dict[str, str | int]:
    key: dict[str, str | int] = {self.partition_key: self.partition_key_value}

    if self.sort_key is not None and self.sort_key_value is not None:
        key[self.sort_key] = self.sort_key_value

    return key
```

load()

Source Code in griptape/drivers/memory/conversation/amazon_dynamodb_conversation_memory_driver.py

```
def load(self) -> tuple[list[Run], dict[str, Any]]:
    response = self.table.get_item(Key=self._get_key())

    if "Item" in response and self.value_attribute_key in response["Item"]:
        memory_dict = json.loads(str(response["Item"][self.value_attribute_key]))
        return self._from_params_dict(memory_dict)
    return [], {}
```

store(runs, metadata)

Source Code in griptape/drivers/memory/conversation/amazon_dynamodb_conversation_memory_driver.py

```
def store(self, runs: list[Run], metadata: dict) -> None:
    self.table.update_item(
        Key=self._get_key(),
        UpdateExpression="set #attr = :value",
        ExpressionAttributeNames={"#attr": self.value_attribute_key},
        ExpressionAttributeValues={
            ":value": json.dumps(self._to_params_dict(runs, metadata)),
        },
    )
```

table()

Source Code in griptape/drivers/memory/conversation/amazon_dynamodb_conversation_memory_driver.py

```
@lazy_property()
def table(self) -> Table:
    return self.session.resource("dynamodb").Table(self.table_name)
```

amazon_dynamodb_conversation_memory_driver

Bases: *BaseConversationMemoryDriver* “)

Source Code in griptape/drivers/memory/conversation/amazon_dynamodb_conversation_memory_driver.py

```
@define
class AmazonDynamoDbConversationMemoryDriver(BaseConversationMemoryDriver):
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    table_name: str = field(kw_only=True, metadata={"serializable": True})
    partition_key: str = field(kw_only=True, metadata={"serializable": True})
    value_attribute_key: str = field(kw_only=True, metadata={"serializable": True})
    partition_key_value: str = field(kw_only=True, metadata={"serializable": True})
    sort_key: Optional[str] = field(default=None, metadata={"serializable": True})
    sort_key_value: Optional[str | int] = field(default=None, metadata={"serializable
↪": True})
    _table: Optional[Table] = field(default=None, kw_only=True, alias="table",␣
↪metadata={"serializable": False})

    @lazy_property()
    def table(self) -> Table:
        return self.session.resource("dynamodb").Table(self.table_name)

    def store(self, runs: list[Run], metadata: dict) -> None:
        self.table.update_item(
            Key=self._get_key(),
            UpdateExpression="set #attr = :value",
            ExpressionAttributeNames={"#attr": self.value_attribute_key},
            ExpressionAttributeValues={
                ":value": json.dumps(self._to_params_dict(runs, metadata)),
            },
        )

    def load(self) -> tuple[list[Run], dict[str, Any]]:
        response = self.table.get_item(Key=self._get_key())

        if "Item" in response and self.value_attribute_key in response["Item"]:
            memory_dict = json.loads(str(response["Item"][self.value_attribute_key]))
            return self._from_params_dict(memory_dict)
        return [], {}

    def _get_key(self) -> dict[str, str | int]:
        key: dict[str, str | int] = {self.partition_key: self.partition_key_value}

        if self.sort_key is not None and self.sort_key_value is not None:
            key[self.sort_key] = self.sort_key_value

        return key
```

- `_table = field(default=None, kw_only=True, alias='table', metadata={'serializable': False})` class-attribute instance-attribute
- `partition_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `partition_key_value = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute

- `sort_key = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `sort_key_value = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `table_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `value_attribute_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_get_key()

Source Code in griptape/drivers/memory/conversation/amazon_dynamodb_conversation_memory_driver.py

```
def _get_key(self) -> dict[str, str | int]:
    key: dict[str, str | int] = {self.partition_key: self.partition_key_value}

    if self.sort_key is not None and self.sort_key_value is not None:
        key[self.sort_key] = self.sort_key_value

    return key
```

load()

Source Code in griptape/drivers/memory/conversation/amazon_dynamodb_conversation_memory_driver.py

```
def load(self) -> tuple[list[Run], dict[str, Any]]:
    response = self.table.get_item(Key=self._get_key())

    if "Item" in response and self.value_attribute_key in response["Item"]:
        memory_dict = json.loads(str(response["Item"][self.value_attribute_key]))
        return self._from_params_dict(memory_dict)
    return [], {}
```

store(runs, metadata)

Source Code in griptape/drivers/memory/conversation/amazon_dynamodb_conversation_memory_driver.py

```
def store(self, runs: list[Run], metadata: dict) -> None:
    self.table.update_item(
        Key=self._get_key(),
        UpdateExpression="set #attr = :value",
        ExpressionAttributeNames={"#attr": self.value_attribute_key},
        ExpressionAttributeValues={
            ":value": json.dumps(self._to_params_dict(runs, metadata)),
        },
    )
```

table()

Source Code in griptape/drivers/memory/conversation/amazon_dynamodb_conversation_memory_driver.py

```
@lazy_property()
def table(self) -> Table:
    return self.session.resource("dynamodb").Table(self.table_name)
```

base_conversation_memory_driver

Bases: *SerializableMixin* “) , ABC

Source Code in griptape/drivers/memory/conversation/base_conversation_memory_driver.py

```
class BaseConversationMemoryDriver(SerializableMixin, ABC):
    @abstractmethod
    def store(self, runs: list[Run], metadata: dict[str, Any]) -> None: ...

    @abstractmethod
    def load(self) -> tuple[list[Run], dict[str, Any]]: ...

    def _to_params_dict(self, runs: list[Run], metadata: dict[str, Any]) -> dict:
        return {"runs": [run.to_dict() for run in runs], "metadata": metadata}

    def _from_params_dict(self, params_dict: dict[str, Any]) -> tuple[list[Run],␣
↪dict[str, Any]]:
        from griptape.memory.structure import Run

        return [Run.from_dict(run) for run in params_dict.get("runs", [])], params_
↪dict.get("metadata", {})
```

_from_params_dict(params_dict)

Source Code in griptape/drivers/memory/conversation/base_conversation_memory_driver.py

```
def _from_params_dict(self, params_dict: dict[str, Any]) -> tuple[list[Run], dict[str,
↪ Any]]:
    from griptape.memory.structure import Run

    return [Run.from_dict(run) for run in params_dict.get("runs", [])], params_dict.
↪get("metadata", {})
```

_to_params_dict(runs, metadata)

Source Code in griptape/drivers/memory/conversation/base_conversation_memory_driver.py

```
def _to_params_dict(self, runs: list[Run], metadata: dict[str, Any]) -> dict:
    return {"runs": [run.to_dict() for run in runs], "metadata": metadata}
```

load()abstractmethod

Source Code in griptape/drivers/memory/conversation/base_conversation_memory_driver.py

```
@abstractmethod
def load(self) -> tuple[list[Run], dict[str, Any]]: ...
```

store(runs, metadata)abstractmethod

Source Code in griptape/drivers/memory/conversation/base_conversation_memory_driver.py

```
@abstractmethod
def store(self, runs: list[Run], metadata: dict[str, Any]) -> None: ...
```

griptape_cloud

- `__all__ = ['GriptapeCloudConversationMemoryDriver']` module-attribute

Bases: *BaseConversationMemoryDriver* “)

Attributes

Name	Type	Description
<a href="griptape_cloud.mdx#griptape.drivers.memory.conversation.griptape_cloud.GriptapeCloudConversationMemoryDriver.thread_id" title="thread_id = field(default=None, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.memory.conversation.griptape_cloud.GriptapeCloudConversationMemoryDriver.thread_id)"><code class="language-python">	*Optional[str]*	The ID of the Thread to store the conversation memory in. If not provided, the driver will attempt to retrieve the ID from the environment variable *GT_CLOUD_THREAD_ID*.

continues on next page

Table 184 – continued from previous page

Name	Type	Description
<a href="griptape_cloud.mdx#griptape.drivers.memory.conversation.griptape_cloud.GriptapeCloudConversationMemoryDriver.alias" title="alias = field(default=None, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.memory.conversation.griptape_cloud.GriptapeCloudConversationMemoryDriver.alias)"><code class="language-python">	*Optional[str]*	The alias of the Thread to store the conversation memory in.
<a href="griptape_cloud.mdx#griptape.drivers.memory.conversation.griptape_cloud.GriptapeCloudConversationMemoryDriver.base_url" title="base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_ 'https://cloud.griptape.ai'))) class-attribute instance-attribute (griptape.drivers.memory.conversation.griptape_cloud.GriptapeCloudConversationMemoryDriver.base_url)"><code class="language-python">	*str*	The base URL of the Gen AI Builder API. Defaults to the value of the environment variable *GT_CLOUD_BASE_URL* or *https://cloud.griptape.ai*.

continues on next page

Table 184 – continued from previous page

Name	Type	Description
<a href="griptape_cloud.mdx#griptape.drivers.memory.conversation.griptape_cloud.GriptapeCloudConversationMemoryDriver.api_key" title="api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_K class-attribute instance-attribute (griptape.drivers.memory.conversation.griptape_cloud.GriptapeCloudConversationMemoryDriver.api_key)"><code class="language-python">	*str*	The API key to use for authenticating with the Gen AI Builder API. If not provided, the driver will attempt to retrieve the API key from the environment variable *GT_CLOUD_API_KEY*.

Raises

Source Code in griptape/drivers/memory/conversation/griptape_cloud_conversation_memory_driver.py

```
@define(kw_only=True)
class GriptapeCloudConversationMemoryDriver(BaseConversationMemoryDriver):
    """A driver for storing conversation memory in the Gen AI Builder.

    Attributes:
        thread_id: The ID of the Thread to store the conversation memory in. If not␣
↪provided, the driver will attempt to
            retrieve the ID from the environment variable `GT_CLOUD_THREAD_ID`.
        alias: The alias of the Thread to store the conversation memory in.
        base_url: The base URL of the Gen AI Builder API. Defaults to the value of␣
↪the environment variable
            `GT_CLOUD_BASE_URL` or `https://cloud.griptape.ai`.
        api_key: The API key to use for authenticating with the Gen AI Builder API.␣
↪If not provided, the driver will
            attempt to retrieve the API key from the environment variable `GT_CLOUD_
↪API_KEY`.

    Raises:
        ValueError: If `api_key` is not provided.
    """

    thread_id: Optional[str] = field(
        default=None,
        metadata={"serializable": True},
```

(continues on next page)

(continued from previous page)

```
    )
    alias: Optional[str] = field(
        default=None,
        metadata={"serializable": True},
    )
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]))
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        init=False,
    )
    _thread: Optional[dict] = field(default=None, init=False)

    @api_key.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_api_key(self, _: Attribute, value: Optional[str]) -> str:
        if value is None:
            raise ValueError(f"{self.__class__.__name__} requires an API key")
        return value

    @property
    def thread(self) -> dict:
        """Try to get the Thread by ID, alias, or create a new one."""
        if self._thread is None:
            thread = None
            if self.thread_id is None:
                self.thread_id = os.getenv("GT_CLOUD_THREAD_ID")

            if self.thread_id is not None:
                res = self._call_api("get", f"/threads/{self.thread_id}", raise_for_
↪status=False)
                if res.status_code == 200:
                    thread = res.json()

            # use name as alias to get thread
            if thread is None and self.alias is not None:
                res = self._call_api("get", f"/threads?alias={self.alias}").json()
                if res.get("threads"):
                    thread = res["threads"][0]
                    self.thread_id = thread.get("thread_id")

            # no thread by name or thread_id
            if thread is None:
                data = {"name": uuid.uuid4().hex} if self.alias is None else {"name":␣
↪self.alias, "alias": self.alias}
                thread = self._call_api("post", "/threads", data).json()
                self.thread_id = thread["thread_id"]
                self.alias = thread.get("alias")
```

(continues on next page)

(continued from previous page)

```
            self._thread = thread

        return self._thread  # pyright: ignore[reportReturnType]

    def store(self, runs: list[Run], metadata: dict[str, Any]) -> None:
        # serialize the run artifacts to json strings
        messages = [
            dict_merge(
                {
                    "input": run.input.to_json(),
                    "output": run.output.to_json(),
                    "metadata": {"run_id": run.id},
                },
                run.meta,
            )
            for run in runs
        ]

        body = dict_merge(
            {
                "messages": messages,
            },
            metadata,
        )

        # patch the Thread with the new messages and metadata
        # all old Messages are replaced with the new ones
        thread_id = self.thread["thread_id"] if self.thread_id is None else self.
↪thread_id
        self._call_api("patch", f"/threads/{thread_id}", body)
        self._thread = None

    def load(self) -> tuple[list[Run], dict[str, Any]]:
        from griptape.memory.structure import Run

        thread_id = self.thread["thread_id"] if self.thread_id is None else self.
↪thread_id

        # get the Messages from the Thread
        messages_response = self._call_api("get", f"/threads/{thread_id}/messages").
↪json()

        # retrieve the Thread to get the metadata
        thread_response = self._call_api("get", f"/threads/{thread_id}").json()

        runs = [
            Run(
                **({"id": message["metadata"].pop("run_id", None)} if "run_id" in␣
↪message.get("metadata") else {}),
                meta=message["metadata"],
                input=BaseArtifact.from_json(message["input"]),
                output=BaseArtifact.from_json(message["output"]),
```

(continues on next page)

(continued from previous page)

```
            )
            for message in messages_response.get("messages", [])
        ]

        return runs, thread_response.get("metadata", {})

    def _get_url(self, path: str) -> str:
        path = path.lstrip("/")
        return griptape_cloud_url(self.base_url, f"api/{path}")

    def _call_api(
        self, method: str, path: str, json: Optional[dict] = None, *, raise_for_
↪status: bool = True
    ) -> requests.Response:
        res = requests.request(method, self._get_url(path), json=json, headers=self.
↪headers)
        if raise_for_status:
            res.raise_for_status()
        return res
```

- `_thread = field(default=None, init=False)` class-attribute instance-attribute
- `alias = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']))` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')))` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), init=False)` class-attribute instance-attribute
- `thread` property
- `thread_id = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

_call_api(method, path, json=None, *, raise_for_status=True)

Source Code in griptape/drivers/memory/conversation/griptape_cloud_conversation_memory_driver.py

```
def _call_api(
    self, method: str, path: str, json: Optional[dict] = None, *, raise_for_status:␣
↪bool = True
) -> requests.Response:
    res = requests.request(method, self._get_url(path), json=json, headers=self.
↪headers)
    if raise_for_status:
        res.raise_for_status()
    return res
```

_get_url(path)

Source Code in griptape/drivers/memory/conversation/griptape_cloud_conversation_memory_driver.py

```
def _get_url(self, path: str) -> str:
    path = path.lstrip("/")
    return griptape_cloud_url(self.base_url, f"api/{path}")
```

load()

Source Code in griptape/drivers/memory/conversation/griptape_cloud_conversation_memory_driver.py

```
def load(self) -> tuple[list[Run], dict[str, Any]]:
    from griptape.memory.structure import Run

    thread_id = self.thread["thread_id"] if self.thread_id is None else self.thread_id

    # get the Messages from the Thread
    messages_response = self._call_api("get", f"/threads/{thread_id}/messages").json()

    # retrieve the Thread to get the metadata
    thread_response = self._call_api("get", f"/threads/{thread_id}").json()

    runs = [
        Run(
            **({"id": message["metadata"].pop("run_id", None)} if "run_id" in message.
↪get("metadata") else {}),
            meta=message["metadata"],
            input=BaseArtifact.from_json(message["input"]),
            output=BaseArtifact.from_json(message["output"]),
        )
        for message in messages_response.get("messages", [])
    ]

    return runs, thread_response.get("metadata", {})
```

store(runs, metadata)

Source Code in griptape/drivers/memory/conversation/griptape_cloud_conversation_memory_driver.py

```
def store(self, runs: list[Run], metadata: dict[str, Any]) -> None:
    # serialize the run artifacts to json strings
    messages = [
        dict_merge(
            {
                "input": run.input.to_json(),
                "output": run.output.to_json(),
                "metadata": {"run_id": run.id},
            },
            run.meta,
        )
        for run in runs
    ]
```

(continues on next page)

(continued from previous page)

```
    body = dict_merge(
        {
            "messages": messages,
        },
        metadata,
    )

    # patch the Thread with the new messages and metadata
    # all old Messages are replaced with the new ones
    thread_id = self.thread["thread_id"] if self.thread_id is None else self.thread_id
    self._call_api("patch", f"/threads/{thread_id}", body)
    self._thread = None
```

validate*api_key*(, value)

Source Code in griptape/drivers/memory/conversation/griptape_cloud_conversation_memory_driver.py

```
@api_key.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_api_key(self, _: Attribute, value: Optional[str]) -> str:
    if value is None:
        raise ValueError(f"{self.__class__.__name__} requires an API key")
    return value
```

griptape_cloud_conversation_memory_driver

Bases: *BaseConversationMemoryDriver* “)

Attributes

Name	Type	Description
<a href="griptape_cloud_conversation_memory_driver.mdx#griptape.drivers.memory.conversation.griptape_cloud_conversation_memory_driver.GriptapeCloudConversationMemoryDriver.thread_id" title="thread_id = field(default=None, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.memory.conversation.griptape_cloud_conversation_memory_driver.GriptapeCloudConversationMemoryDriver.thread_id)"><code class="language-python">	*Optional[str]*	The ID of the Thread to store the conversation memory in. If not provided, the driver will attempt to retrieve the ID from the environment variable *GT_CLOUD_THREAD_ID*.

continues on next page

Table 185 – continued from previous page

Name	Type	Description
<a href="griptape_cloud_conversation_memory_driver.mdx#griptape.drivers.memory.conversation.griptape_cloud_conversation_memory_driver.GriptapeCloudConversationMemoryDriver.alias" title="alias = field(default=None, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.memory.conversation.griptape_cloud_conversation_memory_driver.GriptapeCloudConversationMemoryDriver.alias)"><code class="language-python">	*Optional[str]*	The alias of the Thread to store the conversation memory in.

continues on next page

Table 185 – continued from previous page

Name	Type	Description
<a href="griptape_cloud_conversation_memory_driver.mdx#griptape.drivers.memory.conversation.griptape_cloud_conversation_memory_driver.GriptapeCloudConversationMemoryDriver.base_url" title="base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_ 'https://cloud.griptape.ai'))) class-attribute instance-attribute (griptape.drivers.memory.conversation.griptape_cloud_conversation_memory_driver.GriptapeCloudConversationMemoryDriver.base_url)"><code class="language-python">	*str*	The base URL of the Gen AI Builder API. Defaults to the value of the environment variable *GT_CLOUD_BASE_URL* or *https://cloud.griptape.ai*.

continues on next page

Table 185 – continued from previous page

Name	Type	Description
<a href="griptape_cloud_conversation_memory_driver.mdx#griptape.drivers.memory.conversation.griptape_cloud_conversation_memory_driver.GriptapeCloudConversationMemoryDriver.api_key" title="api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_K class-attribute instance-attribute (griptape.drivers.memory.conversation.griptape_cloud_conversation_memory_driver.GriptapeCloudConversationMemoryDriver.api_key)"><code class="language-python">	*str*	The API key to use for authenticating with the Gen AI Builder API. If not provided, the driver will attempt to retrieve the API key from the environment variable *GT_CLOUD_API_KEY*.

Raises

Source Code in griptape/drivers/memory/conversation/griptape_cloud_conversation_memory_driver.py

```
@define(kw_only=True)
class GriptapeCloudConversationMemoryDriver(BaseConversationMemoryDriver):
    """A driver for storing conversation memory in the Gen AI Builder.

    Attributes:
        thread_id: The ID of the Thread to store the conversation memory in. If not␣
↪provided, the driver will attempt to
            retrieve the ID from the environment variable `GT_CLOUD_THREAD_ID`.
        alias: The alias of the Thread to store the conversation memory in.
        base_url: The base URL of the Gen AI Builder API. Defaults to the value of␣
↪the environment variable
            `GT_CLOUD_BASE_URL` or `https://cloud.griptape.ai`.
        api_key: The API key to use for authenticating with the Gen AI Builder API.␣
↪If not provided, the driver will
            attempt to retrieve the API key from the environment variable `GT_CLOUD_
↪API_KEY`.

    Raises:
```

(continues on next page)

(continued from previous page)

```
        ValueError: If `api_key` is not provided.
    """

    thread_id: Optional[str] = field(
        default=None,
        metadata={"serializable": True},
    )
    alias: Optional[str] = field(
        default=None,
        metadata={"serializable": True},
    )
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]))
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        init=False,
    )
    _thread: Optional[dict] = field(default=None, init=False)

    @api_key.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_api_key(self, _: Attribute, value: Optional[str]) -> str:
        if value is None:
            raise ValueError(f"{self.__class__.__name__} requires an API key")
        return value

    @property
    def thread(self) -> dict:
        """Try to get the Thread by ID, alias, or create a new one."""
        if self._thread is None:
            thread = None
            if self.thread_id is None:
                self.thread_id = os.getenv("GT_CLOUD_THREAD_ID")

            if self.thread_id is not None:
                res = self._call_api("get", f"/threads/{self.thread_id}", raise_for_
↪status=False)
                if res.status_code == 200:
                    thread = res.json()

            # use name as alias to get thread
            if thread is None and self.alias is not None:
                res = self._call_api("get", f"/threads?alias={self.alias}").json()
                if res.get("threads"):
                    thread = res["threads"][0]
                    self.thread_id = thread.get("thread_id")

            # no thread by name or thread_id
            if thread is None:
```

(continues on next page)

(continued from previous page)

```
                data = {"name": uuid.uuid4().hex} if self.alias is None else {"name":␣
↪self.alias, "alias": self.alias}
                thread = self._call_api("post", "/threads", data).json()
                self.thread_id = thread["thread_id"]
                self.alias = thread.get("alias")

            self._thread = thread

        return self._thread  # pyright: ignore[reportReturnType]

    def store(self, runs: list[Run], metadata: dict[str, Any]) -> None:
        # serialize the run artifacts to json strings
        messages = [
            dict_merge(
                {
                    "input": run.input.to_json(),
                    "output": run.output.to_json(),
                    "metadata": {"run_id": run.id},
                },
                run.meta,
            )
            for run in runs
        ]

        body = dict_merge(
            {
                "messages": messages,
            },
            metadata,
        )

        # patch the Thread with the new messages and metadata
        # all old Messages are replaced with the new ones
        thread_id = self.thread["thread_id"] if self.thread_id is None else self.
↪thread_id
        self._call_api("patch", f"/threads/{thread_id}", body)
        self._thread = None

    def load(self) -> tuple[list[Run], dict[str, Any]]:
        from griptape.memory.structure import Run

        thread_id = self.thread["thread_id"] if self.thread_id is None else self.
↪thread_id

        # get the Messages from the Thread
        messages_response = self._call_api("get", f"/threads/{thread_id}/messages").
↪json()

        # retrieve the Thread to get the metadata
        thread_response = self._call_api("get", f"/threads/{thread_id}").json()

        runs = [
```

(continues on next page)

(continued from previous page)

```
            Run(
                **({"id": message["metadata"].pop("run_id", None)} if "run_id" in␣
↪message.get("metadata") else {}),
                meta=message["metadata"],
                input=BaseArtifact.from_json(message["input"]),
                output=BaseArtifact.from_json(message["output"]),
            )
            for message in messages_response.get("messages", [])
        ]

        return runs, thread_response.get("metadata", {})

    def _get_url(self, path: str) -> str:
        path = path.lstrip("/")
        return griptape_cloud_url(self.base_url, f"api/{path}")

    def _call_api(
        self, method: str, path: str, json: Optional[dict] = None, *, raise_for_
↪status: bool = True
    ) -> requests.Response:
        res = requests.request(method, self._get_url(path), json=json, headers=self.
↪headers)
        if raise_for_status:
            res.raise_for_status()
        return res
```

- `_thread = field(default=None, init=False)` class-attribute instance-attribute
- `alias = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']))` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')))` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), init=False)` class-attribute instance-attribute
- `thread` property
- `thread_id = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

_call_api(method, path, json=None, *, raise_for_status=True)

Source Code in griptape/drivers/memory/conversation/griptape_cloud_conversation_memory_driver.py

```
def _call_api(
    self, method: str, path: str, json: Optional[dict] = None, *, raise_for_status:␣
↪bool = True
) -> requests.Response:
    res = requests.request(method, self._get_url(path), json=json, headers=self.
↪headers)
```

(continues on next page)

(continued from previous page)

```
    if raise_for_status:
        res.raise_for_status()
    return res
```

_get_url(path)

Source Code in griptape/drivers/memory/conversation/griptape_cloud_conversation_memory_driver.py

```
def _get_url(self, path: str) -> str:
    path = path.lstrip("/")
    return griptape_cloud_url(self.base_url, f"api/{path}")
```

load()

Source Code in griptape/drivers/memory/conversation/griptape_cloud_conversation_memory_driver.py

```
def load(self) -> tuple[list[Run], dict[str, Any]]:
    from griptape.memory.structure import Run

    thread_id = self.thread["thread_id"] if self.thread_id is None else self.thread_id

    # get the Messages from the Thread
    messages_response = self._call_api("get", f"/threads/{thread_id}/messages").json()

    # retrieve the Thread to get the metadata
    thread_response = self._call_api("get", f"/threads/{thread_id}").json()

    runs = [
        Run(
            **({"id": message["metadata"].pop("run_id", None)} if "run_id" in message.
↪get("metadata") else {}),
            meta=message["metadata"],
            input=BaseArtifact.from_json(message["input"]),
            output=BaseArtifact.from_json(message["output"]),
        )
        for message in messages_response.get("messages", [])
    ]

    return runs, thread_response.get("metadata", {})
```

store(runs, metadata)

Source Code in griptape/drivers/memory/conversation/griptape_cloud_conversation_memory_driver.py

```
def store(self, runs: list[Run], metadata: dict[str, Any]) -> None:
    # serialize the run artifacts to json strings
    messages = [
        dict_merge(
            {
                "input": run.input.to_json(),
                "output": run.output.to_json(),
```

(continues on next page)

(continued from previous page)

```
                "metadata": {"run_id": run.id},
            },
            run.meta,
        )
        for run in runs
    ]

    body = dict_merge(
        {
            "messages": messages,
        },
        metadata,
    )

    # patch the Thread with the new messages and metadata
    # all old Messages are replaced with the new ones
    thread_id = self.thread["thread_id"] if self.thread_id is None else self.thread_id
    self._call_api("patch", f"/threads/{thread_id}", body)
    self._thread = None
```

validate*api_key(*, value)

Source Code in griptape/drivers/memory/conversation/griptape_cloud_conversation_memory_driver.py

```
@api_key.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_api_key(self, _: Attribute, value: Optional[str]) -> str:
    if value is None:
        raise ValueError(f"{self.__class__.__name__} requires an API key")
    return value
```

local

- `__all__ = ['LocalConversationMemoryDriver']` module-attribute

Bases: *BaseConversationMemoryDriver* “)

Source Code in griptape/drivers/memory/conversation/local_conversation_memory_driver.py

```
@define(kw_only=True)
class LocalConversationMemoryDriver(BaseConversationMemoryDriver):
    persist_file: Optional[str] = field(default=None, metadata={"serializable": True})

    def store(self, runs: list[Run], metadata: dict[str, Any]) -> None:
        if self.persist_file is not None:
            Path(self.persist_file).write_text(json.dumps(self._to_params_dict(runs,␣
↪metadata)))

    def load(self) -> tuple[list[Run], dict[str, Any]]:
        if (
            self.persist_file is not None
            and os.path.exists(self.persist_file)
            and (loaded_str := Path(self.persist_file).read_text()) is not None
```

(continues on next page)

(continued from previous page)

```
        ):
            try:
                return self._from_params_dict(json.loads(loaded_str))
            except Exception as e:
                raise ValueError(f"Unable to load data from {self.persist_file}")␣
↪from e

        return [], {}
```

- `persist_file = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

load()

Source Code in griptape/drivers/memory/conversation/local_conversation_memory_driver.py

```
def load(self) -> tuple[list[Run], dict[str, Any]]:
    if (
        self.persist_file is not None
        and os.path.exists(self.persist_file)
        and (loaded_str := Path(self.persist_file).read_text()) is not None
    ):
        try:
            return self._from_params_dict(json.loads(loaded_str))
        except Exception as e:
            raise ValueError(f"Unable to load data from {self.persist_file}") from e

    return [], {}
```

store(runs, metadata)

Source Code in griptape/drivers/memory/conversation/local_conversation_memory_driver.py

```
def store(self, runs: list[Run], metadata: dict[str, Any]) -> None:
    if self.persist_file is not None:
        Path(self.persist_file).write_text(json.dumps(self._to_params_dict(runs,␣
↪metadata)))
```

local_conversation_memory_driver

Bases: *BaseConversationMemoryDriver* “)

Source Code in griptape/drivers/memory/conversation/local_conversation_memory_driver.py

```
@define(kw_only=True)
class LocalConversationMemoryDriver(BaseConversationMemoryDriver):
    persist_file: Optional[str] = field(default=None, metadata={"serializable": True})

    def store(self, runs: list[Run], metadata: dict[str, Any]) -> None:
        if self.persist_file is not None:
            Path(self.persist_file).write_text(json.dumps(self._to_params_dict(runs,␣
↪metadata)))
```

(continues on next page)

(continued from previous page)

```
    def load(self) -> tuple[list[Run], dict[str, Any]]:
        if (
            self.persist_file is not None
            and os.path.exists(self.persist_file)
            and (loaded_str := Path(self.persist_file).read_text()) is not None
        ):
            try:
                return self._from_params_dict(json.loads(loaded_str))
            except Exception as e:
                raise ValueError(f"Unable to load data from {self.persist_file}")␣
↪from e

        return [], {}
```

- `persist_file = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

load()

Source Code in griptape/drivers/memory/conversation/local_conversation_memory_driver.py

```
def load(self) -> tuple[list[Run], dict[str, Any]]:
    if (
        self.persist_file is not None
        and os.path.exists(self.persist_file)
        and (loaded_str := Path(self.persist_file).read_text()) is not None
    ):
        try:
            return self._from_params_dict(json.loads(loaded_str))
        except Exception as e:
            raise ValueError(f"Unable to load data from {self.persist_file}") from e

    return [], {}
```

store(runs, metadata)

Source Code in griptape/drivers/memory/conversation/local_conversation_memory_driver.py

```
def store(self, runs: list[Run], metadata: dict[str, Any]) -> None:
    if self.persist_file is not None:
        Path(self.persist_file).write_text(json.dumps(self._to_params_dict(runs,␣
↪metadata)))
```

redis

- `__all__ = ['RedisConversationMemoryDriver']` module-attribute

Bases: *BaseConversationMemoryDriver* “)

Attributes

Name	Type	Description
<a href="redis.mdx#griptape.drivers.memory.conversation.redis.RedisConversationMemoryDriver.host" title="host = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.memory.conversation.redis.RedisConversationMemoryDriver.host)"><code class="language-python">	*str*	The host of the Redis instance.
<a href="redis.mdx#griptape.drivers.memory.conversation.redis.RedisConversationMemoryDriver.port" title="port = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.memory.conversation.redis.RedisConversationMemoryDriver.port)"><code class="language-python">	*int*	The port of the Redis instance.

continues on next page

Table 186 – continued from previous page

Name	Type	Description
<a href="redis.mdx#griptape.drivers.memory.conversation.redis.RedisConversationMemoryDriver.db" title="db = field(kw_only=True, default=0, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.memory.conversation.redis.RedisConversationMemoryDriver.db)"><code class="language-python">	*int*	The database of the Redis instance.
<a href="redis.mdx#griptape.drivers.memory.conversation.redis.RedisConversationMemoryDriver.username" title="username = field(kw_only=True, default='default', metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.memory.conversation.redis.RedisConversationMemoryDriver.username)"><code class="language-python">	*str*	The username of the Redis instance.

continues on next page

Table 186 – continued from previous page

Name	Type	Description
<a href="redis.mdx#griptape.drivers.memory.conversation.redis.RedisConversationMemoryDriver.password" title="password = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.memory.conversation.redis.RedisConversationMemoryDriver.password)"><code class="language-python">	*Optional[str]*	The password of the Redis instance.
<a href="redis.mdx#griptape.drivers.memory.conversation.redis.RedisConversationMemoryDriver.index" title="index = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.memory.conversation.redis.RedisConversationMemoryDriver.index)"><code class="language-python">	*str*	The name of the index to use.

continues on next page

Table 186 – continued from previous page

Name	Type	Description
<a href="redis.mdx#griptape.drivers.memory.conversation.redis.RedisConversationMemoryDriver.conversation_id" title="conversation_id = field(kw_only=True, default=uuid.uuid4().hex) class-attribute instance-attribute (griptape.drivers.memory.conversation.redis.RedisConversationMemoryDriver.conversation_id)"><code class="language-python">	*str*	The id of the conversation.

Source Code in griptape/drivers/memory/conversation/redis_conversation_memory_driver.py

```
@define
class RedisConversationMemoryDriver(BaseConversationMemoryDriver):
    """A Conversation Memory Driver for Redis.

    This driver interfaces with a Redis instance and utilizes the Redis hashes and RediSearch module to store,
    retrieve, and query conversations in a structured manner.
    Proper setup of the Redis instance and RediSearch is necessary for the driver to function correctly.

    Attributes:
        host: The host of the Redis instance.
        port: The port of the Redis instance.
        db: The database of the Redis instance.
        username: The username of the Redis instance.
        password: The password of the Redis instance.
        index: The name of the index to use.
        conversation_id: The id of the conversation.
    """

    host: str = field(kw_only=True, metadata={"serializable": True})
    username: str = field(kw_only=True, default="default", metadata={"serializable": False})
    port: int = field(kw_only=True, metadata={"serializable": True})
    db: int = field(kw_only=True, default=0, metadata={"serializable": True})
    password: Optional[str] = field(default=None, kw_only=True, metadata={"serializable": False})
    index: str = field(kw_only=True, metadata={"serializable": True})
```

(continues on next page)

(continued from previous page)

```
    conversation_id: str = field(kw_only=True, default=uuid.uuid4().hex)

    client: Redis = field(
        default=Factory(
            lambda self: import_optional_dependency("redis").Redis(
                host=self.host,
                port=self.port,
                db=self.db,
                username=self.username,
                password=self.password,
                decode_responses=False,
            ),
            takes_self=True,
        ),
    )

    def store(self, runs: list[Run], metadata: dict[str, Any]) -> None:
        self.client.hset(self.index, self.conversation_id, json.dumps(self._to_params_
↪dict(runs, metadata)))

    def load(self) -> tuple[list[Run], dict[str, Any]]:
        memory_json = self.client.hget(self.index, self.conversation_id)
        if memory_json is not None:
            return self._from_params_dict(json.loads(memory_json))  # pyright:␣
↪ignore[reportArgumentType] https://github.com/redis/redis-py/issues/2399
        return [], {}
```

- `client = field(default=Factory(lambda self: import_optional_dependency('redis').Redis(host=self.host, port=self.port, db=self.db, username=self.username, password=self.password, decode_responses=False), takes_self=True))` class-attribute instance-attribute
- `conversation_id = field(kw_only=True, default=uuid.uuid4().hex)` class-attribute instance-attribute
- `db = field(kw_only=True, default=0, metadata={'serializable': True})` class-attribute instance-attribute
- `host = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `index = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `password = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `port = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `username = field(kw_only=True, default='default', metadata={'serializable': False})` class-attribute instance-attribute

load()

Source Code in griptape/drivers/memory/conversation/redis_conversation_memory_driver.py

```
def load(self) -> tuple[list[Run], dict[str, Any]]:
    memory_json = self.client.hget(self.index, self.conversation_id)
    if memory_json is not None:
        return self._from_params_dict(json.loads(memory_json))  # pyright:␣
↪ignore[reportArgumentType] https://github.com/redis/redis-py/issues/2399
    return [], {}
```

store(runs, metadata)

Source Code in griptape/drivers/memory/conversation/redis_conversation_memory_driver.py

```
def store(self, runs: list[Run], metadata: dict[str, Any]) -> None:
    self.client.hset(self.index, self.conversation_id, json.dumps(self._to_params_
↪dict(runs, metadata)))
```

redis_conversation_memory_driver

Bases: *BaseConversationMemoryDriver* “)

Attributes

Name	Type	Description
<a href="redis_conversation_memory_driver.mdx#griptape.drivers.memory.conversation.redis_conversation_memory_driver.RedisConversationMemoryDriver.host" title="host = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.memory.conversation.redis_conversation_memory_driver.RedisConversationMemoryDriver.host)"><code class="language-python">	*str*	The host of the Redis instance.

continues on next page

Table 187 – continued from previous page

Name	Type	Description
<a href="redis_conversation_memory_driver.mdx#griptape.drivers.memory.conversation.redis_conversation_memory_driver.RedisConversationMemoryDriver.port" title="port = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.memory.conversation.redis_conversation_memory_driver.RedisConversationMemoryDriver.port)"><code class="language-python">	*int*	The port of the Redis instance.
<a href="redis_conversation_memory_driver.mdx#griptape.drivers.memory.conversation.redis_conversation_memory_driver.RedisConversationMemoryDriver.db" title="db = field(kw_only=True, default=0, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.memory.conversation.redis_conversation_memory_driver.RedisConversationMemoryDriver.db)"><code class="language-python">	*int*	The database of the Redis instance.

continues on next page

Table 187 – continued from previous page

Name	Type	Description
<a href="redis_conversation_memory_driver.mdx#griptape.drivers.memory.conversation.redis_conversation_memory_driver.RedisConversationMemoryDriver.username" title="username = field(kw_only=True, default='default', metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.memory.conversation.redis_conversation_memory_driver.RedisConversationMemoryDriver.username)"><code class="language-python">	*str*	The username of the Redis instance.
<a href="redis_conversation_memory_driver.mdx#griptape.drivers.memory.conversation.redis_conversation_memory_driver.RedisConversationMemoryDriver.password" title="password = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.memory.conversation.redis_conversation_memory_driver.RedisConversationMemoryDriver.password)"><code class="language-python">	*Optional[str]*	The password of the Redis instance.

continues on next page

Table 187 – continued from previous page

Name	Type	Description
<a href="redis_conversation_memory_driver.mdx#griptape.drivers.memory.conversation.redis_conversation_memory_driver.RedisConversationMemoryDriver.index" title="index = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.memory.conversation.redis_conversation_memory_driver.RedisConversationMemoryDriver.index)"><code class="language-python">	*str*	The name of the index to use.
<a href="redis_conversation_memory_driver.mdx#griptape.drivers.memory.conversation.redis_conversation_memory_driver.RedisConversationMemoryDriver.conversation_id" title="conversation_id = field(kw_only=True, default=uuid.uuid4().hex) class-attribute instance-attribute (griptape.drivers.memory.conversation.redis_conversation_memory_driver.RedisConversationMemoryDriver.conversation_id)"><code class="language-python">	*str*	The id of the conversation.

Source Code in griptape/drivers/memory/conversation/redis_conversation_memory_driver.py

```
@define
class RedisConversationMemoryDriver(BaseConversationMemoryDriver):
```

(continues on next page)

(continued from previous page)

```
    """A Conversation Memory Driver for Redis.

    This driver interfaces with a Redis instance and utilizes the Redis hashes and␣
↪RediSearch module to store,
    retrieve, and query conversations in a structured manner.
    Proper setup of the Redis instance and RediSearch is necessary for the driver to␣
↪function correctly.

    Attributes:
        host: The host of the Redis instance.
        port: The port of the Redis instance.
        db: The database of the Redis instance.
        username: The username of the Redis instance.
        password: The password of the Redis instance.
        index: The name of the index to use.
        conversation_id: The id of the conversation.
    """

    host: str = field(kw_only=True, metadata={"serializable": True})
    username: str = field(kw_only=True, default="default", metadata={"serializable":␣
↪False})
    port: int = field(kw_only=True, metadata={"serializable": True})
    db: int = field(kw_only=True, default=0, metadata={"serializable": True})
    password: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": False})
    index: str = field(kw_only=True, metadata={"serializable": True})
    conversation_id: str = field(kw_only=True, default=uuid.uuid4().hex)

    client: Redis = field(
        default=Factory(
            lambda self: import_optional_dependency("redis").Redis(
                host=self.host,
                port=self.port,
                db=self.db,
                username=self.username,
                password=self.password,
                decode_responses=False,
            ),
            takes_self=True,
        ),
    )

    def store(self, runs: list[Run], metadata: dict[str, Any]) -> None:
        self.client.hset(self.index, self.conversation_id, json.dumps(self._to_params_
↪dict(runs, metadata)))

    def load(self) -> tuple[list[Run], dict[str, Any]]:
        memory_json = self.client.hget(self.index, self.conversation_id)
        if memory_json is not None:
            return self._from_params_dict(json.loads(memory_json))  # pyright:␣
↪ignore[reportArgumentType] https://github.com/redis/redis-py/issues/2399
        return [], {}
```

- `client = field(default=Factory(lambda self: import_optional_dependency('redis').Redis(host=self.host, port=self.port, db=self.db, username=self.username, password=self.password, decode_responses=False), takes_self=True))` class-attribute instance-attribute
- `conversation_id = field(kw_only=True, default=uuid.uuid4().hex)` class-attribute instance-attribute
- `db = field(kw_only=True, default=0, metadata={'serializable': True})` class-attribute instance-attribute
- `host = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `index = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `password = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `port = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `username = field(kw_only=True, default='default', metadata={'serializable': False})` class-attribute instance-attribute

load()

Source Code in griptape/drivers/memory/conversation/redis_conversation_memory_driver.py

```
def load(self) -> tuple[list[Run], dict[str, Any]]:
    memory_json = self.client.hget(self.index, self.conversation_id)
    if memory_json is not None:
        return self._from_params_dict(json.loads(memory_json))  # pyright:␣
↪ignore[reportArgumentType] https://github.com/redis/redis-py/issues/2399
    return [], {}
```

store(runs, metadata)

Source Code in griptape/drivers/memory/conversation/redis_conversation_memory_driver.py

```
def store(self, runs: list[Run], metadata: dict[str, Any]) -> None:
    self.client.hset(self.index, self.conversation_id, json.dumps(self._to_params_
↪dict(runs, metadata)))
```

observability

- `__all__ = ['BaseObservabilityDriver']` module-attribute

Bases:

ABC

Source Code in griptape/drivers/observability/base_observability_driver.py

```
@define
class BaseObservabilityDriver(ABC):
    def __enter__(self) -> None:  # noqa: B027
```

(continues on next page)

(continued from previous page)

```
        pass

    def __exit__(
        self,
        exc_type: Optional[type[BaseException]],
        exc_value: Optional[BaseException],
        exc_traceback: Optional[TracebackType],
    ) -> bool:
        return False

    @abstractmethod
    def observe(self, call: Observable.Call) -> Any: ...

    @abstractmethod
    def get_span_id(self) -> Optional[str]: ...
```

enter()

Source Code in griptape/drivers/observability/base_observability_driver.py

```
def __enter__(self) -> None:  # noqa: B027
    pass
```

exit(exc_type, exc_value, exc_traceback)

Source Code in griptape/drivers/observability/base_observability_driver.py

```
def __exit__(
    self,
    exc_type: Optional[type[BaseException]],
    exc_value: Optional[BaseException],
    exc_traceback: Optional[TracebackType],
) -> bool:
    return False
```

get_span_id()abstractmethod

Source Code in griptape/drivers/observability/base_observability_driver.py

```
@abstractmethod
def get_span_id(self) -> Optional[str]: ...
```

observe(call)abstractmethod

Source Code in griptape/drivers/observability/base_observability_driver.py

```
@abstractmethod
def observe(self, call: Observable.Call) -> Any: ...
```

base_observability_driver

Bases:

ABC

Source Code in griptape/drivers/observability/base_observability_driver.py

```
@define
class BaseObservabilityDriver(ABC):
    def __enter__(self) -> None:  # noqa: B027
        pass

    def __exit__(
        self,
        exc_type: Optional[type[BaseException]],
        exc_value: Optional[BaseException],
        exc_traceback: Optional[TracebackType],
    ) -> bool:
        return False

    @abstractmethod
    def observe(self, call: Observable.Call) -> Any: ...

    @abstractmethod
    def get_span_id(self) -> Optional[str]: ...
```

enter()

Source Code in griptape/drivers/observability/base_observability_driver.py

```
def __enter__(self) -> None:  # noqa: B027
    pass
```

exit(exc_type, exc_value, exc_traceback)

Source Code in griptape/drivers/observability/base_observability_driver.py

```
def __exit__(
    self,
    exc_type: Optional[type[BaseException]],
    exc_value: Optional[BaseException],
    exc_traceback: Optional[TracebackType],
) -> bool:
    return False
```

get_span_id()abstractmethod

Source Code in griptape/drivers/observability/base_observability_driver.py

```
@abstractmethod
def get_span_id(self) -> Optional[str]: ...
```

observe(call)abstractmethod

Source Code in griptape/drivers/observability/base_observability_driver.py

```
@abstractmethod
def observe(self, call: Observable.Call) -> Any: ...
```

datadog

- `__all__ = ['DatadogObservabilityDriver']` module-attribute

Bases: *OpenTelemetryObservabilityDriver* “)

Source Code in griptape/drivers/observability/datadog_observability_driver.py

```
@define
class DatadogObservabilityDriver(OpenTelemetryObservabilityDriver):
    datadog_agent_endpoint: str = field(
        default=Factory(lambda: os.getenv("DD_AGENT_ENDPOINT", "http://localhost:4318
↪")), kw_only=True
    )
    span_processor: SpanProcessor = field(
        default=Factory(
            lambda self: import_optional_dependency("opentelemetry.sdk.trace.export").
↪BatchSpanProcessor(
                import_optional_dependency("opentelemetry.exporter.otlp.proto.http.
↪trace_exporter").OTLPSpanExporter(
                    endpoint=f"{self.datadog_agent_endpoint}/v1/traces"
                )
            ),
            takes_self=True,
        ),
        kw_only=True,
    )
```

- `datadog_agent_endpoint = field(default=Factory(lambda: os.getenv('DD_AGENT_ENDPOINT', 'http://localhost:4318')), kw_only=True)` class-attribute instance-attribute
- `span_processor = field(default=Factory(lambda self: import_optional_dependency('opentelemetry.sdk.trace.export').BatchSpanProcessor(import_optional_dependency('opentelemetry.exporter.otlp.proto.http.trace_exporter').OTLPSpanExporter(endpoint=f'{self.datadog_agent_endpoint}/v1/traces')), takes_self=True), kw_only=True)` class-attribute instance-attribute

datadog_observability_driver

Bases: *OpenTelemetryObservabilityDriver* “)

Source Code in griptape/drivers/observability/datadog_observability_driver.py

```
@define
class DatadogObservabilityDriver(OpenTelemetryObservabilityDriver):
    datadog_agent_endpoint: str = field(
        default=Factory(lambda: os.getenv("DD_AGENT_ENDPOINT", "http://localhost:4318
```

(continues on next page)

(continued from previous page)

```
↪")), kw_only=True
    )
    span_processor: SpanProcessor = field(
        default=Factory(
            lambda self: import_optional_dependency("opentelemetry.sdk.trace.export").
↪BatchSpanProcessor(
                import_optional_dependency("opentelemetry.exporter.otlp.proto.http.
↪trace_exporter").OTLPSpanExporter(
                    endpoint=f"{self.datadog_agent_endpoint}/v1/traces"
                )
            ),
            takes_self=True,
        ),
        kw_only=True,
    )
```

- `datadog_agent_endpoint = field(default=Factory(lambda: os.getenv('DD_AGENT_ENDPOINT', 'http://localhost:4318')), kw_only=True)` class-attribute instance-attribute
- `span_processor = field(default=Factory(lambda self: import_optional_dependency('opentelemetry.sdk.trace.export').BatchSpanProcessor(import_optional_dependency('opentelemetry.exporter.otlp.proto.http.trace_exporter').OTLPSpanExporter(endpoint=f'{self.datadog_agent_endpoint}/v1/traces')), takes_self=True), kw_only=True)` class-attribute instance-attribute

griptape_cloud

- `__all__ = ['GriptapeCloudObservabilityDriver']` module-attribute

Bases: *OpenTelemetryObservabilityDriver* “)

Source Code in griptape/drivers/observability/griptape_cloud_observability_driver.py

```
@define
class GriptapeCloudObservabilityDriver(OpenTelemetryObservabilityDriver):
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")), kw_only=True
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]), kw_
↪only=True)
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True), kw_only=True
    )
    structure_run_id: Optional[str] = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_STRUCTURE_RUN_ID")), kw_only=True
    )
    span_processor: SpanProcessor = field(
        default=Factory(
            lambda self: import_optional_dependency("opentelemetry.sdk.trace.export").
↪BatchSpanProcessor(
                GriptapeCloudObservabilityDriver.build_span_exporter(
```

(continues on next page)

(continued from previous page)

```
                    base_url=self.base_url,
                    api_key=self.api_key,
                    headers=self.headers,
                    structure_run_id=self.structure_run_id,
                )
            ),
            takes_self=True,
        ),
        kw_only=True,
    )

    @structure_run_id.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_run_id(self, _: Attribute, structure_run_id: Optional[str]) -> None:
        if structure_run_id is None:
            raise ValueError(
                "structure_run_id must be set either in the constructor or as an␣
↪environment variable (GT_CLOUD_STRUCTURE_RUN_ID)."
            )

    @staticmethod
    def format_trace_id(trace_id: int) -> str:
        return str(UUID(int=trace_id))

    @staticmethod
    def format_span_id(span_id: int) -> str:
        return str(UUID(int=span_id))

    @staticmethod
    def build_span_exporter(base_url: str, api_key: str, headers: dict, structure_run_
↪id: str) -> SpanExporter:
        @define
        class SpanExporter(import_optional_dependency("opentelemetry.sdk.trace.export
↪").SpanExporter):
            base_url: str = field(kw_only=True)
            api_key: str = field(kw_only=True)
            headers: dict = field(kw_only=True)
            structure_run_id: str = field(kw_only=True)

            def export(self, spans: Sequence[ReadableSpan]) -> SpanExportResult:
                opentelemetry_util = import_optional_dependency("opentelemetry.sdk.
↪util")
                opentelemetry_trace_export = import_optional_dependency(
↪"opentelemetry.sdk.trace.export")

                url = griptape_cloud_url(self.base_url, f"api/structure-runs/{self.
↪structure_run_id}/spans")
                payload = [
                    {
                        "trace_id": GriptapeCloudObservabilityDriver.format_trace_
↪id(span.context.trace_id),
                        "span_id": GriptapeCloudObservabilityDriver.format_span_
```

(continues on next page)

(continued from previous page)

```
↪id(span.context.span_id),
                        "parent_id": GriptapeCloudObservabilityDriver.format_span_
↪id(span.parent.span_id)
                        if span.parent
                        else None,
                        "name": span.name,
                        "start_time": opentelemetry_util.ns_to_iso_str(span.start_
↪time) if span.start_time else None,
                        "end_time": opentelemetry_util.ns_to_iso_str(span.end_time)␣
↪if span.end_time else None,
                        "status": span.status.status_code.name,
                        "attributes": {**span.attributes} if span.attributes else {},
                        "events": [
                            {
                                "name": event.name,
                                "timestamp": opentelemetry_util.ns_to_iso_str(event.
↪timestamp)
                                if event.timestamp
                                else None,
                                "attributes": {**event.attributes} if event.
↪attributes else {},
                            }
                            for event in span.events
                        ],
                    }
                    for span in spans
                    if span.context is not None
                ]
                response = requests.post(url=url, json=payload, headers=self.headers)
                return (
                    opentelemetry_trace_export.SpanExportResult.SUCCESS
                    if response.status_code == 200
                    else opentelemetry_trace_export.SpanExportResult.FAILURE
                )

        return SpanExporter(
            base_url=base_url,
            api_key=api_key,
            headers=headers,
            structure_run_id=structure_run_id,
        )

    def get_span_id(self) -> Optional[str]:
        opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
        span = opentelemetry_trace.get_current_span()
        if span is opentelemetry_trace.INVALID_SPAN:
            return None
        return GriptapeCloudObservabilityDriver.format_span_id(span.get_span_
↪context().span_id)
```

- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']), kw_only=True)` class-attribute instance-attribute

- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')), kw_only=True)` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `span_processor = field(default=Factory(lambda self: import_optional_dependency('opentelemetry.sdk.trace.export').BatchSpanProcessor(GriptapeCloudObservabilityDriver.build_span_exporter(base_url=self.base_url, api_key=self.api_key, headers=self.headers, structure_run_id=self.structure_run_id)), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `structure_run_id = field(default=Factory(lambda: os.getenv('GT_CLOUD_STRUCTURE_RUN_ID')), kw_only=True)` class-attribute instance-attribute

build_span_exporter(base_url, api_key, headers, structure_run_id)staticmethod

Source Code in griptape/drivers/observability/griptape_cloud_observability_driver.py

```
@staticmethod
def build_span_exporter(base_url: str, api_key: str, headers: dict, structure_run_id:␣
↪str) -> SpanExporter:
    @define
    class SpanExporter(import_optional_dependency("opentelemetry.sdk.trace.export").
↪SpanExporter):
        base_url: str = field(kw_only=True)
        api_key: str = field(kw_only=True)
        headers: dict = field(kw_only=True)
        structure_run_id: str = field(kw_only=True)

        def export(self, spans: Sequence[ReadableSpan]) -> SpanExportResult:
            opentelemetry_util = import_optional_dependency("opentelemetry.sdk.util")
            opentelemetry_trace_export = import_optional_dependency("opentelemetry.
↪sdk.trace.export")

            url = griptape_cloud_url(self.base_url, f"api/structure-runs/{self.
↪structure_run_id}/spans")
            payload = [
                {
                    "trace_id": GriptapeCloudObservabilityDriver.format_trace_id(span.
↪context.trace_id),
                    "span_id": GriptapeCloudObservabilityDriver.format_span_id(span.
↪context.span_id),
                    "parent_id": GriptapeCloudObservabilityDriver.format_span_id(span.
↪parent.span_id)
                    if span.parent
                    else None,
                    "name": span.name,
                    "start_time": opentelemetry_util.ns_to_iso_str(span.start_time)␣
↪if span.start_time else None,
                    "end_time": opentelemetry_util.ns_to_iso_str(span.end_time) if␣
↪span.end_time else None,
                    "status": span.status.status_code.name,
                    "attributes": {**span.attributes} if span.attributes else {},
                    "events": [
```

(continues on next page)

(continued from previous page)

```
                        {
                            "name": event.name,
                            "timestamp": opentelemetry_util.ns_to_iso_str(event.
↪timestamp)
                            if event.timestamp
                            else None,
                            "attributes": {**event.attributes} if event.attributes␣
↪else {},
                        }
                        for event in span.events
                    ],
                }
                for span in spans
                if span.context is not None
            ]
            response = requests.post(url=url, json=payload, headers=self.headers)
            return (
                opentelemetry_trace_export.SpanExportResult.SUCCESS
                if response.status_code == 200
                else opentelemetry_trace_export.SpanExportResult.FAILURE
            )

    return SpanExporter(
        base_url=base_url,
        api_key=api_key,
        headers=headers,
        structure_run_id=structure_run_id,
    )
```

format_span_id(span_id)staticmethod

Source Code in griptape/drivers/observability/griptape_cloud_observability_driver.py

```
@staticmethod
def format_span_id(span_id: int) -> str:
    return str(UUID(int=span_id))
```

format_trace_id(trace_id)staticmethod

Source Code in griptape/drivers/observability/griptape_cloud_observability_driver.py

```
@staticmethod
def format_trace_id(trace_id: int) -> str:
    return str(UUID(int=trace_id))
```

get_span_id()

Source Code in griptape/drivers/observability/griptape_cloud_observability_driver.py

```
def get_span_id(self) -> Optional[str]:
    opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
```

(continues on next page)

(continued from previous page)

```
    span = opentelemetry_trace.get_current_span()
    if span is opentelemetry_trace.INVALID_SPAN:
        return None
    return GriptapeCloudObservabilityDriver.format_span_id(span.get_span_context().
↪span_id)
```

validate*run_id(*, structure_run_id)

Source Code in griptape/drivers/observability/griptape_cloud_observability_driver.py

```
@structure_run_id.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_run_id(self, _: Attribute, structure_run_id: Optional[str]) -> None:
    if structure_run_id is None:
        raise ValueError(
            "structure_run_id must be set either in the constructor or as an␣
↪environment variable (GT_CLOUD_STRUCTURE_RUN_ID)."
        )
```

griptape_cloud_observability_driver

Bases: *OpenTelemetryObservabilityDriver* “)

Source Code in griptape/drivers/observability/griptape_cloud_observability_driver.py

```
@define
class GriptapeCloudObservabilityDriver(OpenTelemetryObservabilityDriver):
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")), kw_only=True
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]), kw_
↪only=True)
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True), kw_only=True
    )
    structure_run_id: Optional[str] = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_STRUCTURE_RUN_ID")), kw_only=True
    )
    span_processor: SpanProcessor = field(
        default=Factory(
            lambda self: import_optional_dependency("opentelemetry.sdk.trace.export").
↪BatchSpanProcessor(
                GriptapeCloudObservabilityDriver.build_span_exporter(
                    base_url=self.base_url,
                    api_key=self.api_key,
                    headers=self.headers,
                    structure_run_id=self.structure_run_id,
                )
            ),
            takes_self=True,
```

(continues on next page)

(continued from previous page)

```
        ),
        kw_only=True,
    )

    @structure_run_id.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_run_id(self, _: Attribute, structure_run_id: Optional[str]) -> None:
        if structure_run_id is None:
            raise ValueError(
                "structure_run_id must be set either in the constructor or as an␣
↪environment variable (GT_CLOUD_STRUCTURE_RUN_ID)."
            )

    @staticmethod
    def format_trace_id(trace_id: int) -> str:
        return str(UUID(int=trace_id))

    @staticmethod
    def format_span_id(span_id: int) -> str:
        return str(UUID(int=span_id))

    @staticmethod
    def build_span_exporter(base_url: str, api_key: str, headers: dict, structure_run_
↪id: str) -> SpanExporter:
        @define
        class SpanExporter(import_optional_dependency("opentelemetry.sdk.trace.export
↪").SpanExporter):
            base_url: str = field(kw_only=True)
            api_key: str = field(kw_only=True)
            headers: dict = field(kw_only=True)
            structure_run_id: str = field(kw_only=True)

            def export(self, spans: Sequence[ReadableSpan]) -> SpanExportResult:
                opentelemetry_util = import_optional_dependency("opentelemetry.sdk.
↪util")
                opentelemetry_trace_export = import_optional_dependency(
↪"opentelemetry.sdk.trace.export")

                url = griptape_cloud_url(self.base_url, f"api/structure-runs/{self.
↪structure_run_id}/spans")
                payload = [
                    {
                        "trace_id": GriptapeCloudObservabilityDriver.format_trace_
↪id(span.context.trace_id),
                        "span_id": GriptapeCloudObservabilityDriver.format_span_
↪id(span.context.span_id),
                        "parent_id": GriptapeCloudObservabilityDriver.format_span_
↪id(span.parent.span_id)
                        if span.parent
                        else None,
                        "name": span.name,
                        "start_time": opentelemetry_util.ns_to_iso_str(span.start_
```

(continues on next page)

(continued from previous page)

```
↪time) if span.start_time else None,
                        "end_time": opentelemetry_util.ns_to_iso_str(span.end_time)␣
↪if span.end_time else None,
                        "status": span.status.status_code.name,
                        "attributes": {**span.attributes} if span.attributes else {},
                        "events": [
                            {
                                "name": event.name,
                                "timestamp": opentelemetry_util.ns_to_iso_str(event.
↪timestamp)
                                if event.timestamp
                                else None,
                                "attributes": {**event.attributes} if event.
↪attributes else {},
                            }
                            for event in span.events
                        ],
                    }
                    for span in spans
                    if span.context is not None
                ]
                response = requests.post(url=url, json=payload, headers=self.headers)
                return (
                    opentelemetry_trace_export.SpanExportResult.SUCCESS
                    if response.status_code == 200
                    else opentelemetry_trace_export.SpanExportResult.FAILURE
                )

        return SpanExporter(
            base_url=base_url,
            api_key=api_key,
            headers=headers,
            structure_run_id=structure_run_id,
        )

    def get_span_id(self) -> Optional[str]:
        opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
        span = opentelemetry_trace.get_current_span()
        if span is opentelemetry_trace.INVALID_SPAN:
            return None
        return GriptapeCloudObservabilityDriver.format_span_id(span.get_span_
↪context().span_id)
```

- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']), kw_only=True)` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')), kw_only=True)` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `span_processor = field(default=Factory(lambda self: import_optional_dependency('opentelemetry.sdk.trace.export').BatchSpanProcessor(GriptapeCloudObservabilityDriver.`

build_span_exporter(base_url=self.base_url, api_key=self.api_key, headers=self.headers, structure_run_id=self.structure_run_id)), takes_self=True), kw_only=True) class-attribute instance-attribute

- structure_run_id = field(default=Factory(lambda: os.getenv('GT_CLOUD_STRUCTURE_RUN_ID')), kw_only=True) class-attribute instance-attribute

build_span_exporter(base_url, api_key, headers, structure_run_id)staticmethod

Source Code in griptape/drivers/observability/griptape_cloud_observability_driver.py

```
@staticmethod
def build_span_exporter(base_url: str, api_key: str, headers: dict, structure_run_id:␣
↪str) -> SpanExporter:
    @define
    class SpanExporter(import_optional_dependency("opentelemetry.sdk.trace.export").
↪SpanExporter):
        base_url: str = field(kw_only=True)
        api_key: str = field(kw_only=True)
        headers: dict = field(kw_only=True)
        structure_run_id: str = field(kw_only=True)

        def export(self, spans: Sequence[ReadableSpan]) -> SpanExportResult:
            opentelemetry_util = import_optional_dependency("opentelemetry.sdk.util")
            opentelemetry_trace_export = import_optional_dependency("opentelemetry.
↪sdk.trace.export")

            url = griptape_cloud_url(self.base_url, f"api/structure-runs/{self.
↪structure_run_id}/spans")
            payload = [
                {
                    "trace_id": GriptapeCloudObservabilityDriver.format_trace_id(span.
↪context.trace_id),
                    "span_id": GriptapeCloudObservabilityDriver.format_span_id(span.
↪context.span_id),
                    "parent_id": GriptapeCloudObservabilityDriver.format_span_id(span.
↪parent.span_id)
                    if span.parent
                    else None,
                    "name": span.name,
                    "start_time": opentelemetry_util.ns_to_iso_str(span.start_time)␣
↪if span.start_time else None,
                    "end_time": opentelemetry_util.ns_to_iso_str(span.end_time) if␣
↪span.end_time else None,
                    "status": span.status.status_code.name,
                    "attributes": {**span.attributes} if span.attributes else {},
                    "events": [
                        {
                            "name": event.name,
                            "timestamp": opentelemetry_util.ns_to_iso_str(event.
↪timestamp)
                            if event.timestamp
                            else None,
                            "attributes": {**event.attributes} if event.attributes␣
```

(continues on next page)

(continued from previous page)

```
↪else {},
                        }
                        for event in span.events
                    ],
                }
                for span in spans
                if span.context is not None
            ]
            response = requests.post(url=url, json=payload, headers=self.headers)
            return (
                opentelemetry_trace_export.SpanExportResult.SUCCESS
                if response.status_code == 200
                else opentelemetry_trace_export.SpanExportResult.FAILURE
            )

    return SpanExporter(
        base_url=base_url,
        api_key=api_key,
        headers=headers,
        structure_run_id=structure_run_id,
    )
```

format_span_id(span_id)staticmethod

Source Code in griptape/drivers/observability/griptape_cloud_observability_driver.py

```
@staticmethod
def format_span_id(span_id: int) -> str:
    return str(UUID(int=span_id))
```

format_trace_id(trace_id)staticmethod

Source Code in griptape/drivers/observability/griptape_cloud_observability_driver.py

```
@staticmethod
def format_trace_id(trace_id: int) -> str:
    return str(UUID(int=trace_id))
```

get_span_id()

Source Code in griptape/drivers/observability/griptape_cloud_observability_driver.py

```
def get_span_id(self) -> Optional[str]:
    opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
    span = opentelemetry_trace.get_current_span()
    if span is opentelemetry_trace.INVALID_SPAN:
        return None
    return GriptapeCloudObservabilityDriver.format_span_id(span.get_span_context().
↪span_id)
```

validate*run_id(*, structure_run_id)

Source Code in griptape/drivers/observability/griptape_cloud_observability_driver.py

```
@structure_run_id.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_run_id(self, _: Attribute, structure_run_id: Optional[str]) -> None:
    if structure_run_id is None:
        raise ValueError(
            "structure_run_id must be set either in the constructor or as an␣
↪environment variable (GT_CLOUD_STRUCTURE_RUN_ID)."
        )
```

no_op

- `__all__ = ['NoOpObservabilityDriver']` module-attribute

Bases: *BaseObservabilityDriver* “)

Source Code in griptape/drivers/observability/no_op_observability_driver.py

```
@define
class NoOpObservabilityDriver(BaseObservabilityDriver):
    def observe(self, call: Observable.Call) -> Any:
        return call()

    def get_span_id(self) -> Optional[str]:
        return None
```

get_span_id()

Source Code in griptape/drivers/observability/no_op_observability_driver.py

```
def get_span_id(self) -> Optional[str]:
    return None
```

observe(call)

Source Code in griptape/drivers/observability/no_op_observability_driver.py

```
def observe(self, call: Observable.Call) -> Any:
    return call()
```

no_op_observability_driver

Bases: *BaseObservabilityDriver* “)

Source Code in griptape/drivers/observability/no_op_observability_driver.py

```
@define
class NoOpObservabilityDriver(BaseObservabilityDriver):
    def observe(self, call: Observable.Call) -> Any:
        return call()
```

(continues on next page)

(continued from previous page)

```
    def get_span_id(self) -> Optional[str]:
        return None
```

get_span_id()

Source Code in griptape/drivers/observability/no_op_observability_driver.py

```
def get_span_id(self) -> Optional[str]:
    return None
```

observe(call)

Source Code in griptape/drivers/observability/no_op_observability_driver.py

```
def observe(self, call: Observable.Call) -> Any:
    return call()
```

open_telemetry

- `__all__ = ['OpenTelemetryObservabilityDriver']` module-attribute

Bases: *BaseObservabilityDriver* “)

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
@define
class OpenTelemetryObservabilityDriver(BaseObservabilityDriver):
    service_name: str = field(default="griptape", kw_only=True)
    span_processor: SpanProcessor = field(kw_only=True)
    service_version: Optional[str] = field(default=None, kw_only=True)
    deployment_env: Optional[str] = field(default=None, kw_only=True)
    trace_provider: TracerProvider = field(
        default=Factory(
            lambda self: self._trace_provider_factory(),
            takes_self=True,
        ),
        kw_only=True,
    )
    _tracer: Tracer = field(init=False)
    _root_span_context_manager: Any = None

    def _trace_provider_factory(self) -> TracerProvider:
        opentelemetry_trace = import_optional_dependency("opentelemetry.sdk.trace")

        attributes = {"service.name": self.service_name}
        if self.service_version is not None:
            attributes["service.version"] = self.service_version
        if self.deployment_env is not None:
            attributes["deployment.environment"] = self.deployment_env
        return opentelemetry_trace.TracerProvider(
            resource=import_optional_dependency("opentelemetry.sdk.resources").
↪Resource(attributes=attributes)
```

(continues on next page)

(continued from previous page)

```
        )  # pyright: ignore[reportArgumentType]

    def __attrs_post_init__(self) -> None:
        opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
        self.trace_provider.add_span_processor(self.span_processor)
        self._tracer = opentelemetry_trace.get_tracer(self.service_name, tracer_
↪provider=self.trace_provider)

    def __enter__(self) -> None:
        opentelemetry_instrumentation_threading = import_optional_dependency(
↪"opentelemetry.instrumentation.threading")

        opentelemetry_instrumentation_threading.ThreadingInstrumentor().instrument()
        self._root_span_context_manager = self._tracer.start_as_current_span("main")
↪# pyright: ignore[reportCallIssue]
        self._root_span_context_manager.__enter__()

    def __exit__(
        self,
        exc_type: Optional[type[BaseException]],
        exc_value: Optional[BaseException],
        exc_traceback: Optional[TracebackType],
    ) -> bool:
        opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
        opentelemetry_instrumentation_threading = import_optional_dependency(
↪"opentelemetry.instrumentation.threading")
        root_span = opentelemetry_trace.get_current_span()
        if exc_value:
            root_span = opentelemetry_trace.get_current_span()
            root_span.set_status(opentelemetry_trace.Status(opentelemetry_trace.
↪StatusCode.ERROR))
            root_span.record_exception(exc_value)
        else:
            root_span.set_status(opentelemetry_trace.Status(opentelemetry_trace.
↪StatusCode.OK))
        if self._root_span_context_manager:
            self._root_span_context_manager.__exit__(exc_type, exc_value, exc_
↪traceback)
            self._root_span_context_manager = None
        self.trace_provider.force_flush()
        opentelemetry_instrumentation_threading.ThreadingInstrumentor().uninstrument()
        return False

    def observe(self, call: Observable.Call) -> Any:
        open_telemetry_trace = import_optional_dependency("opentelemetry.trace")
        func = call.func
        instance = call.instance
        tags = call.tags

        class_name = f"{instance.__class__.__name__}." if instance else ""
        span_name = f"{class_name}{func.__name__}()"
        with self._tracer.start_as_current_span(span_name) as span:  # pyright:␣
```

(continues on next page)

(continued from previous page)

```
↪ignore[reportCallIssue]
            if tags is not None:
                span.set_attribute("tags", tags)

            try:
                result = call()
                span.set_status(open_telemetry_trace.Status(open_telemetry_trace.
↪StatusCode.OK))
                return result
            except Exception as e:
                span.set_status(open_telemetry_trace.Status(open_telemetry_trace.
↪StatusCode.ERROR))
                span.record_exception(e)
                raise e

    def get_span_id(self) -> Optional[str]:
        opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
        span = opentelemetry_trace.get_current_span()
        if span is opentelemetry_trace.INVALID_SPAN:
            return None
        return opentelemetry_trace.format_span_id(span.get_span_context().span_id)
```

- `_root_span_context_manager = None` class-attribute instance-attribute
- `_tracer = field(init=False)` class-attribute instance-attribute
- `deployment_env = field(default=None, kw_only=True)` class-attribute instance-attribute
- `service_name = field(default='griptape', kw_only=True)` class-attribute instance-attribute
- `service_version = field(default=None, kw_only=True)` class-attribute instance-attribute
- `span_processor = field(kw_only=True)` class-attribute instance-attribute
- `trace_provider = field(default=Factory(lambda self: self._trace_provider_factory(), takes_self=True), kw_only=True)` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
def __attrs_post_init__(self) -> None:
    opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
    self.trace_provider.add_span_processor(self.span_processor)
    self._tracer = opentelemetry_trace.get_tracer(self.service_name, tracer_
↪provider=self.trace_provider)
```

enter()

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
def __enter__(self) -> None:
    opentelemetry_instrumentation_threading = import_optional_dependency(
↪"opentelemetry.instrumentation.threading")
```

(continues on next page)

(continued from previous page)

```
    opentelemetry_instrumentation_threading.ThreadingInstrumentor().instrument()
    self._root_span_context_manager = self._tracer.start_as_current_span("main")  #␣
↪pyright: ignore[reportCallIssue]
    self._root_span_context_manager.__enter__()
```

exit(exc_type, exc_value, exc_traceback)

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
def __exit__(
    self,
    exc_type: Optional[type[BaseException]],
    exc_value: Optional[BaseException],
    exc_traceback: Optional[TracebackType],
) -> bool:
    opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
    opentelemetry_instrumentation_threading = import_optional_dependency(
↪"opentelemetry.instrumentation.threading")
    root_span = opentelemetry_trace.get_current_span()
    if exc_value:
        root_span = opentelemetry_trace.get_current_span()
        root_span.set_status(opentelemetry_trace.Status(opentelemetry_trace.
↪StatusCode.ERROR))
        root_span.record_exception(exc_value)
    else:
        root_span.set_status(opentelemetry_trace.Status(opentelemetry_trace.
↪StatusCode.OK))
    if self._root_span_context_manager:
        self._root_span_context_manager.__exit__(exc_type, exc_value, exc_traceback)
        self._root_span_context_manager = None
    self.trace_provider.force_flush()
    opentelemetry_instrumentation_threading.ThreadingInstrumentor().uninstrument()
    return False
```

_trace_provider_factory()

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
def _trace_provider_factory(self) -> TracerProvider:
    opentelemetry_trace = import_optional_dependency("opentelemetry.sdk.trace")

    attributes = {"service.name": self.service_name}
    if self.service_version is not None:
        attributes["service.version"] = self.service_version
    if self.deployment_env is not None:
        attributes["deployment.environment"] = self.deployment_env
    return opentelemetry_trace.TracerProvider(
        resource=import_optional_dependency("opentelemetry.sdk.resources").
↪Resource(attributes=attributes)
    )  # pyright: ignore[reportArgumentType]
```

get_span_id()

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
def get_span_id(self) -> Optional[str]:
    opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
    span = opentelemetry_trace.get_current_span()
    if span is opentelemetry_trace.INVALID_SPAN:
        return None
    return opentelemetry_trace.format_span_id(span.get_span_context().span_id)
```

observe(call)

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
def observe(self, call: Observable.Call) -> Any:
    open_telemetry_trace = import_optional_dependency("opentelemetry.trace")
    func = call.func
    instance = call.instance
    tags = call.tags

    class_name = f"{instance.__class__.__name__}." if instance else ""
    span_name = f"{class_name}{func.__name__}()"
    with self._tracer.start_as_current_span(span_name) as span:  # pyright:␣
↪ignore[reportCallIssue]
        if tags is not None:
            span.set_attribute("tags", tags)

        try:
            result = call()
            span.set_status(open_telemetry_trace.Status(open_telemetry_trace.
↪StatusCode.OK))
            return result
        except Exception as e:
            span.set_status(open_telemetry_trace.Status(open_telemetry_trace.
↪StatusCode.ERROR))
            span.record_exception(e)
            raise e
```

open_telemetry_observability_driver

Bases: *BaseObservabilityDriver* “)

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
@define
class OpenTelemetryObservabilityDriver(BaseObservabilityDriver):
    service_name: str = field(default="griptape", kw_only=True)
    span_processor: SpanProcessor = field(kw_only=True)
    service_version: Optional[str] = field(default=None, kw_only=True)
    deployment_env: Optional[str] = field(default=None, kw_only=True)
    trace_provider: TracerProvider = field(
        default=Factory(
```

(continues on next page)

(continued from previous page)

```
            lambda self: self._trace_provider_factory(),
            takes_self=True,
        ),
        kw_only=True,
    )
    _tracer: Tracer = field(init=False)
    _root_span_context_manager: Any = None

    def _trace_provider_factory(self) -> TracerProvider:
        opentelemetry_trace = import_optional_dependency("opentelemetry.sdk.trace")

        attributes = {"service.name": self.service_name}
        if self.service_version is not None:
            attributes["service.version"] = self.service_version
        if self.deployment_env is not None:
            attributes["deployment.environment"] = self.deployment_env
        return opentelemetry_trace.TracerProvider(
            resource=import_optional_dependency("opentelemetry.sdk.resources").
↪Resource(attributes=attributes)
        )  # pyright: ignore[reportArgumentType]

    def __attrs_post_init__(self) -> None:
        opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
        self.trace_provider.add_span_processor(self.span_processor)
        self._tracer = opentelemetry_trace.get_tracer(self.service_name, tracer_
↪provider=self.trace_provider)

    def __enter__(self) -> None:
        opentelemetry_instrumentation_threading = import_optional_dependency(
↪"opentelemetry.instrumentation.threading")

        opentelemetry_instrumentation_threading.ThreadingInstrumentor().instrument()
        self._root_span_context_manager = self._tracer.start_as_current_span("main")
↪# pyright: ignore[reportCallIssue]
        self._root_span_context_manager.__enter__()

    def __exit__(
        self,
        exc_type: Optional[type[BaseException]],
        exc_value: Optional[BaseException],
        exc_traceback: Optional[TracebackType],
    ) -> bool:
        opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
        opentelemetry_instrumentation_threading = import_optional_dependency(
↪"opentelemetry.instrumentation.threading")
        root_span = opentelemetry_trace.get_current_span()
        if exc_value:
            root_span = opentelemetry_trace.get_current_span()
            root_span.set_status(opentelemetry_trace.Status(opentelemetry_trace.
↪StatusCode.ERROR))
            root_span.record_exception(exc_value)
        else:
```

(continues on next page)

(continued from previous page)

```
            root_span.set_status(opentelemetry_trace.Status(opentelemetry_trace.
↪StatusCode.OK))
        if self._root_span_context_manager:
            self._root_span_context_manager.__exit__(exc_type, exc_value, exc_
↪traceback)
            self._root_span_context_manager = None
        self.trace_provider.force_flush()
        opentelemetry_instrumentation_threading.ThreadingInstrumentor().uninstrument()
        return False

    def observe(self, call: Observable.Call) -> Any:
        open_telemetry_trace = import_optional_dependency("opentelemetry.trace")
        func = call.func
        instance = call.instance
        tags = call.tags

        class_name = f"{instance.__class__.__name__}." if instance else ""
        span_name = f"{class_name}{func.__name__}()"
        with self._tracer.start_as_current_span(span_name) as span:  # pyright:␣
↪ignore[reportCallIssue]
            if tags is not None:
                span.set_attribute("tags", tags)

            try:
                result = call()
                span.set_status(open_telemetry_trace.Status(open_telemetry_trace.
↪StatusCode.OK))
                return result
            except Exception as e:
                span.set_status(open_telemetry_trace.Status(open_telemetry_trace.
↪StatusCode.ERROR))
                span.record_exception(e)
                raise e

    def get_span_id(self) -> Optional[str]:
        opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
        span = opentelemetry_trace.get_current_span()
        if span is opentelemetry_trace.INVALID_SPAN:
            return None
        return opentelemetry_trace.format_span_id(span.get_span_context().span_id)
```

- `_root_span_context_manager = None` class-attribute instance-attribute
- `_tracer = field(init=False)` class-attribute instance-attribute
- `deployment_env = field(default=None, kw_only=True)` class-attribute instance-attribute
- `service_name = field(default='griptape', kw_only=True)` class-attribute instance-attribute
- `service_version = field(default=None, kw_only=True)` class-attribute instance-attribute
- `span_processor = field(kw_only=True)` class-attribute instance-attribute
- `trace_provider = field(default=Factory(lambda self: self._trace_provider_factory(), takes_self=True), kw_only=True)` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
def __attrs_post_init__(self) -> None:
    opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
    self.trace_provider.add_span_processor(self.span_processor)
    self._tracer = opentelemetry_trace.get_tracer(self.service_name, tracer_
↪provider=self.trace_provider)
```

enter()

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
def __enter__(self) -> None:
    opentelemetry_instrumentation_threading = import_optional_dependency(
↪"opentelemetry.instrumentation.threading")

    opentelemetry_instrumentation_threading.ThreadingInstrumentor().instrument()
    self._root_span_context_manager = self._tracer.start_as_current_span("main")  #␣
↪pyright: ignore[reportCallIssue]
    self._root_span_context_manager.__enter__()
```

exit(exc_type, exc_value, exc_traceback)

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
def __exit__(
    self,
    exc_type: Optional[type[BaseException]],
    exc_value: Optional[BaseException],
    exc_traceback: Optional[TracebackType],
) -> bool:
    opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
    opentelemetry_instrumentation_threading = import_optional_dependency(
↪"opentelemetry.instrumentation.threading")
    root_span = opentelemetry_trace.get_current_span()
    if exc_value:
        root_span = opentelemetry_trace.get_current_span()
        root_span.set_status(opentelemetry_trace.Status(opentelemetry_trace.
↪StatusCode.ERROR))
        root_span.record_exception(exc_value)
    else:
        root_span.set_status(opentelemetry_trace.Status(opentelemetry_trace.
↪StatusCode.OK))
    if self._root_span_context_manager:
        self._root_span_context_manager.__exit__(exc_type, exc_value, exc_traceback)
        self._root_span_context_manager = None
    self.trace_provider.force_flush()
    opentelemetry_instrumentation_threading.ThreadingInstrumentor().uninstrument()
    return False
```

_trace_provider_factory()

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
def _trace_provider_factory(self) -> TracerProvider:
    opentelemetry_trace = import_optional_dependency("opentelemetry.sdk.trace")

    attributes = {"service.name": self.service_name}
    if self.service_version is not None:
        attributes["service.version"] = self.service_version
    if self.deployment_env is not None:
        attributes["deployment.environment"] = self.deployment_env
    return opentelemetry_trace.TracerProvider(
        resource=import_optional_dependency("opentelemetry.sdk.resources").
↪Resource(attributes=attributes)
    )  # pyright: ignore[reportArgumentType]
```

get_span_id()

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
def get_span_id(self) -> Optional[str]:
    opentelemetry_trace = import_optional_dependency("opentelemetry.trace")
    span = opentelemetry_trace.get_current_span()
    if span is opentelemetry_trace.INVALID_SPAN:
        return None
    return opentelemetry_trace.format_span_id(span.get_span_context().span_id)
```

observe(call)

Source Code in griptape/drivers/observability/open_telemetry_observability_driver.py

```
def observe(self, call: Observable.Call) -> Any:
    open_telemetry_trace = import_optional_dependency("opentelemetry.trace")
    func = call.func
    instance = call.instance
    tags = call.tags

    class_name = f"{instance.__class__.__name__}." if instance else ""
    span_name = f"{class_name}{func.__name__}()"
    with self._tracer.start_as_current_span(span_name) as span:  # pyright:␣
↪ignore[reportCallIssue]
        if tags is not None:
            span.set_attribute("tags", tags)

        try:
            result = call()
            span.set_status(open_telemetry_trace.Status(open_telemetry_trace.
↪StatusCode.OK))
            return result
        except Exception as e:
            span.set_status(open_telemetry_trace.Status(open_telemetry_trace.
↪StatusCode.ERROR))
```

(continues on next page)

(continued from previous page)

```
            span.record_exception(e)
            raise e
```

prompt

- `__all__ = ['BasePromptDriver']` module-attribute

Bases: *SerializableMixin* “) , *ExponentialBackoffMixin* ”) , ABC

Attributes

Name	Type	Description
<code class="language-python">	*float*	The temperature to use for the completion.
<code class="language-python">	*Optional[int]*	The maximum number of tokens to generate. If not specified, the value will be automatically generated based by the tokenizer.
	str	A function that converts a *PromptStack* to a string.

continues on next page

Table 188 – continued from previous page

Name	Type	Description
<code class="language-python">	*tuple[type[Exception], ...]*	A tuple of exception types to ignore.
<code class="language-python">	*str*	The model name.
<code class="language-python">	<code class="language-python">BaseTokenizer</code>	An instance of *BaseTokenizer* to when calculating tokens.
<a href="./#griptape.drivers.prompt.BasePromptDriver.stream" title="stream = field(default=False, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.BasePromptDriver.stream)"><code class="language-python">	*bool*	Whether to stream the completion or not. *CompletionChunkEvent`s will be published to the `Structure* if one is provided.

continues on next page

Table 188 – continued from previous page

Name	Type	Description
<a href="./#griptape.drivers.prompt.BasePromptDriver.use_native_tools" title="use_native_tools = field(default=False, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.BasePromptDriver.use_native_tools)"><code class="language-python">	*bool*	Whether to use LLM's native function calling capabilities. Must be supported by the model.
<a href="./#griptape.drivers.prompt.BasePromptDriver.extra_params" title="extra_params = field(factory=dict, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.BasePromptDriver.extra_params)"><code class="language-python">	*dict*	Extra parameters to pass to the model.

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
@define(kw_only=True)
class BasePromptDriver(SerializableMixin, ExponentialBackoffMixin, ABC):
    """Base class for the Prompt Drivers.

    Attributes:
        temperature: The temperature to use for the completion.
        max_tokens: The maximum number of tokens to generate. If not specified, the␣
↪value will be automatically generated based by the tokenizer.
        prompt_stack_to_string: A function that converts a `PromptStack` to a string.
        ignored_exception_types: A tuple of exception types to ignore.
        model: The model name.
        tokenizer: An instance of `BaseTokenizer` to when calculating tokens.
        stream: Whether to stream the completion or not. `CompletionChunkEvent`s will␣
↪be published to the `Structure` if one is provided.
        use_native_tools: Whether to use LLMs native function calling capabilities.␣
↪Must be supported by the model.
        extra_params: Extra parameters to pass to the model.
    """
```

(continues on next page)

(continued from previous page)

```
    temperature: float = field(default=0.1, metadata={"serializable": True})
    max_tokens: Optional[int] = field(default=None, metadata={"serializable": True})
    ignored_exception_types: tuple[type[Exception], ...] =␣
↪field(default=Factory(lambda: (ImportError, ValueError)))
    model: str = field(metadata={"serializable": True})
    tokenizer: BaseTokenizer
    stream: bool = field(default=False, kw_only=True, metadata={"serializable": True})
    use_native_tools: bool = field(default=False, kw_only=True, metadata={
↪"serializable": True})
    structured_output_strategy: StructuredOutputStrategy = field(
        default="rule", kw_only=True, metadata={"serializable": True}
    )
    extra_params: dict = field(factory=dict, kw_only=True, metadata={"serializable":␣
↪True})

    def before_run(self, prompt_stack: PromptStack) -> None:
        self._init_structured_output(prompt_stack)
        EventBus.publish_event(StartPromptEvent(model=self.model, prompt_stack=prompt_
↪stack))

    def after_run(self, result: Message) -> None:
        EventBus.publish_event(
            FinishPromptEvent(
                model=self.model,
                result=result.value,
                input_token_count=result.usage.input_tokens,
                output_token_count=result.usage.output_tokens,
            ),
        )

    @observable(tags=["PromptDriver.run()"])
    def run(self, prompt_input: PromptStack | BaseArtifact) -> Message:
        if isinstance(prompt_input, BaseArtifact):
            prompt_stack = PromptStack.from_artifact(prompt_input)
        else:
            prompt_stack = prompt_input

        for attempt in self.retrying():
            with attempt:
                self.before_run(prompt_stack)

                result = self.__process_stream(prompt_stack) if self.stream else self.
↪__process_run(prompt_stack)

                self.after_run(result)

                return result
        raise Exception("prompt driver failed after all retry attempts")

    def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
        """Converts a Prompt Stack to a string for token counting or model prompt_
↪input.
```

(continues on next page)

(continued from previous page)

```
        This base implementation is only a rough approximation, and should be␣
↪overridden by subclasses with model-specific tokens.

        Args:
            prompt_stack: The Prompt Stack to convert to a string.

        Returns:
            A single string representation of the Prompt Stack.
        """
        prompt_lines = []

        for i in prompt_stack.messages:
            content = i.to_text()
            if i.is_user():
                prompt_lines.append(f"User: {content}")
            elif i.is_assistant():
                prompt_lines.append(f"Assistant: {content}")
            else:
                prompt_lines.append(content)

        prompt_lines.append("Assistant:")

        return "\n\n".join(prompt_lines)

    @abstractmethod
    def try_run(self, prompt_stack: PromptStack) -> Message: ...

    @abstractmethod
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]: ...

    def _init_structured_output(self, prompt_stack: PromptStack) -> None:
        from griptape.tools import StructuredOutputTool

        if (output_schema := prompt_stack.output_schema) is not None:
            if self.structured_output_strategy == "tool":
                structured_output_tool = StructuredOutputTool(output_schema=output_
↪schema)
                if structured_output_tool not in prompt_stack.tools:
                    prompt_stack.tools.append(structured_output_tool)
            elif self.structured_output_strategy == "rule":
                output_artifact = TextArtifact(JsonSchemaRule(prompt_stack.to_output_
↪json_schema()).to_text())
                system_messages = prompt_stack.system_messages
                if system_messages:
                    last_system_message = prompt_stack.system_messages[-1]
                    last_system_message.content.extend(
                        [
                            TextMessageContent(TextArtifact("\n\n")),
                            TextMessageContent(output_artifact),
                        ]
                    )
```

(continues on next page)

(continued from previous page)

```
                else:
                    prompt_stack.messages.insert(
                        0,
                        Message(
                            content=[TextMessageContent(output_artifact)],
                            role=Message.SYSTEM_ROLE,
                        ),
                    )

    def __process_run(self, prompt_stack: PromptStack) -> Message:
        return self.try_run(prompt_stack)

    def __process_stream(self, prompt_stack: PromptStack) -> Message:
        delta_contents: dict[int, list[BaseDeltaMessageContent]] = {}
        usage = DeltaMessage.Usage()

        # Aggregate all content deltas from the stream
        message_deltas = self.try_stream(prompt_stack)
        for message_delta in message_deltas:
            usage += message_delta.usage
            content = message_delta.content

            if content is not None:
                if content.index in delta_contents:
                    delta_contents[content.index].append(content)
                else:
                    delta_contents[content.index] = [content]
                if isinstance(content, TextDeltaMessageContent):
                    EventBus.publish_event(TextChunkEvent(token=content.text,␣
↪index=content.index))
                elif isinstance(content, AudioDeltaMessageContent) and content.data␣
↪is not None:
                    EventBus.publish_event(AudioChunkEvent(data=content.data))
                elif isinstance(content, ActionCallDeltaMessageContent):
                    EventBus.publish_event(
                        ActionChunkEvent(
                            partial_input=content.partial_input,
                            tag=content.tag,
                            name=content.name,
                            path=content.path,
                            index=content.index,
                        ),
                    )

        # Build a complete content from the content deltas
        return self.__build_message(list(delta_contents.values()), usage)

    def __build_message(
        self, delta_contents: list[list[BaseDeltaMessageContent]], usage:␣
↪DeltaMessage.Usage
    ) -> Message:
        content = []
```

(continues on next page)

(continued from previous page)

```
        for delta_content in delta_contents:
            text_deltas = [delta for delta in delta_content if isinstance(delta,␣
↪TextDeltaMessageContent)]
            audio_deltas = [delta for delta in delta_content if isinstance(delta,␣
↪AudioDeltaMessageContent)]
            action_deltas = [delta for delta in delta_content if isinstance(delta,␣
↪ActionCallDeltaMessageContent)]

            if text_deltas:
                content.append(TextMessageContent.from_deltas(text_deltas))
            if audio_deltas:
                content.append(AudioMessageContent.from_deltas(audio_deltas))
            if action_deltas:
                content.append(ActionCallMessageContent.from_deltas(action_deltas))

        return Message(
            content=content,
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(input_tokens=usage.input_tokens, output_tokens=usage.
↪output_tokens),
        )
```

- `extra_params = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `ignored_exception_types = field(default=Factory(lambda: (ImportError, ValueError)))` class-attribute instance-attribute
- `max_tokens = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(metadata={'serializable': True})` class-attribute instance-attribute
- `stream = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='rule', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `temperature = field(default=0.1, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer` instance-attribute
- `use_native_tools = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__build_message(delta_contents, usage)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def __build_message(
    self, delta_contents: list[list[BaseDeltaMessageContent]], usage: DeltaMessage.
↪Usage
) -> Message:
    content = []
```

(continues on next page)

(continued from previous page)

```
    for delta_content in delta_contents:
        text_deltas = [delta for delta in delta_content if isinstance(delta,␣
↪TextDeltaMessageContent)]
        audio_deltas = [delta for delta in delta_content if isinstance(delta,␣
↪AudioDeltaMessageContent)]
        action_deltas = [delta for delta in delta_content if isinstance(delta,␣
↪ActionCallDeltaMessageContent)]

        if text_deltas:
            content.append(TextMessageContent.from_deltas(text_deltas))
        if audio_deltas:
            content.append(AudioMessageContent.from_deltas(audio_deltas))
        if action_deltas:
            content.append(ActionCallMessageContent.from_deltas(action_deltas))

    return Message(
        content=content,
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(input_tokens=usage.input_tokens, output_tokens=usage.
↪output_tokens),
    )
```

__process_run(prompt_stack)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def __process_run(self, prompt_stack: PromptStack) -> Message:
    return self.try_run(prompt_stack)
```

__process_stream(prompt_stack)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def __process_stream(self, prompt_stack: PromptStack) -> Message:
    delta_contents: dict[int, list[BaseDeltaMessageContent]] = {}
    usage = DeltaMessage.Usage()

    # Aggregate all content deltas from the stream
    message_deltas = self.try_stream(prompt_stack)
    for message_delta in message_deltas:
        usage += message_delta.usage
        content = message_delta.content

        if content is not None:
            if content.index in delta_contents:
                delta_contents[content.index].append(content)
            else:
                delta_contents[content.index] = [content]
            if isinstance(content, TextDeltaMessageContent):
                EventBus.publish_event(TextChunkEvent(token=content.text,␣
↪index=content.index))
```

(continues on next page)

(continued from previous page)

```
            elif isinstance(content, AudioDeltaMessageContent) and content.data is␣
↪not None:
                EventBus.publish_event(AudioChunkEvent(data=content.data))
            elif isinstance(content, ActionCallDeltaMessageContent):
                EventBus.publish_event(
                    ActionChunkEvent(
                        partial_input=content.partial_input,
                        tag=content.tag,
                        name=content.name,
                        path=content.path,
                        index=content.index,
                    ),
                )

    # Build a complete content from the content deltas
    return self.__build_message(list(delta_contents.values()), usage)
```

_init_structured_output(prompt_stack)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def _init_structured_output(self, prompt_stack: PromptStack) -> None:
    from griptape.tools import StructuredOutputTool

    if (output_schema := prompt_stack.output_schema) is not None:
        if self.structured_output_strategy == "tool":
            structured_output_tool = StructuredOutputTool(output_schema=output_schema)
            if structured_output_tool not in prompt_stack.tools:
                prompt_stack.tools.append(structured_output_tool)
        elif self.structured_output_strategy == "rule":
            output_artifact = TextArtifact(JsonSchemaRule(prompt_stack.to_output_json_
↪schema()).to_text())
            system_messages = prompt_stack.system_messages
            if system_messages:
                last_system_message = prompt_stack.system_messages[-1]
                last_system_message.content.extend(
                    [
                        TextMessageContent(TextArtifact("\n\n")),
                        TextMessageContent(output_artifact),
                    ]
                )
            else:
                prompt_stack.messages.insert(
                    0,
                    Message(
                        content=[TextMessageContent(output_artifact)],
                        role=Message.SYSTEM_ROLE,
                    ),
                )
```

after_run(result)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def after_run(self, result: Message) -> None:
    EventBus.publish_event(
        FinishPromptEvent(
            model=self.model,
            result=result.value,
            input_token_count=result.usage.input_tokens,
            output_token_count=result.usage.output_tokens,
        ),
    )
```

before_run(prompt_stack)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def before_run(self, prompt_stack: PromptStack) -> None:
    self._init_structured_output(prompt_stack)
    EventBus.publish_event(StartPromptEvent(model=self.model, prompt_stack=prompt_
↪stack))
```

prompt_stack_to_string(prompt_stack)

Converts a Prompt Stack to a string for token counting or model prompt_input.

This base implementation is only a rough approximation, and should be overridden by subclasses with model-specific tokens.

Parameters

Name	Type	Description	Default
prompt_stack	<a href="../../common#griptape.common.PromptStack" title="PromptStack (griptape.common.PromptStack)"><code class="language-pythc	The Prompt Stack to convert to a string. 	*required*

Returns

Type	Description
str	A single string representation of the Prompt Stack.

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
    """Converts a Prompt Stack to a string for token counting or model prompt_input.

    This base implementation is only a rough approximation, and should be overridden␣
↪by subclasses with model-specific tokens.

    Args:
        prompt_stack: The Prompt Stack to convert to a string.

    Returns:
        A single string representation of the Prompt Stack.
    """
    prompt_lines = []

    for i in prompt_stack.messages:
        content = i.to_text()
        if i.is_user():
            prompt_lines.append(f"User: {content}")
        elif i.is_assistant():
            prompt_lines.append(f"Assistant: {content}")
        else:
            prompt_lines.append(content)

    prompt_lines.append("Assistant:")

    return "\n\n".join(prompt_lines)
```

run(prompt_input)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
@observable(tags=["PromptDriver.run()"])
def run(self, prompt_input: PromptStack | BaseArtifact) -> Message:
    if isinstance(prompt_input, BaseArtifact):
        prompt_stack = PromptStack.from_artifact(prompt_input)
    else:
        prompt_stack = prompt_input

    for attempt in self.retrying():
        with attempt:
            self.before_run(prompt_stack)

            result = self.__process_stream(prompt_stack) if self.stream else self.__
↪process_run(prompt_stack)

            self.after_run(result)

            return result
    raise Exception("prompt driver failed after all retry attempts")
```

try_run(prompt_stack)abstractmethod

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
@abstractmethod
def try_run(self, prompt_stack: PromptStack) -> Message: ...
```

try_stream(prompt_stack)abstractmethod

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
@abstractmethod
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]: ...
```

amazon_bedrock

- `__all__ = ['AmazonBedrockPromptDriver']` module-attribute

Bases: *BasePromptDriver* “)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
@define
class AmazonBedrockPromptDriver(BasePromptDriver):
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    additional_model_request_fields: dict = field(default=Factory(dict), kw_only=True)
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: AmazonBedrockTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
    use_native_tools: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})
    structured_output_strategy: StructuredOutputStrategy = field(
        default="tool", kw_only=True, metadata={"serializable": True}
    )
    tool_choice: dict = field(default=Factory(lambda: {"auto": {}}), kw_only=True,␣
↪metadata={"serializable": True})
    _client: Optional[Any] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @structured_output_strategy.validator  # pyright:␣
↪ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
        if value == "native":
            raise ValueError(f"{__class__.__name__} does not support `{value}`␣
↪structured output strategy.")

        return value

    @lazy_property()
    def client(self) -> Any:
        return self.session.client("bedrock-runtime")
```

(continues on next page)

(continued from previous page)

```
    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        params = self._base_params(prompt_stack)
        logger.debug(params)
        response = self.client.converse(**params)
        logger.debug(response)

        usage = response["usage"]
        output_message = response["output"]["message"]

        message_content = output_message.get("content", [])

        # Move reasoning content to the beginning of the content to have it appear␣
↪first in output
        reasoning_content_index = next(
            (i for i, content in enumerate(message_content) if "reasoningContent" in␣
↪content),
            None,
        )
        if reasoning_content_index is not None:
            message_content.insert(0, message_content.pop(reasoning_content_index))

        return Message(
            content=[self.__to_prompt_stack_message_content(content) for content in␣
↪message_content],
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(input_tokens=usage["inputTokens"], output_
↪tokens=usage["outputTokens"]),
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        params = self._base_params(prompt_stack)
        logger.debug(params)
        response = self.client.converse_stream(**params)

        stream = response.get("stream")
        if stream is not None:
            for event in stream:
                logger.debug(event)
                if "contentBlockDelta" in event or "contentBlockStart" in event:
                    yield DeltaMessage(content=self.__to_prompt_stack_delta_message_
↪content(event))
                elif "metadata" in event:
                    usage = event["metadata"]["usage"]
                    yield DeltaMessage(
                        usage=DeltaMessage.Usage(
                            input_tokens=usage["inputTokens"],
                            output_tokens=usage["outputTokens"],
                        ),
                    )
```

(continues on next page)

(continued from previous page)

```
        else:
            raise Exception("model response is empty")

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        system_messages = [{"text": message.to_text()} for message in prompt_stack.
↪system_messages]
        messages = self.__to_bedrock_messages([message for message in prompt_stack.
↪messages if not message.is_system()])

        params = {
            "modelId": self.model,
            "messages": messages,
            "system": system_messages,
            "inferenceConfig": {
                "temperature": self.temperature,
                **({"maxTokens": self.max_tokens} if self.max_tokens is not None else
↪{}),
            },
            "additionalModelRequestFields": self.additional_model_request_fields,
            **self.extra_params,
        }

        if prompt_stack.tools and self.use_native_tools:
            params["toolConfig"] = {
                "tools": [],
                "toolChoice": self.tool_choice,
            }

            if prompt_stack.output_schema is not None and self.structured_output_
↪strategy == "tool":
                params["toolConfig"]["toolChoice"] = {"any": {}}

            params["toolConfig"]["tools"] = self.__to_bedrock_tools(prompt_stack.
↪tools)

        return params

    def __to_bedrock_messages(self, messages: list[Message]) -> list[dict]:
        return [
            {
                "role": self.__to_bedrock_role(message),
                "content": [self.__to_bedrock_message_content(content) for content in␣
↪message.content],
            }
            for message in messages
        ]

    def __to_bedrock_role(self, message: Message) -> str:
        if message.is_assistant():
            return "assistant"
        return "user"
```

(continues on next page)

(continued from previous page)

```
    def __to_bedrock_tools(self, tools: list[BaseTool]) -> list[dict]:
        return [
            {
                "toolSpec": {
                    "name": tool.to_native_tool_name(activity),
                    "description": tool.activity_description(activity),
                    "inputSchema": {
                        "json": tool.to_activity_json_schema(activity, "http://json-
↪schema.org/draft-07/schema#"),
                    },
                },
            }
            for tool in tools
            for activity in tool.activities()
        ]

    def __to_bedrock_message_content(self, content: BaseMessageContent) -> dict:
        if isinstance(content, TextMessageContent):
            return {"text": content.artifact.to_text()}
        if isinstance(content, ImageMessageContent):
            artifact = content.artifact
            if isinstance(artifact, ImageArtifact):
                return {"image": {"format": artifact.format, "source": {"bytes":␣
↪artifact.value}}}
            if isinstance(artifact, ImageUrlArtifact):
                return {"image": {"format": "png", "source": {"s3Location": {"uri":␣
↪artifact.value}}}}
            raise ValueError(f"Unsupported image artifact type: {type(artifact)}")
        if isinstance(content, ActionCallMessageContent):
            action_call = content.artifact.value

            return {
                "toolUse": {
                    "toolUseId": action_call.tag,
                    "name": f"{action_call.name}_{action_call.path}",
                    "input": action_call.input,
                },
            }
        if isinstance(content, ActionResultMessageContent):
            artifact = content.artifact

            if isinstance(artifact, ListArtifact):
                message_content = [self.__to_bedrock_tool_use_content(artifact) for␣
↪artifact in artifact.value]
            else:
                message_content = [self.__to_bedrock_tool_use_content(artifact)]

            return {
                "toolResult": {
                    "toolUseId": content.action.tag,
                    "content": message_content,
                    "status": "error" if isinstance(artifact, ErrorArtifact) else
```

(continues on next page)

(continued from previous page)

```
↪"success",
                },
            }
        return content.artifact.value

    def __to_bedrock_tool_use_content(self, artifact: BaseArtifact) -> dict:
        if isinstance(artifact, ImageArtifact):
            return {"image": {"format": artifact.format, "source": {"bytes": artifact.
↪value}}}
        return {"text": artifact.to_text()}

    def __to_prompt_stack_message_content(self, content: dict) -> BaseMessageContent:
        if "text" in content:
            return TextMessageContent(TextArtifact(content["text"]))
        if "toolUse" in content:
            name, path = ToolAction.from_native_tool_name(content["toolUse"]["name"])
            return ActionCallMessageContent(
                artifact=ActionArtifact(
                    value=ToolAction(
                        tag=content["toolUse"]["toolUseId"],
                        name=name,
                        path=path,
                        input=content["toolUse"]["input"],
                    ),
                ),
            )
        if "reasoningContent" in content:
            return TextMessageContent(TextArtifact(content["reasoningContent"][
↪"reasoningText"]["text"]))
        raise ValueError(f"Unsupported message content type: {content}")

    def __to_prompt_stack_delta_message_content(self, event: dict) ->␣
↪BaseDeltaMessageContent:
        if "contentBlockStart" in event:
            content_block = event["contentBlockStart"]["start"]

            if "toolUse" in content_block:
                name, path = ToolAction.from_native_tool_name(content_block["toolUse
↪"]["name"])

                return ActionCallDeltaMessageContent(
                    index=event["contentBlockStart"]["contentBlockIndex"],
                    tag=content_block["toolUse"]["toolUseId"],
                    name=name,
                    path=path,
                )
            if "text" in content_block:
                return TextDeltaMessageContent(
                    content_block["text"],
                    index=event["contentBlockStart"]["contentBlockIndex"],
                )
            raise ValueError(f"Unsupported message content type: {event}")
```

(continues on next page)

(continued from previous page)

```
        if "contentBlockDelta" in event:
            content_block_delta = event["contentBlockDelta"]

            if "text" in content_block_delta["delta"]:
                return TextDeltaMessageContent(
                    content_block_delta["delta"]["text"],
                    index=content_block_delta["contentBlockIndex"],
                )
            if "toolUse" in content_block_delta["delta"]:
                return ActionCallDeltaMessageContent(
                    index=content_block_delta["contentBlockIndex"],
                    partial_input=content_block_delta["delta"]["toolUse"]["input"],
                )
            raise ValueError(f"Unsupported message content type: {event}")
        raise ValueError(f"Unsupported message content type: {event}")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `additional_model_request_fields = field(default=Factory(dict), kw_only=True)` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `structured_output_strategy = field(default='tool', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: AmazonBedrockTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `tool_choice = field(default=Factory(lambda: {'auto': {}}), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `use_native_tools = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__to_bedrock_message_content(content)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_bedrock_message_content(self, content: BaseMessageContent) -> dict:
    if isinstance(content, TextMessageContent):
        return {"text": content.artifact.to_text()}
    if isinstance(content, ImageMessageContent):
        artifact = content.artifact
        if isinstance(artifact, ImageArtifact):
            return {"image": {"format": artifact.format, "source": {"bytes": artifact.
↪value}}}
        if isinstance(artifact, ImageUrlArtifact):
            return {"image": {"format": "png", "source": {"s3Location": {"uri":␣
↪artifact.value}}}}
        raise ValueError(f"Unsupported image artifact type: {type(artifact)}")
    if isinstance(content, ActionCallMessageContent):
        action_call = content.artifact.value
```

(continues on next page)

(continued from previous page)

```
        return {
            "toolUse": {
                "toolUseId": action_call.tag,
                "name": f"{action_call.name}_{action_call.path}",
                "input": action_call.input,
            },
        }
    if isinstance(content, ActionResultMessageContent):
        artifact = content.artifact

        if isinstance(artifact, ListArtifact):
            message_content = [self.__to_bedrock_tool_use_content(artifact) for␣
↪artifact in artifact.value]
        else:
            message_content = [self.__to_bedrock_tool_use_content(artifact)]

        return {
            "toolResult": {
                "toolUseId": content.action.tag,
                "content": message_content,
                "status": "error" if isinstance(artifact, ErrorArtifact) else "success
↪",
            },
        }
    return content.artifact.value
```

__to_bedrock_messages(messages)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_bedrock_messages(self, messages: list[Message]) -> list[dict]:
    return [
        {
            "role": self.__to_bedrock_role(message),
            "content": [self.__to_bedrock_message_content(content) for content in␣
↪message.content],
        }
        for message in messages
    ]
```

__to_bedrock_role(message)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_bedrock_role(self, message: Message) -> str:
    if message.is_assistant():
        return "assistant"
    return "user"
```

__to_bedrock_tool_use_content(artifact)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_bedrock_tool_use_content(self, artifact: BaseArtifact) -> dict:
    if isinstance(artifact, ImageArtifact):
        return {"image": {"format": artifact.format, "source": {"bytes": artifact.
↪value}}}
    return {"text": artifact.to_text()}
```

__to_bedrock_tools(tools)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_bedrock_tools(self, tools: list[BaseTool]) -> list[dict]:
    return [
        {
            "toolSpec": {
                "name": tool.to_native_tool_name(activity),
                "description": tool.activity_description(activity),
                "inputSchema": {
                    "json": tool.to_activity_json_schema(activity, "http://json-
↪schema.org/draft-07/schema#"),
                },
            },
        }
        for tool in tools
        for activity in tool.activities()
    ]
```

__to_prompt_stack_delta_message_content(event)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_prompt_stack_delta_message_content(self, event: dict) ->␣
↪BaseDeltaMessageContent:
    if "contentBlockStart" in event:
        content_block = event["contentBlockStart"]["start"]

        if "toolUse" in content_block:
            name, path = ToolAction.from_native_tool_name(content_block["toolUse"][
↪"name"])

            return ActionCallDeltaMessageContent(
                index=event["contentBlockStart"]["contentBlockIndex"],
                tag=content_block["toolUse"]["toolUseId"],
                name=name,
                path=path,
            )
        if "text" in content_block:
            return TextDeltaMessageContent(
                content_block["text"],
                index=event["contentBlockStart"]["contentBlockIndex"],
```

(continues on next page)

(continued from previous page)

```
            )
        raise ValueError(f"Unsupported message content type: {event}")
    if "contentBlockDelta" in event:
        content_block_delta = event["contentBlockDelta"]

        if "text" in content_block_delta["delta"]:
            return TextDeltaMessageContent(
                content_block_delta["delta"]["text"],
                index=content_block_delta["contentBlockIndex"],
            )
        if "toolUse" in content_block_delta["delta"]:
            return ActionCallDeltaMessageContent(
                index=content_block_delta["contentBlockIndex"],
                partial_input=content_block_delta["delta"]["toolUse"]["input"],
            )
        raise ValueError(f"Unsupported message content type: {event}")
    raise ValueError(f"Unsupported message content type: {event}")
```

__to_prompt_stack_message_content(content)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_prompt_stack_message_content(self, content: dict) -> BaseMessageContent:
    if "text" in content:
        return TextMessageContent(TextArtifact(content["text"]))
    if "toolUse" in content:
        name, path = ToolAction.from_native_tool_name(content["toolUse"]["name"])
        return ActionCallMessageContent(
            artifact=ActionArtifact(
                value=ToolAction(
                    tag=content["toolUse"]["toolUseId"],
                    name=name,
                    path=path,
                    input=content["toolUse"]["input"],
                ),
            ),
        )
    if "reasoningContent" in content:
        return TextMessageContent(TextArtifact(content["reasoningContent"][
↪"reasoningText"]["text"]))
    raise ValueError(f"Unsupported message content type: {content}")
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    system_messages = [{"text": message.to_text()} for message in prompt_stack.system_
↪messages]
    messages = self.__to_bedrock_messages([message for message in prompt_stack.
↪messages if not message.is_system()])
```

(continues on next page)

(continued from previous page)

```
    params = {
        "modelId": self.model,
        "messages": messages,
        "system": system_messages,
        "inferenceConfig": {
            "temperature": self.temperature,
            **({"maxTokens": self.max_tokens} if self.max_tokens is not None else {}),
        },
        "additionalModelRequestFields": self.additional_model_request_fields,
        **self.extra_params,
    }

    if prompt_stack.tools and self.use_native_tools:
        params["toolConfig"] = {
            "tools": [],
            "toolChoice": self.tool_choice,
        }

        if prompt_stack.output_schema is not None and self.structured_output_strategy␣
↪== "tool":
            params["toolConfig"]["toolChoice"] = {"any": {}}

        params["toolConfig"]["tools"] = self.__to_bedrock_tools(prompt_stack.tools)

    return params
```

client()

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
@lazy_property()
def client(self) -> Any:
    return self.session.client("bedrock-runtime")
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    params = self._base_params(prompt_stack)
    logger.debug(params)
    response = self.client.converse(**params)
    logger.debug(response)

    usage = response["usage"]
    output_message = response["output"]["message"]

    message_content = output_message.get("content", [])
```

(continues on next page)

(continued from previous page)

```
    # Move reasoning content to the beginning of the content to have it appear first␣
↪in output
    reasoning_content_index = next(
        (i for i, content in enumerate(message_content) if "reasoningContent" in␣
↪content),
        None,
    )
    if reasoning_content_index is not None:
        message_content.insert(0, message_content.pop(reasoning_content_index))

    return Message(
        content=[self.__to_prompt_stack_message_content(content) for content in␣
↪message_content],
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(input_tokens=usage["inputTokens"], output_tokens=usage[
↪"outputTokens"]),
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    params = self._base_params(prompt_stack)
    logger.debug(params)
    response = self.client.converse_stream(**params)

    stream = response.get("stream")
    if stream is not None:
        for event in stream:
            logger.debug(event)
            if "contentBlockDelta" in event or "contentBlockStart" in event:
                yield DeltaMessage(content=self.__to_prompt_stack_delta_message_
↪content(event))
            elif "metadata" in event:
                usage = event["metadata"]["usage"]
                yield DeltaMessage(
                    usage=DeltaMessage.Usage(
                        input_tokens=usage["inputTokens"],
                        output_tokens=usage["outputTokens"],
                    ),
                )
    else:
        raise Exception("model response is empty")
```

validate*structured_output_strategy(*, value)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
@structured_output_strategy.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
    if value == "native":
        raise ValueError(f"{__class__.__name__} does not support `{value}` structured␣
↪output strategy.")

    return value
```

amazon_bedrock_prompt_driver

- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *BasePromptDriver* “)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
@define
class AmazonBedrockPromptDriver(BasePromptDriver):
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    additional_model_request_fields: dict = field(default=Factory(dict), kw_only=True)
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: AmazonBedrockTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
    use_native_tools: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})
    structured_output_strategy: StructuredOutputStrategy = field(
        default="tool", kw_only=True, metadata={"serializable": True}
    )
    tool_choice: dict = field(default=Factory(lambda: {"auto": {}}), kw_only=True,␣
↪metadata={"serializable": True})
    _client: Optional[Any] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @structured_output_strategy.validator  # pyright:␣
↪ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
        if value == "native":
            raise ValueError(f"{__class__.__name__} does not support `{value}`␣
↪structured output strategy.")

        return value

    @lazy_property()
    def client(self) -> Any:
        return self.session.client("bedrock-runtime")

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        params = self._base_params(prompt_stack)
```

(continues on next page)

(continued from previous page)

```
        logger.debug(params)
        response = self.client.converse(**params)
        logger.debug(response)

        usage = response["usage"]
        output_message = response["output"]["message"]

        message_content = output_message.get("content", [])

        # Move reasoning content to the beginning of the content to have it appear␣
↪first in output
        reasoning_content_index = next(
            (i for i, content in enumerate(message_content) if "reasoningContent" in␣
↪content),
            None,
        )
        if reasoning_content_index is not None:
            message_content.insert(0, message_content.pop(reasoning_content_index))

        return Message(
            content=[self.__to_prompt_stack_message_content(content) for content in␣
↪message_content],
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(input_tokens=usage["inputTokens"], output_
↪tokens=usage["outputTokens"]),
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        params = self._base_params(prompt_stack)
        logger.debug(params)
        response = self.client.converse_stream(**params)

        stream = response.get("stream")
        if stream is not None:
            for event in stream:
                logger.debug(event)
                if "contentBlockDelta" in event or "contentBlockStart" in event:
                    yield DeltaMessage(content=self.__to_prompt_stack_delta_message_
↪content(event))
                elif "metadata" in event:
                    usage = event["metadata"]["usage"]
                    yield DeltaMessage(
                        usage=DeltaMessage.Usage(
                            input_tokens=usage["inputTokens"],
                            output_tokens=usage["outputTokens"],
                        ),
                    )
        else:
            raise Exception("model response is empty")

    def _base_params(self, prompt_stack: PromptStack) -> dict:
```

(continues on next page)

(continued from previous page)

```
        system_messages = [{"text": message.to_text()} for message in prompt_stack.
↪system_messages]
        messages = self.__to_bedrock_messages([message for message in prompt_stack.
↪messages if not message.is_system()])

        params = {
            "modelId": self.model,
            "messages": messages,
            "system": system_messages,
            "inferenceConfig": {
                "temperature": self.temperature,
                **({"maxTokens": self.max_tokens} if self.max_tokens is not None else
↪{}),
            },
            "additionalModelRequestFields": self.additional_model_request_fields,
            **self.extra_params,
        }

        if prompt_stack.tools and self.use_native_tools:
            params["toolConfig"] = {
                "tools": [],
                "toolChoice": self.tool_choice,
            }

            if prompt_stack.output_schema is not None and self.structured_output_
↪strategy == "tool":
                params["toolConfig"]["toolChoice"] = {"any": {}}

            params["toolConfig"]["tools"] = self.__to_bedrock_tools(prompt_stack.
↪tools)

        return params

    def __to_bedrock_messages(self, messages: list[Message]) -> list[dict]:
        return [
            {
                "role": self.__to_bedrock_role(message),
                "content": [self.__to_bedrock_message_content(content) for content in␣
↪message.content],
            }
            for message in messages
        ]

    def __to_bedrock_role(self, message: Message) -> str:
        if message.is_assistant():
            return "assistant"
        return "user"

    def __to_bedrock_tools(self, tools: list[BaseTool]) -> list[dict]:
        return [
            {
                "toolSpec": {
```

(continues on next page)

(continued from previous page)

```
                    "name": tool.to_native_tool_name(activity),
                    "description": tool.activity_description(activity),
                    "inputSchema": {
                        "json": tool.to_activity_json_schema(activity, "http://json-
↪schema.org/draft-07/schema#"),
                    },
                },
            }
            for tool in tools
            for activity in tool.activities()
        ]

    def __to_bedrock_message_content(self, content: BaseMessageContent) -> dict:
        if isinstance(content, TextMessageContent):
            return {"text": content.artifact.to_text()}
        if isinstance(content, ImageMessageContent):
            artifact = content.artifact
            if isinstance(artifact, ImageArtifact):
                return {"image": {"format": artifact.format, "source": {"bytes":␣
↪artifact.value}}}
            if isinstance(artifact, ImageUrlArtifact):
                return {"image": {"format": "png", "source": {"s3Location": {"uri":␣
↪artifact.value}}}}
            raise ValueError(f"Unsupported image artifact type: {type(artifact)}")
        if isinstance(content, ActionCallMessageContent):
            action_call = content.artifact.value

            return {
                "toolUse": {
                    "toolUseId": action_call.tag,
                    "name": f"{action_call.name}_{action_call.path}",
                    "input": action_call.input,
                },
            }
        if isinstance(content, ActionResultMessageContent):
            artifact = content.artifact

            if isinstance(artifact, ListArtifact):
                message_content = [self.__to_bedrock_tool_use_content(artifact) for␣
↪artifact in artifact.value]
            else:
                message_content = [self.__to_bedrock_tool_use_content(artifact)]

            return {
                "toolResult": {
                    "toolUseId": content.action.tag,
                    "content": message_content,
                    "status": "error" if isinstance(artifact, ErrorArtifact) else
↪"success",
                },
            }
        return content.artifact.value
```

(continues on next page)

(continued from previous page)

```
    def __to_bedrock_tool_use_content(self, artifact: BaseArtifact) -> dict:
        if isinstance(artifact, ImageArtifact):
            return {"image": {"format": artifact.format, "source": {"bytes": artifact.
↪value}}}
        return {"text": artifact.to_text()}

    def __to_prompt_stack_message_content(self, content: dict) -> BaseMessageContent:
        if "text" in content:
            return TextMessageContent(TextArtifact(content["text"]))
        if "toolUse" in content:
            name, path = ToolAction.from_native_tool_name(content["toolUse"]["name"])
            return ActionCallMessageContent(
                artifact=ActionArtifact(
                    value=ToolAction(
                        tag=content["toolUse"]["toolUseId"],
                        name=name,
                        path=path,
                        input=content["toolUse"]["input"],
                    ),
                ),
            )
        if "reasoningContent" in content:
            return TextMessageContent(TextArtifact(content["reasoningContent"][
↪"reasoningText"]["text"]))
        raise ValueError(f"Unsupported message content type: {content}")

    def __to_prompt_stack_delta_message_content(self, event: dict) ->␣
↪BaseDeltaMessageContent:
        if "contentBlockStart" in event:
            content_block = event["contentBlockStart"]["start"]

            if "toolUse" in content_block:
                name, path = ToolAction.from_native_tool_name(content_block["toolUse
↪"]["name"])

                return ActionCallDeltaMessageContent(
                    index=event["contentBlockStart"]["contentBlockIndex"],
                    tag=content_block["toolUse"]["toolUseId"],
                    name=name,
                    path=path,
                )
            if "text" in content_block:
                return TextDeltaMessageContent(
                    content_block["text"],
                    index=event["contentBlockStart"]["contentBlockIndex"],
                )
            raise ValueError(f"Unsupported message content type: {event}")
        if "contentBlockDelta" in event:
            content_block_delta = event["contentBlockDelta"]

            if "text" in content_block_delta["delta"]:
```

(continues on next page)

(continued from previous page)

```
                return TextDeltaMessageContent(
                    content_block_delta["delta"]["text"],
                    index=content_block_delta["contentBlockIndex"],
                )
            if "toolUse" in content_block_delta["delta"]:
                return ActionCallDeltaMessageContent(
                    index=content_block_delta["contentBlockIndex"],
                    partial_input=content_block_delta["delta"]["toolUse"]["input"],
                )
            raise ValueError(f"Unsupported message content type: {event}")
        raise ValueError(f"Unsupported message content type: {event}")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `additional_model_request_fields = field(default=Factory(dict), kw_only=True)` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `structured_output_strategy = field(default='tool', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: AmazonBedrockTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `tool_choice = field(default=Factory(lambda: {'auto': {}}), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `use_native_tools = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__to_bedrock_message_content(content)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_bedrock_message_content(self, content: BaseMessageContent) -> dict:
    if isinstance(content, TextMessageContent):
        return {"text": content.artifact.to_text()}
    if isinstance(content, ImageMessageContent):
        artifact = content.artifact
        if isinstance(artifact, ImageArtifact):
            return {"image": {"format": artifact.format, "source": {"bytes": artifact.
↪value}}}
        if isinstance(artifact, ImageUrlArtifact):
            return {"image": {"format": "png", "source": {"s3Location": {"uri":␣
↪artifact.value}}}}
        raise ValueError(f"Unsupported image artifact type: {type(artifact)}")
    if isinstance(content, ActionCallMessageContent):
        action_call = content.artifact.value

        return {
            "toolUse": {
                "toolUseId": action_call.tag,
```

(continues on next page)

(continued from previous page)

```
                "name": f"{action_call.name}_{action_call.path}",
                "input": action_call.input,
            },
        }
    if isinstance(content, ActionResultMessageContent):
        artifact = content.artifact

        if isinstance(artifact, ListArtifact):
            message_content = [self.__to_bedrock_tool_use_content(artifact) for␣
↪artifact in artifact.value]
        else:
            message_content = [self.__to_bedrock_tool_use_content(artifact)]

        return {
            "toolResult": {
                "toolUseId": content.action.tag,
                "content": message_content,
                "status": "error" if isinstance(artifact, ErrorArtifact) else "success
↪",
            },
        }
    return content.artifact.value
```

__to_bedrock_messages(messages)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_bedrock_messages(self, messages: list[Message]) -> list[dict]:
    return [
        {
            "role": self.__to_bedrock_role(message),
            "content": [self.__to_bedrock_message_content(content) for content in␣
↪message.content],
        }
        for message in messages
    ]
```

__to_bedrock_role(message)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_bedrock_role(self, message: Message) -> str:
    if message.is_assistant():
        return "assistant"
    return "user"
```

__to_bedrock_tool_use_content(artifact)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_bedrock_tool_use_content(self, artifact: BaseArtifact) -> dict:
    if isinstance(artifact, ImageArtifact):
        return {"image": {"format": artifact.format, "source": {"bytes": artifact.
↪value}}}
    return {"text": artifact.to_text()}
```

__to_bedrock_tools(tools)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_bedrock_tools(self, tools: list[BaseTool]) -> list[dict]:
    return [
        {
            "toolSpec": {
                "name": tool.to_native_tool_name(activity),
                "description": tool.activity_description(activity),
                "inputSchema": {
                    "json": tool.to_activity_json_schema(activity, "http://json-
↪schema.org/draft-07/schema#"),
                },
            },
        }
        for tool in tools
        for activity in tool.activities()
    ]
```

__to_prompt_stack_delta_message_content(event)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_prompt_stack_delta_message_content(self, event: dict) ->␣
↪BaseDeltaMessageContent:
    if "contentBlockStart" in event:
        content_block = event["contentBlockStart"]["start"]

        if "toolUse" in content_block:
            name, path = ToolAction.from_native_tool_name(content_block["toolUse"][
↪"name"])

            return ActionCallDeltaMessageContent(
                index=event["contentBlockStart"]["contentBlockIndex"],
                tag=content_block["toolUse"]["toolUseId"],
                name=name,
                path=path,
            )
        if "text" in content_block:
            return TextDeltaMessageContent(
                content_block["text"],
                index=event["contentBlockStart"]["contentBlockIndex"],
            )
        raise ValueError(f"Unsupported message content type: {event}")
    if "contentBlockDelta" in event:
```

(continues on next page)

(continued from previous page)

```
        content_block_delta = event["contentBlockDelta"]

        if "text" in content_block_delta["delta"]:
            return TextDeltaMessageContent(
                content_block_delta["delta"]["text"],
                index=content_block_delta["contentBlockIndex"],
            )
        if "toolUse" in content_block_delta["delta"]:
            return ActionCallDeltaMessageContent(
                index=content_block_delta["contentBlockIndex"],
                partial_input=content_block_delta["delta"]["toolUse"]["input"],
            )
        raise ValueError(f"Unsupported message content type: {event}")
    raise ValueError(f"Unsupported message content type: {event}")
```

__to_prompt_stack_message_content(content)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def __to_prompt_stack_message_content(self, content: dict) -> BaseMessageContent:
    if "text" in content:
        return TextMessageContent(TextArtifact(content["text"]))
    if "toolUse" in content:
        name, path = ToolAction.from_native_tool_name(content["toolUse"]["name"])
        return ActionCallMessageContent(
            artifact=ActionArtifact(
                value=ToolAction(
                    tag=content["toolUse"]["toolUseId"],
                    name=name,
                    path=path,
                    input=content["toolUse"]["input"],
                ),
            ),
        )
    if "reasoningContent" in content:
        return TextMessageContent(TextArtifact(content["reasoningContent"][
↪"reasoningText"]["text"]))
    raise ValueError(f"Unsupported message content type: {content}")
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    system_messages = [{"text": message.to_text()} for message in prompt_stack.system_
↪messages]
    messages = self.__to_bedrock_messages([message for message in prompt_stack.
↪messages if not message.is_system()])

    params = {
        "modelId": self.model,
```

(continues on next page)

(continued from previous page)

```
        "messages": messages,
        "system": system_messages,
        "inferenceConfig": {
            "temperature": self.temperature,
            **({"maxTokens": self.max_tokens} if self.max_tokens is not None else {}),
        },
        "additionalModelRequestFields": self.additional_model_request_fields,
        **self.extra_params,
    }

    if prompt_stack.tools and self.use_native_tools:
        params["toolConfig"] = {
            "tools": [],
            "toolChoice": self.tool_choice,
        }

        if prompt_stack.output_schema is not None and self.structured_output_strategy
↪== "tool":
            params["toolConfig"]["toolChoice"] = {"any": {}}

        params["toolConfig"]["tools"] = self.__to_bedrock_tools(prompt_stack.tools)

    return params
```

client()

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
@lazy_property()
def client(self) -> Any:
    return self.session.client("bedrock-runtime")
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    params = self._base_params(prompt_stack)
    logger.debug(params)
    response = self.client.converse(**params)
    logger.debug(response)

    usage = response["usage"]
    output_message = response["output"]["message"]

    message_content = output_message.get("content", [])

    # Move reasoning content to the beginning of the content to have it appear first
↪in output
    reasoning_content_index = next(
```

(continues on next page)

(continued from previous page)

```
        (i for i, content in enumerate(message_content) if "reasoningContent" in␣
↪content),
        None,
    )
    if reasoning_content_index is not None:
        message_content.insert(0, message_content.pop(reasoning_content_index))

    return Message(
        content=[self.__to_prompt_stack_message_content(content) for content in␣
↪message_content],
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(input_tokens=usage["inputTokens"], output_tokens=usage[
↪"outputTokens"]),
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    params = self._base_params(prompt_stack)
    logger.debug(params)
    response = self.client.converse_stream(**params)

    stream = response.get("stream")
    if stream is not None:
        for event in stream:
            logger.debug(event)
            if "contentBlockDelta" in event or "contentBlockStart" in event:
                yield DeltaMessage(content=self.__to_prompt_stack_delta_message_
↪content(event))
            elif "metadata" in event:
                usage = event["metadata"]["usage"]
                yield DeltaMessage(
                    usage=DeltaMessage.Usage(
                        input_tokens=usage["inputTokens"],
                        output_tokens=usage["outputTokens"],
                    ),
                )
    else:
        raise Exception("model response is empty")
```

validate*structured_output_strategy(*, value)

Source Code in griptape/drivers/prompt/amazon_bedrock_prompt_driver.py

```
@structured_output_strategy.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
    if value == "native":
```

(continues on next page)

(continued from previous page)

```
        raise ValueError(f"{__class__.__name__} does not support `{value}` structured␣
↪output strategy.")

    return value
```

amazon_sagemaker_jumpstart

- `__all__ = ['AmazonSageMakerJumpstartPromptDriver']` module-attribute

Bases: *BasePromptDriver* “)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
@define
class AmazonSageMakerJumpstartPromptDriver(BasePromptDriver):
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    endpoint: str = field(kw_only=True, metadata={"serializable": True})
    custom_attributes: str = field(default="accept_eula=true", kw_only=True, metadata=
↪{"serializable": True})
    inference_component_name: Optional[str] = field(default=None, kw_only=True,␣
↪metadata={"serializable": True})
    stream: bool = field(default=False, kw_only=True, metadata={"serializable": True})
    max_tokens: int = field(default=250, kw_only=True, metadata={"serializable": True}
↪)
    tokenizer: HuggingFaceTokenizer = field(
        default=Factory(
            lambda self: HuggingFaceTokenizer(model=self.model, max_output_
↪tokens=self.max_tokens),
            takes_self=True,
        ),
        kw_only=True,
    )
    structured_output_strategy: StructuredOutputStrategy = field(
        default="rule", kw_only=True, metadata={"serializable": True}
    )
    _client: Optional[Any] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @structured_output_strategy.validator  # pyright:␣
↪ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
        if value != "rule":
            raise ValueError(f"{__class__.__name__} does not support `{value}`␣
↪structured output strategy.")

        return value

    @lazy_property()
    def client(self) -> Any:
        return self.session.client("sagemaker-runtime")

    @stream.validator  # pyright: ignore[reportAttributeAccessIssue]
```

(continues on next page)

(continued from previous page)

```
    def validate_stream(self, _: Attribute, stream: bool) -> None:  # noqa: FBT001
        if stream:
            raise ValueError("streaming is not supported")

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        payload = {
            "inputs": self.prompt_stack_to_string(prompt_stack),
            "parameters": {**self._base_params(prompt_stack)},
        }
        logger.debug(payload)

        response = self.client.invoke_endpoint(
            EndpointName=self.endpoint,
            ContentType="application/json",
            Body=json.dumps(payload),
            CustomAttributes=self.custom_attributes,
            **(
                {"InferenceComponentName": self.inference_component_name}
                if self.inference_component_name is not None
                else {}
            ),
        )

        decoded_body = json.loads(response["Body"].read().decode("utf8"))
        logger.debug(decoded_body)

        if isinstance(decoded_body, list):
            if decoded_body:
                generated_text = decoded_body[0]["generated_text"]
            else:
                raise ValueError("model response is empty")
        else:
            generated_text = decoded_body["generated_text"]

        input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))
        output_tokens = len(self.tokenizer.tokenizer.encode(generated_text))  #␣
↪pyright: ignore[reportArgumentType]

        return Message(
            content=[TextMessageContent(TextArtifact(generated_text))],
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(input_tokens=input_tokens, output_tokens=output_
↪tokens),
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        raise NotImplementedError("streaming is not supported")

    def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
        return self.tokenizer.tokenizer.decode(self.__prompt_stack_to_tokens(prompt_
```

(continues on next page)

(continued from previous page)

```
↪stack))  # pyright: ignore[reportArgumentType]

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        return {
            "temperature": self.temperature,
            "max_new_tokens": self.max_tokens,
            "do_sample": True,
            "eos_token_id": self.tokenizer.tokenizer.eos_token_id,
            "stop_strings": self.tokenizer.stop_sequences,
            "return_full_text": False,
            **self.extra_params,
        }

    def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
        messages = []

        for message in prompt_stack.messages:
            messages.append({"role": message.role, "content": message.to_text()})

        return messages

    def __prompt_stack_to_tokens(self, prompt_stack: PromptStack) -> list[int]:
        messages = self._prompt_stack_to_messages(prompt_stack)

        tokens = self.tokenizer.tokenizer.apply_chat_template(messages, add_
↪generation_prompt=True, tokenize=True)

        if isinstance(tokens, list):
            return tokens  # pyright: ignore[reportReturnType] According to the␣
↪[docs](https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils
↪#transformers.PreTrainedTokenizerBase.apply_chat_template), the return type is␣
↪List[int].
        raise ValueError("Invalid output type.")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `custom_attributes = field(default='accept_eula=true', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `inference_component_name = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `max_tokens = field(default=250, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `stream = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='rule', kw_only=True,`

`metadata={'serializable': True})` class-attribute instance-attribute

- `tokenizer = field(default=Factory(lambda self: HuggingFaceTokenizer(model=self.model, max_output_tokens=self.max_tokens), takes_self=True), kw_only=True)` class-attribute instance-attribute

__prompt_stack_to_tokens(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
def __prompt_stack_to_tokens(self, prompt_stack: PromptStack) -> list[int]:
    messages = self._prompt_stack_to_messages(prompt_stack)

    tokens = self.tokenizer.tokenizer.apply_chat_template(messages, add_generation_
↪prompt=True, tokenize=True)

    if isinstance(tokens, list):
        return tokens  # pyright: ignore[reportReturnType] According to the␣
↪[docs](https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils
↪#transformers.PreTrainedTokenizerBase.apply_chat_template), the return type is␣
↪List[int].
    raise ValueError("Invalid output type.")
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    return {
        "temperature": self.temperature,
        "max_new_tokens": self.max_tokens,
        "do_sample": True,
        "eos_token_id": self.tokenizer.tokenizer.eos_token_id,
        "stop_strings": self.tokenizer.stop_sequences,
        "return_full_text": False,
        **self.extra_params,
    }
```

_prompt_stack_to_messages(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
    messages = []

    for message in prompt_stack.messages:
        messages.append({"role": message.role, "content": message.to_text()})

    return messages
```

client()

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
@lazy_property()
def client(self) -> Any:
    return self.session.client("sagemaker-runtime")
```

prompt_stack_to_string(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
    return self.tokenizer.tokenizer.decode(self.__prompt_stack_to_tokens(prompt_
↪stack))  # pyright: ignore[reportArgumentType]
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    payload = {
        "inputs": self.prompt_stack_to_string(prompt_stack),
        "parameters": {**self._base_params(prompt_stack)},
    }
    logger.debug(payload)

    response = self.client.invoke_endpoint(
        EndpointName=self.endpoint,
        ContentType="application/json",
        Body=json.dumps(payload),
        CustomAttributes=self.custom_attributes,
        **(
            {"InferenceComponentName": self.inference_component_name}
            if self.inference_component_name is not None
            else {}
        ),
    )

    decoded_body = json.loads(response["Body"].read().decode("utf8"))
    logger.debug(decoded_body)

    if isinstance(decoded_body, list):
        if decoded_body:
            generated_text = decoded_body[0]["generated_text"]
        else:
            raise ValueError("model response is empty")
    else:
        generated_text = decoded_body["generated_text"]

    input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))
    output_tokens = len(self.tokenizer.tokenizer.encode(generated_text))  # pyright:␣
```

(continues on next page)

(continued from previous page)

```
↪ignore[reportArgumentType]

    return Message(
        content=[TextMessageContent(TextArtifact(generated_text))],
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(input_tokens=input_tokens, output_tokens=output_tokens),
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    raise NotImplementedError("streaming is not supported")
```

validate*stream(*, stream)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
@stream.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_stream(self, _: Attribute, stream: bool) -> None:  # noqa: FBT001
    if stream:
        raise ValueError("streaming is not supported")
```

validate*structured_output_strategy(*, value)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
@structured_output_strategy.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
    if value != "rule":
        raise ValueError(f"{__class__.__name__} does not support `{value}` structured␣
↪output strategy.")

    return value
```

amazon_sagemaker_jumpstart_prompt_driver

- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *BasePromptDriver* “)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
@define
class AmazonSageMakerJumpstartPromptDriver(BasePromptDriver):
    session: boto3.Session = field(default=Factory(lambda: import_optional_dependency(
↪"boto3").Session()), kw_only=True)
    endpoint: str = field(kw_only=True, metadata={"serializable": True})
    custom_attributes: str = field(default="accept_eula=true", kw_only=True, metadata=
```

(continues on next page)

(continued from previous page)

```
↪{"serializable": True})
    inference_component_name: Optional[str] = field(default=None, kw_only=True,␣
↪metadata={"serializable": True})
    stream: bool = field(default=False, kw_only=True, metadata={"serializable": True})
    max_tokens: int = field(default=250, kw_only=True, metadata={"serializable": True}
↪)
    tokenizer: HuggingFaceTokenizer = field(
        default=Factory(
            lambda self: HuggingFaceTokenizer(model=self.model, max_output_
↪tokens=self.max_tokens),
            takes_self=True,
        ),
        kw_only=True,
    )
    structured_output_strategy: StructuredOutputStrategy = field(
        default="rule", kw_only=True, metadata={"serializable": True}
    )
    _client: Optional[Any] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @structured_output_strategy.validator  # pyright:␣
↪ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
        if value != "rule":
            raise ValueError(f"{__class__.__name__} does not support `{value}`␣
↪structured output strategy.")

        return value

    @lazy_property()
    def client(self) -> Any:
        return self.session.client("sagemaker-runtime")

    @stream.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_stream(self, _: Attribute, stream: bool) -> None:  # noqa: FBT001
        if stream:
            raise ValueError("streaming is not supported")

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        payload = {
            "inputs": self.prompt_stack_to_string(prompt_stack),
            "parameters": {**self._base_params(prompt_stack)},
        }
        logger.debug(payload)

        response = self.client.invoke_endpoint(
            EndpointName=self.endpoint,
            ContentType="application/json",
            Body=json.dumps(payload),
            CustomAttributes=self.custom_attributes,
            **(
```

(continues on next page)

(continued from previous page)

```
                {"InferenceComponentName": self.inference_component_name}
                if self.inference_component_name is not None
                else {}
            ),
        )

        decoded_body = json.loads(response["Body"].read().decode("utf8"))
        logger.debug(decoded_body)

        if isinstance(decoded_body, list):
            if decoded_body:
                generated_text = decoded_body[0]["generated_text"]
            else:
                raise ValueError("model response is empty")
        else:
            generated_text = decoded_body["generated_text"]

        input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))
        output_tokens = len(self.tokenizer.tokenizer.encode(generated_text))  #␣
↪pyright: ignore[reportArgumentType]

        return Message(
            content=[TextMessageContent(TextArtifact(generated_text))],
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(input_tokens=input_tokens, output_tokens=output_
↪tokens),
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        raise NotImplementedError("streaming is not supported")

    def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
        return self.tokenizer.tokenizer.decode(self.__prompt_stack_to_tokens(prompt_
↪stack))  # pyright: ignore[reportArgumentType]

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        return {
            "temperature": self.temperature,
            "max_new_tokens": self.max_tokens,
            "do_sample": True,
            "eos_token_id": self.tokenizer.tokenizer.eos_token_id,
            "stop_strings": self.tokenizer.stop_sequences,
            "return_full_text": False,
            **self.extra_params,
        }

    def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
        messages = []

        for message in prompt_stack.messages:
            messages.append({"role": message.role, "content": message.to_text()})
```

(continues on next page)

(continued from previous page)

```
        return messages

    def __prompt_stack_to_tokens(self, prompt_stack: PromptStack) -> list[int]:
        messages = self._prompt_stack_to_messages(prompt_stack)

        tokens = self.tokenizer.tokenizer.apply_chat_template(messages, add_
↪generation_prompt=True, tokenize=True)

        if isinstance(tokens, list):
            return tokens  # pyright: ignore[reportReturnType] According to the␣
↪[docs](https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils
↪#transformers.PreTrainedTokenizerBase.apply_chat_template), the return type is␣
↪List[int].
        raise ValueError("Invalid output type.")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `custom_attributes = field(default='accept_eula=true', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `inference_component_name = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `max_tokens = field(default=250, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute
- `stream = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='rule', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: HuggingFaceTokenizer(model=self.model, max_output_tokens=self.max_tokens), takes_self=True), kw_only=True)` class-attribute instance-attribute

__prompt_stack_to_tokens(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
def __prompt_stack_to_tokens(self, prompt_stack: PromptStack) -> list[int]:
    messages = self._prompt_stack_to_messages(prompt_stack)

    tokens = self.tokenizer.tokenizer.apply_chat_template(messages, add_generation_
↪prompt=True, tokenize=True)

    if isinstance(tokens, list):
        return tokens  # pyright: ignore[reportReturnType] According to the␣
```

(continues on next page)

(continued from previous page)

```
↪[docs](https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils
↪#transformers.PreTrainedTokenizerBase.apply_chat_template), the return type is␣
↪List[int].
    raise ValueError("Invalid output type.")
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    return {
        "temperature": self.temperature,
        "max_new_tokens": self.max_tokens,
        "do_sample": True,
        "eos_token_id": self.tokenizer.tokenizer.eos_token_id,
        "stop_strings": self.tokenizer.stop_sequences,
        "return_full_text": False,
        **self.extra_params,
    }
```

_prompt_stack_to_messages(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
    messages = []

    for message in prompt_stack.messages:
        messages.append({"role": message.role, "content": message.to_text()})

    return messages
```

client()

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
@lazy_property()
def client(self) -> Any:
    return self.session.client("sagemaker-runtime")
```

prompt_stack_to_string(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
    return self.tokenizer.tokenizer.decode(self.__prompt_stack_to_tokens(prompt_
↪stack))  # pyright: ignore[reportArgumentType]
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    payload = {
        "inputs": self.prompt_stack_to_string(prompt_stack),
        "parameters": {**self._base_params(prompt_stack)},
    }
    logger.debug(payload)

    response = self.client.invoke_endpoint(
        EndpointName=self.endpoint,
        ContentType="application/json",
        Body=json.dumps(payload),
        CustomAttributes=self.custom_attributes,
        **(
            {"InferenceComponentName": self.inference_component_name}
            if self.inference_component_name is not None
            else {}
        ),
    )

    decoded_body = json.loads(response["Body"].read().decode("utf8"))
    logger.debug(decoded_body)

    if isinstance(decoded_body, list):
        if decoded_body:
            generated_text = decoded_body[0]["generated_text"]
        else:
            raise ValueError("model response is empty")
    else:
        generated_text = decoded_body["generated_text"]

    input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))
    output_tokens = len(self.tokenizer.tokenizer.encode(generated_text))  # pyright:␣
↪ignore[reportArgumentType]

    return Message(
        content=[TextMessageContent(TextArtifact(generated_text))],
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(input_tokens=input_tokens, output_tokens=output_tokens),
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    raise NotImplementedError("streaming is not supported")
```

validate*stream*(, stream)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
@stream.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_stream(self, _: Attribute, stream: bool) -> None:  # noqa: FBT001
    if stream:
        raise ValueError("streaming is not supported")
```

validate*structured_output_strategy*(, value)

Source Code in griptape/drivers/prompt/amazon_sagemaker_jumpstart_prompt_driver.py

```
@structured_output_strategy.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
    if value != "rule":
        raise ValueError(f"{__class__.__name__} does not support `{value}` structured␣
↪output strategy.")

    return value
```

anthropic

- `__all__ = ['AnthropicPromptDriver']` module-attribute

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="anthropic.mdx#griptape.drivers.prompt.anthropic.AnthropicPromptDriver.api_key" title="api_key = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.prompt.anthropic.AnthropicPromptDriver.api_key)"><code class="language-python">	*Optional[str]*	Anthropic API key.

continues on next page

Table 191 – continued from previous page

Name	Type	Description
<a href="anthropic.mdx#griptape.drivers.prompt.anthropic.AnthropicPromptDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.anthropic.AnthropicPromptDriver.model)"><code class="language-python">	*str*	Anthropic model name.
<a href="anthropic.mdx#griptape.drivers.prompt.anthropic.AnthropicPromptDriver.client" title="client() (griptape.drivers.prompt.anthropic.AnthropicPromptDriver.client)"><code class="language-python">	*Client*	Custom *Anthropic* client.

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
@define
class AnthropicPromptDriver(BasePromptDriver):
    """Anthropic Prompt Driver.

    Attributes:
        api_key: Anthropic API key.
        model: Anthropic model name.
        client: Custom `Anthropic` client.
    """

    api_key: Optional[str] = field(kw_only=True, default=None, metadata={"serializable
↪": False})
    model: str = field(kw_only=True, metadata={"serializable": True})
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: AnthropicTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
    top_p: float = field(default=0.999, kw_only=True, metadata={"serializable": True})
    top_k: int = field(default=250, kw_only=True, metadata={"serializable": True})
```

(continues on next page)

(continued from previous page)

```
    tool_choice: dict = field(default=Factory(lambda: {"type": "auto"}), kw_only=True,
↪ metadata={"serializable": False})
    use_native_tools: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})
    structured_output_strategy: StructuredOutputStrategy = field(
        default="tool", kw_only=True, metadata={"serializable": True}
    )
    max_tokens: int = field(default=1000, kw_only=True, metadata={"serializable":␣
↪True})
    _client: Optional[Client] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> Client:
        return import_optional_dependency("anthropic").Anthropic(api_key=self.api_key)

    @structured_output_strategy.validator  # pyright:␣
↪ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
        if value == "native":
            raise ValueError(f"{__class__.__name__} does not support `{value}`␣
↪structured output strategy.")

        return value

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        params = self._base_params(prompt_stack)
        logger.debug(params)
        response = self.client.messages.create(**params)

        logger.debug(response.model_dump())

        return Message(
            content=[self.__to_prompt_stack_message_content(content) for content in␣
↪response.content],
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(input_tokens=response.usage.input_tokens, output_
↪tokens=response.usage.output_tokens),
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        params = {**self._base_params(prompt_stack), "stream": True}
        logger.debug(params)
        events = self.client.messages.create(**params)

        for event in events:
            logger.debug(event)
            if event.type == "content_block_delta" or event.type == "content_block_
↪start":
                yield DeltaMessage(content=self.__to_prompt_stack_delta_message_
```

(continues on next page)

(continued from previous page)

```
↪content(event))
            elif event.type == "message_start":
                yield DeltaMessage(usage=DeltaMessage.Usage(input_tokens=event.
↪message.usage.input_tokens))
            elif event.type == "message_delta":
                yield DeltaMessage(usage=DeltaMessage.Usage(output_tokens=event.usage.
↪output_tokens))

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        messages = self.__to_anthropic_messages([i for i in prompt_stack.messages if␣
↪not i.is_system()])

        system_messages = prompt_stack.system_messages
        system_message = system_messages[0].to_text() if system_messages else None

        params = {
            "model": self.model,
            "temperature": self.temperature,
            "stop_sequences": self.tokenizer.stop_sequences,
            "top_p": self.top_p,
            "top_k": self.top_k,
            "max_tokens": self.max_tokens,
            "messages": messages,
            **({"system": system_message} if system_message else {}),
            **self.extra_params,
        }

        if prompt_stack.tools and self.use_native_tools:
            params["tool_choice"] = self.tool_choice

            if prompt_stack.output_schema is not None and self.structured_output_
↪strategy == "tool":
                params["tool_choice"] = {"type": "any"}

            params["tools"] = self.__to_anthropic_tools(prompt_stack.tools)

        return params

    def __to_anthropic_messages(self, messages: list[Message]) -> list[dict]:
        return [
            {"role": self.__to_anthropic_role(message), "content": self.__to_
↪anthropic_content(message)}
            for message in messages
        ]

    def __to_anthropic_role(self, message: Message) -> str:
        if message.is_assistant():
            return "assistant"
        return "user"

    def __to_anthropic_tools(self, tools: list[BaseTool]) -> list[dict]:
        tool_schemas = [
```

(continues on next page)

(continued from previous page)

```
            {
                "name": tool.to_native_tool_name(activity),
                "description": tool.activity_description(activity),
                "input_schema": tool.to_activity_json_schema(activity, "Input Schema
↪"),
            }
            for tool in tools
            for activity in tool.activities()
        ]

        # Anthropic doesnt support $schema and $id
        for tool_schema in tool_schemas:
            del tool_schema["input_schema"]["$schema"]
            del tool_schema["input_schema"]["$id"]

        return tool_schemas

    def __to_anthropic_content(self, message: Message) -> str | list[dict]:
        if message.has_all_content_type(TextMessageContent):
            return message.to_text()
        return [self.__to_anthropic_message_content(content) for content in message.
↪content]

    def __to_anthropic_message_content(self, content: BaseMessageContent) -> dict:
        if isinstance(content, TextMessageContent):
            return {"type": "text", "text": content.artifact.value}
        if isinstance(content, ImageMessageContent):
            artifact = content.artifact
            if isinstance(artifact, ImageArtifact):
                return {
                    "type": "image",
                    "source": {"type": "base64", "media_type": artifact.mime_type,
↪"data": artifact.base64},
                }
            if isinstance(artifact, ImageUrlArtifact):
                return {
                    "type": "image",
                    "source": {"type": "url", "url": artifact.value},
                }
            raise ValueError(f"Unsupported image artifact type: {type(artifact)}")
        if isinstance(content, ActionCallMessageContent):
            action = content.artifact.value

            return {"type": "tool_use", "id": action.tag, "name": action.to_native_
↪tool_name(), "input": action.input}
        if isinstance(content, ActionResultMessageContent):
            artifact = content.artifact

            if isinstance(artifact, ListArtifact):
                message_content = [self.__to_anthropic_tool_result_content(artifact)␣
↪for artifact in artifact.value]
            else:
```

(continues on next page)

(continued from previous page)

```
                message_content = [self.__to_anthropic_tool_result_content(artifact)]

            return {
                "type": "tool_result",
                "tool_use_id": content.action.tag,
                "content": message_content,
                "is_error": isinstance(artifact, ErrorArtifact),
            }
        return content.artifact.value

    def __to_anthropic_tool_result_content(self, artifact: BaseArtifact) -> dict:
        if isinstance(artifact, ImageArtifact):
            return {
                "type": "image",
                "source": {"type": "base64", "media_type": artifact.mime_type, "data
↪": artifact.base64},
            }
        if isinstance(artifact, (TextArtifact, ErrorArtifact, InfoArtifact)):
            return {"type": "text", "text": artifact.to_text()}
        raise ValueError(f"Unsupported tool result artifact type: {type(artifact)}")

    def __to_prompt_stack_message_content(self, content: ContentBlock) ->␣
↪BaseMessageContent:
        if content.type == "text":
            return TextMessageContent(TextArtifact(content.text))
        if content.type == "tool_use":
            name, path = ToolAction.from_native_tool_name(content.name)

            return ActionCallMessageContent(
                artifact=ActionArtifact(
                    value=ToolAction(tag=content.id, name=name, path=path,␣
↪input=content.input),  # pyright: ignore[reportArgumentType]
                ),
            )
        raise ValueError(f"Unsupported message content type: {content.type}")

    def __to_prompt_stack_delta_message_content(
        self,
        event: ContentBlockDeltaEvent | ContentBlockStartEvent,
    ) -> BaseDeltaMessageContent:
        if event.type == "content_block_start":
            content_block = event.content_block

            if content_block.type == "tool_use":
                name, path = ToolAction.from_native_tool_name(content_block.name)

                return ActionCallDeltaMessageContent(index=event.index, tag=content_
↪block.id, name=name, path=path)
            if content_block.type == "text":
                return TextDeltaMessageContent(content_block.text, index=event.index)
            raise ValueError(f"Unsupported content block type: {content_block.type}")
        if event.type == "content_block_delta":
```

(continues on next page)

(continued from previous page)

```
            content_block_delta = event.delta

            if content_block_delta.type == "text_delta":
                return TextDeltaMessageContent(content_block_delta.text, index=event.
↪index)
            if content_block_delta.type == "input_json_delta":
                return ActionCallDeltaMessageContent(index=event.index, partial_
↪input=content_block_delta.partial_json)
            raise ValueError(f"Unsupported message content type: {event}")
        raise ValueError(f"Unsupported message content type: {event}")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `max_tokens = field(default=1000, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='tool', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: AnthropicTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `tool_choice = field(default=Factory(lambda: {'type': 'auto'}), kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `top_k = field(default=250, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `top_p = field(default=0.999, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `use_native_tools = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__to_anthropic_content(message)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_anthropic_content(self, message: Message) -> str | list[dict]:
    if message.has_all_content_type(TextMessageContent):
        return message.to_text()
    return [self.__to_anthropic_message_content(content) for content in message.
↪content]
```

__to_anthropic_message_content(content)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_anthropic_message_content(self, content: BaseMessageContent) -> dict:
    if isinstance(content, TextMessageContent):
        return {"type": "text", "text": content.artifact.value}
    if isinstance(content, ImageMessageContent):
        artifact = content.artifact
        if isinstance(artifact, ImageArtifact):
            return {
                "type": "image",
                "source": {"type": "base64", "media_type": artifact.mime_type, "data
↪": artifact.base64},
            }
        if isinstance(artifact, ImageUrlArtifact):
            return {
                "type": "image",
                "source": {"type": "url", "url": artifact.value},
            }
        raise ValueError(f"Unsupported image artifact type: {type(artifact)}")
    if isinstance(content, ActionCallMessageContent):
        action = content.artifact.value

        return {"type": "tool_use", "id": action.tag, "name": action.to_native_tool_
↪name(), "input": action.input}
    if isinstance(content, ActionResultMessageContent):
        artifact = content.artifact

        if isinstance(artifact, ListArtifact):
            message_content = [self.__to_anthropic_tool_result_content(artifact) for␣
↪artifact in artifact.value]
        else:
            message_content = [self.__to_anthropic_tool_result_content(artifact)]

        return {
            "type": "tool_result",
            "tool_use_id": content.action.tag,
            "content": message_content,
            "is_error": isinstance(artifact, ErrorArtifact),
        }
    return content.artifact.value
```

__to_anthropic_messages(messages)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_anthropic_messages(self, messages: list[Message]) -> list[dict]:
    return [
        {"role": self.__to_anthropic_role(message), "content": self.__to_anthropic_
↪content(message)}
        for message in messages
    ]
```

__to_anthropic_role(message)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_anthropic_role(self, message: Message) -> str:
    if message.is_assistant():
        return "assistant"
    return "user"
```

__to_anthropic_tool_result_content(artifact)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_anthropic_tool_result_content(self, artifact: BaseArtifact) -> dict:
    if isinstance(artifact, ImageArtifact):
        return {
            "type": "image",
            "source": {"type": "base64", "media_type": artifact.mime_type, "data":␣
↪artifact.base64},
        }
    if isinstance(artifact, (TextArtifact, ErrorArtifact, InfoArtifact)):
        return {"type": "text", "text": artifact.to_text()}
    raise ValueError(f"Unsupported tool result artifact type: {type(artifact)}")
```

__to_anthropic_tools(tools)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_anthropic_tools(self, tools: list[BaseTool]) -> list[dict]:
    tool_schemas = [
        {
            "name": tool.to_native_tool_name(activity),
            "description": tool.activity_description(activity),
            "input_schema": tool.to_activity_json_schema(activity, "Input Schema"),
        }
        for tool in tools
        for activity in tool.activities()
    ]

    # Anthropic doesnt support $schema and $id
    for tool_schema in tool_schemas:
        del tool_schema["input_schema"]["$schema"]
        del tool_schema["input_schema"]["$id"]

    return tool_schemas
```

__to_prompt_stack_delta_message_content(event)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_prompt_stack_delta_message_content(
    self,
    event: ContentBlockDeltaEvent | ContentBlockStartEvent,
```

(continues on next page)

(continued from previous page)

```
) -> BaseDeltaMessageContent:
    if event.type == "content_block_start":
        content_block = event.content_block

        if content_block.type == "tool_use":
            name, path = ToolAction.from_native_tool_name(content_block.name)

            return ActionCallDeltaMessageContent(index=event.index, tag=content_block.
↪id, name=name, path=path)
        if content_block.type == "text":
            return TextDeltaMessageContent(content_block.text, index=event.index)
        raise ValueError(f"Unsupported content block type: {content_block.type}")
    if event.type == "content_block_delta":
        content_block_delta = event.delta

        if content_block_delta.type == "text_delta":
            return TextDeltaMessageContent(content_block_delta.text, index=event.
↪index)
        if content_block_delta.type == "input_json_delta":
            return ActionCallDeltaMessageContent(index=event.index, partial_
↪input=content_block_delta.partial_json)
        raise ValueError(f"Unsupported message content type: {event}")
    raise ValueError(f"Unsupported message content type: {event}")
```

__to_prompt_stack_message_content(content)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_prompt_stack_message_content(self, content: ContentBlock) ->␣
↪BaseMessageContent:
    if content.type == "text":
        return TextMessageContent(TextArtifact(content.text))
    if content.type == "tool_use":
        name, path = ToolAction.from_native_tool_name(content.name)

        return ActionCallMessageContent(
            artifact=ActionArtifact(
                value=ToolAction(tag=content.id, name=name, path=path, input=content.
↪input),  # pyright: ignore[reportArgumentType]
            ),
        )
    raise ValueError(f"Unsupported message content type: {content.type}")
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    messages = self.__to_anthropic_messages([i for i in prompt_stack.messages if not␣
↪i.is_system()])
```

(continues on next page)

(continued from previous page)

```
    system_messages = prompt_stack.system_messages
    system_message = system_messages[0].to_text() if system_messages else None

    params = {
        "model": self.model,
        "temperature": self.temperature,
        "stop_sequences": self.tokenizer.stop_sequences,
        "top_p": self.top_p,
        "top_k": self.top_k,
        "max_tokens": self.max_tokens,
        "messages": messages,
        **({"system": system_message} if system_message else {}),
        **self.extra_params,
    }

    if prompt_stack.tools and self.use_native_tools:
        params["tool_choice"] = self.tool_choice

        if prompt_stack.output_schema is not None and self.structured_output_strategy␣
↪== "tool":
            params["tool_choice"] = {"type": "any"}

        params["tools"] = self.__to_anthropic_tools(prompt_stack.tools)

    return params
```

client()

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
@lazy_property()
def client(self) -> Client:
    return import_optional_dependency("anthropic").Anthropic(api_key=self.api_key)
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    params = self._base_params(prompt_stack)
    logger.debug(params)
    response = self.client.messages.create(**params)

    logger.debug(response.model_dump())

    return Message(
        content=[self.__to_prompt_stack_message_content(content) for content in␣
↪response.content],
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(input_tokens=response.usage.input_tokens, output_
```

(continues on next page)

(continued from previous page)

```
↪tokens=response.usage.output_tokens),
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    params = {**self._base_params(prompt_stack), "stream": True}
    logger.debug(params)
    events = self.client.messages.create(**params)

    for event in events:
        logger.debug(event)
        if event.type == "content_block_delta" or event.type == "content_block_start":
            yield DeltaMessage(content=self.__to_prompt_stack_delta_message_
↪content(event))
        elif event.type == "message_start":
            yield DeltaMessage(usage=DeltaMessage.Usage(input_tokens=event.message.
↪usage.input_tokens))
        elif event.type == "message_delta":
            yield DeltaMessage(usage=DeltaMessage.Usage(output_tokens=event.usage.
↪output_tokens))
```

validate*structured_output_strategy(*, value)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
@structured_output_strategy.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
    if value == "native":
        raise ValueError(f"{__class__.__name__} does not support `{value}` structured␣
↪output strategy.")

    return value
```

anthropic_prompt_driver

- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="anthropic_prompt_driver.mtape.drivers.prompt.anthropic_prompt_driver.AnthropicPromptDriver.api_key" title="api_key = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.prompt.anthropic_prompt_driver.AnthropicPromptDriver.api_key)"><code class="language-python">	*Optional[str]*	Anthropic API key.
<a href="anthropic_prompt_driver.mtape.drivers.prompt.anthropic_prompt_driver.AnthropicPromptDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.anthropic_prompt_driver.AnthropicPromptDriver.model)"><code class="language-python">	*str*	Anthropic model name.
<a href="anthropic_prompt_driver.mtape.drivers.prompt.anthropic_prompt_driver.AnthropicPromptDriver.client" title="client() (griptape.drivers.prompt.anthropic_prompt_driver.AnthropicPromptDriver.client)"><code class="language-python">	*Client*	Custom *Anthropic* client.

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
@define
class AnthropicPromptDriver(BasePromptDriver):
```

(continues on next page)

(continued from previous page)

```
    """Anthropic Prompt Driver.

    Attributes:
        api_key: Anthropic API key.
        model: Anthropic model name.
        client: Custom `Anthropic` client.
    """

    api_key: Optional[str] = field(kw_only=True, default=None, metadata={"serializable
↪": False})
    model: str = field(kw_only=True, metadata={"serializable": True})
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: AnthropicTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
    top_p: float = field(default=0.999, kw_only=True, metadata={"serializable": True})
    top_k: int = field(default=250, kw_only=True, metadata={"serializable": True})
    tool_choice: dict = field(default=Factory(lambda: {"type": "auto"}), kw_only=True,
↪ metadata={"serializable": False})
    use_native_tools: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})
    structured_output_strategy: StructuredOutputStrategy = field(
        default="tool", kw_only=True, metadata={"serializable": True}
    )
    max_tokens: int = field(default=1000, kw_only=True, metadata={"serializable":␣
↪True})
    _client: Optional[Client] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> Client:
        return import_optional_dependency("anthropic").Anthropic(api_key=self.api_key)

    @structured_output_strategy.validator  # pyright:␣
↪ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
        if value == "native":
            raise ValueError(f"{__class__.__name__} does not support `{value}`␣
↪structured output strategy.")

        return value

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        params = self._base_params(prompt_stack)
        logger.debug(params)
        response = self.client.messages.create(**params)

        logger.debug(response.model_dump())

        return Message(
```

(continues on next page)

(continued from previous page)

```
            content=[self.__to_prompt_stack_message_content(content) for content in␣
↪response.content],
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(input_tokens=response.usage.input_tokens, output_
↪tokens=response.usage.output_tokens),
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        params = {**self._base_params(prompt_stack), "stream": True}
        logger.debug(params)
        events = self.client.messages.create(**params)

        for event in events:
            logger.debug(event)
            if event.type == "content_block_delta" or event.type == "content_block_
↪start":
                yield DeltaMessage(content=self.__to_prompt_stack_delta_message_
↪content(event))
            elif event.type == "message_start":
                yield DeltaMessage(usage=DeltaMessage.Usage(input_tokens=event.
↪message.usage.input_tokens))
            elif event.type == "message_delta":
                yield DeltaMessage(usage=DeltaMessage.Usage(output_tokens=event.usage.
↪output_tokens))

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        messages = self.__to_anthropic_messages([i for i in prompt_stack.messages if␣
↪not i.is_system()])

        system_messages = prompt_stack.system_messages
        system_message = system_messages[0].to_text() if system_messages else None

        params = {
            "model": self.model,
            "temperature": self.temperature,
            "stop_sequences": self.tokenizer.stop_sequences,
            "top_p": self.top_p,
            "top_k": self.top_k,
            "max_tokens": self.max_tokens,
            "messages": messages,
            **({"system": system_message} if system_message else {}),
            **self.extra_params,
        }

        if prompt_stack.tools and self.use_native_tools:
            params["tool_choice"] = self.tool_choice

            if prompt_stack.output_schema is not None and self.structured_output_
↪strategy == "tool":
                params["tool_choice"] = {"type": "any"}
```

(continues on next page)

(continued from previous page)

```
            params["tools"] = self.__to_anthropic_tools(prompt_stack.tools)

        return params

    def __to_anthropic_messages(self, messages: list[Message]) -> list[dict]:
        return [
            {"role": self.__to_anthropic_role(message), "content": self.__to_
↪anthropic_content(message)}
            for message in messages
        ]

    def __to_anthropic_role(self, message: Message) -> str:
        if message.is_assistant():
            return "assistant"
        return "user"

    def __to_anthropic_tools(self, tools: list[BaseTool]) -> list[dict]:
        tool_schemas = [
            {
                "name": tool.to_native_tool_name(activity),
                "description": tool.activity_description(activity),
                "input_schema": tool.to_activity_json_schema(activity, "Input Schema
↪"),
            }
            for tool in tools
            for activity in tool.activities()
        ]

        # Anthropic doesnt support $schema and $id
        for tool_schema in tool_schemas:
            del tool_schema["input_schema"]["$schema"]
            del tool_schema["input_schema"]["$id"]

        return tool_schemas

    def __to_anthropic_content(self, message: Message) -> str | list[dict]:
        if message.has_all_content_type(TextMessageContent):
            return message.to_text()
        return [self.__to_anthropic_message_content(content) for content in message.
↪content]

    def __to_anthropic_message_content(self, content: BaseMessageContent) -> dict:
        if isinstance(content, TextMessageContent):
            return {"type": "text", "text": content.artifact.value}
        if isinstance(content, ImageMessageContent):
            artifact = content.artifact
            if isinstance(artifact, ImageArtifact):
                return {
                    "type": "image",
                    "source": {"type": "base64", "media_type": artifact.mime_type,
↪"data": artifact.base64},
                }
```

(continues on next page)

(continued from previous page)

```
            if isinstance(artifact, ImageUrlArtifact):
                return {
                    "type": "image",
                    "source": {"type": "url", "url": artifact.value},
                }
            raise ValueError(f"Unsupported image artifact type: {type(artifact)}")
        if isinstance(content, ActionCallMessageContent):
            action = content.artifact.value

            return {"type": "tool_use", "id": action.tag, "name": action.to_native_
↪tool_name(), "input": action.input}
        if isinstance(content, ActionResultMessageContent):
            artifact = content.artifact

            if isinstance(artifact, ListArtifact):
                message_content = [self.__to_anthropic_tool_result_content(artifact)␣
↪for artifact in artifact.value]
            else:
                message_content = [self.__to_anthropic_tool_result_content(artifact)]

            return {
                "type": "tool_result",
                "tool_use_id": content.action.tag,
                "content": message_content,
                "is_error": isinstance(artifact, ErrorArtifact),
            }
        return content.artifact.value

    def __to_anthropic_tool_result_content(self, artifact: BaseArtifact) -> dict:
        if isinstance(artifact, ImageArtifact):
            return {
                "type": "image",
                "source": {"type": "base64", "media_type": artifact.mime_type, "data
↪": artifact.base64},
            }
        if isinstance(artifact, (TextArtifact, ErrorArtifact, InfoArtifact)):
            return {"type": "text", "text": artifact.to_text()}
        raise ValueError(f"Unsupported tool result artifact type: {type(artifact)}")

    def __to_prompt_stack_message_content(self, content: ContentBlock) ->␣
↪BaseMessageContent:
        if content.type == "text":
            return TextMessageContent(TextArtifact(content.text))
        if content.type == "tool_use":
            name, path = ToolAction.from_native_tool_name(content.name)

            return ActionCallMessageContent(
                artifact=ActionArtifact(
                    value=ToolAction(tag=content.id, name=name, path=path,␣
↪input=content.input),  # pyright: ignore[reportArgumentType]
                ),
            )
```

(continues on next page)

(continued from previous page)

```
        raise ValueError(f"Unsupported message content type: {content.type}")

    def __to_prompt_stack_delta_message_content(
        self,
        event: ContentBlockDeltaEvent | ContentBlockStartEvent,
    ) -> BaseDeltaMessageContent:
        if event.type == "content_block_start":
            content_block = event.content_block

            if content_block.type == "tool_use":
                name, path = ToolAction.from_native_tool_name(content_block.name)

                return ActionCallDeltaMessageContent(index=event.index, tag=content_
↪block.id, name=name, path=path)
            if content_block.type == "text":
                return TextDeltaMessageContent(content_block.text, index=event.index)
            raise ValueError(f"Unsupported content block type: {content_block.type}")
        if event.type == "content_block_delta":
            content_block_delta = event.delta

            if content_block_delta.type == "text_delta":
                return TextDeltaMessageContent(content_block_delta.text, index=event.
↪index)
            if content_block_delta.type == "input_json_delta":
                return ActionCallDeltaMessageContent(index=event.index, partial_
↪input=content_block_delta.partial_json)
            raise ValueError(f"Unsupported message content type: {event}")
        raise ValueError(f"Unsupported message content type: {event}")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `max_tokens = field(default=1000, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='tool', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: AnthropicTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `tool_choice = field(default=Factory(lambda: {'type': 'auto'}), kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `top_k = field(default=250, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `top_p = field(default=0.999, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

- `use_native_tools = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__to_anthropic_content(message)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_anthropic_content(self, message: Message) -> str | list[dict]:
    if message.has_all_content_type(TextMessageContent):
        return message.to_text()
    return [self.__to_anthropic_message_content(content) for content in message.
↪content]
```

__to_anthropic_message_content(content)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_anthropic_message_content(self, content: BaseMessageContent) -> dict:
    if isinstance(content, TextMessageContent):
        return {"type": "text", "text": content.artifact.value}
    if isinstance(content, ImageMessageContent):
        artifact = content.artifact
        if isinstance(artifact, ImageArtifact):
            return {
                "type": "image",
                "source": {"type": "base64", "media_type": artifact.mime_type, "data
↪": artifact.base64},
            }
        if isinstance(artifact, ImageUrlArtifact):
            return {
                "type": "image",
                "source": {"type": "url", "url": artifact.value},
            }
        raise ValueError(f"Unsupported image artifact type: {type(artifact)}")
    if isinstance(content, ActionCallMessageContent):
        action = content.artifact.value

        return {"type": "tool_use", "id": action.tag, "name": action.to_native_tool_
↪name(), "input": action.input}
    if isinstance(content, ActionResultMessageContent):
        artifact = content.artifact

        if isinstance(artifact, ListArtifact):
            message_content = [self.__to_anthropic_tool_result_content(artifact) for␣
↪artifact in artifact.value]
        else:
            message_content = [self.__to_anthropic_tool_result_content(artifact)]

        return {
            "type": "tool_result",
            "tool_use_id": content.action.tag,
            "content": message_content,
            "is_error": isinstance(artifact, ErrorArtifact),
```

(continues on next page)

(continued from previous page)

```
        }
    return content.artifact.value
```

__to_anthropic_messages(messages)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_anthropic_messages(self, messages: list[Message]) -> list[dict]:
    return [
        {"role": self.__to_anthropic_role(message), "content": self.__to_anthropic_
↪content(message)}
        for message in messages
    ]
```

__to_anthropic_role(message)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_anthropic_role(self, message: Message) -> str:
    if message.is_assistant():
        return "assistant"
    return "user"
```

__to_anthropic_tool_result_content(artifact)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_anthropic_tool_result_content(self, artifact: BaseArtifact) -> dict:
    if isinstance(artifact, ImageArtifact):
        return {
            "type": "image",
            "source": {"type": "base64", "media_type": artifact.mime_type, "data":␣
↪artifact.base64},
        }
    if isinstance(artifact, (TextArtifact, ErrorArtifact, InfoArtifact)):
        return {"type": "text", "text": artifact.to_text()}
    raise ValueError(f"Unsupported tool result artifact type: {type(artifact)}")
```

__to_anthropic_tools(tools)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_anthropic_tools(self, tools: list[BaseTool]) -> list[dict]:
    tool_schemas = [
        {
            "name": tool.to_native_tool_name(activity),
            "description": tool.activity_description(activity),
            "input_schema": tool.to_activity_json_schema(activity, "Input Schema"),
        }
        for tool in tools
        for activity in tool.activities()
```

(continues on next page)

(continued from previous page)

```
    ]

    # Anthropic doesnt support $schema and $id
    for tool_schema in tool_schemas:
        del tool_schema["input_schema"]["$schema"]
        del tool_schema["input_schema"]["$id"]

    return tool_schemas
```

__to_prompt_stack_delta_message_content(event)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_prompt_stack_delta_message_content(
    self,
    event: ContentBlockDeltaEvent | ContentBlockStartEvent,
) -> BaseDeltaMessageContent:
    if event.type == "content_block_start":
        content_block = event.content_block

        if content_block.type == "tool_use":
            name, path = ToolAction.from_native_tool_name(content_block.name)

            return ActionCallDeltaMessageContent(index=event.index, tag=content_block.
↪id, name=name, path=path)
        if content_block.type == "text":
            return TextDeltaMessageContent(content_block.text, index=event.index)
        raise ValueError(f"Unsupported content block type: {content_block.type}")
    if event.type == "content_block_delta":
        content_block_delta = event.delta

        if content_block_delta.type == "text_delta":
            return TextDeltaMessageContent(content_block_delta.text, index=event.
↪index)
        if content_block_delta.type == "input_json_delta":
            return ActionCallDeltaMessageContent(index=event.index, partial_
↪input=content_block_delta.partial_json)
        raise ValueError(f"Unsupported message content type: {event}")
    raise ValueError(f"Unsupported message content type: {event}")
```

__to_prompt_stack_message_content(content)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def __to_prompt_stack_message_content(self, content: ContentBlock) ->␣
↪BaseMessageContent:
    if content.type == "text":
        return TextMessageContent(TextArtifact(content.text))
    if content.type == "tool_use":
        name, path = ToolAction.from_native_tool_name(content.name)
```

(continues on next page)

(continued from previous page)

```
        return ActionCallMessageContent(
            artifact=ActionArtifact(
                value=ToolAction(tag=content.id, name=name, path=path, input=content.
↪input),  # pyright: ignore[reportArgumentType]
            ),
        )
    raise ValueError(f"Unsupported message content type: {content.type}")
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    messages = self.__to_anthropic_messages([i for i in prompt_stack.messages if not␣
↪i.is_system()])

    system_messages = prompt_stack.system_messages
    system_message = system_messages[0].to_text() if system_messages else None

    params = {
        "model": self.model,
        "temperature": self.temperature,
        "stop_sequences": self.tokenizer.stop_sequences,
        "top_p": self.top_p,
        "top_k": self.top_k,
        "max_tokens": self.max_tokens,
        "messages": messages,
        **({"system": system_message} if system_message else {}),
        **self.extra_params,
    }

    if prompt_stack.tools and self.use_native_tools:
        params["tool_choice"] = self.tool_choice

        if prompt_stack.output_schema is not None and self.structured_output_strategy␣
↪== "tool":
            params["tool_choice"] = {"type": "any"}

        params["tools"] = self.__to_anthropic_tools(prompt_stack.tools)

    return params
```

client()

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
@lazy_property()
def client(self) -> Client:
    return import_optional_dependency("anthropic").Anthropic(api_key=self.api_key)
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    params = self._base_params(prompt_stack)
    logger.debug(params)
    response = self.client.messages.create(**params)

    logger.debug(response.model_dump())

    return Message(
        content=[self.__to_prompt_stack_message_content(content) for content in␣
↪response.content],
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(input_tokens=response.usage.input_tokens, output_
↪tokens=response.usage.output_tokens),
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    params = {**self._base_params(prompt_stack), "stream": True}
    logger.debug(params)
    events = self.client.messages.create(**params)

    for event in events:
        logger.debug(event)
        if event.type == "content_block_delta" or event.type == "content_block_start":
            yield DeltaMessage(content=self.__to_prompt_stack_delta_message_
↪content(event))
        elif event.type == "message_start":
            yield DeltaMessage(usage=DeltaMessage.Usage(input_tokens=event.message.
↪usage.input_tokens))
        elif event.type == "message_delta":
            yield DeltaMessage(usage=DeltaMessage.Usage(output_tokens=event.usage.
↪output_tokens))
```

validate*structured_output_strategy(*, value)

Source Code in griptape/drivers/prompt/anthropic_prompt_driver.py

```
@structured_output_strategy.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
    if value == "native":
        raise ValueError(f"{__class__.__name__} does not support `{value}` structured␣
↪output strategy.")
```

(continues on next page)

(continued from previous page)

```
    return value
```

azure_openai_chat_prompt_driver

Bases: *AzureOpenAiChatPromptDriver* “)

Attributes

Name	Type	Description
<a href="azure_openai_chat_prompt_driver.griptape.drivers.prompt.azure_openai_chat_prompt_driver.AzureOpenAiChatPromptDriver.azure_deployment" title="azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.azure_openai_chat_prompt_driver.AzureOpenAiChatPromptDriver.azure_deployment)"><code class="language-python">	*str*	An optional Azure OpenAi deployment id. Defaults to the model name.
<a href="azure_openai_chat_prompt_driver.griptape.drivers.prompt.azure_openai_chat_prompt_driver.AzureOpenAiChatPromptDriver.azure_endpoint" title="azure_endpoint = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.azure_openai_chat_prompt_driver.AzureOpenAiChatPromptDriver.azure_endpoint)"><code class="language-python">	*str*	An Azure OpenAi endpoint.

continues on next page

Table 193 – continued from previous page

Name	Type	Description
<a href="azure_openai_chat_prompt_driver.r tape.drivers.prompt.azure nai_chat_prompt_driver.A OpenAiChatPrompt-Driver.azure_ad_token" title="azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.prompt.azure nai_chat_prompt_driver.A OpenAiChatPrompt-Driver.azure_ad_token)"><code class="language-python">	*Optional[str]*	An optional Azure Active Directory token.
<a href="azure_openai_chat_prompt_driver.r tape.drivers.prompt.azure nai_chat_prompt_driver.A OpenAiChatPrompt-Driver.azure_ad_token_provider" title="azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.prompt.azure nai_chat_prompt_driver.A OpenAiChatPrompt-Driver.azure_ad_token_provider)"><code class="language-python">	*Optional[Callable[[], str]]*	An optional Azure Active Directory token provider.

continues on next page

Table 193 – continued from previous page

Name	Type	Description
<a href="azure_openai_chat_prompt_driver.griptape.drivers.prompt.azure_openai_chat_prompt_driver.AzureOpenAiChatPromptDriver.api_version" title="api_version = field(default='2024-10-21', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.azure_openai_chat_prompt_driver.AzureOpenAiChatPromptDriver.api_version)"><code class="language-python">	*str*	An Azure OpenAi API version.
<a href="azure_openai_chat_prompt_driver.griptape.drivers.prompt.azure_openai_chat_prompt_driver.AzureOpenAiChatPromptDriver.client" title="client() (griptape.drivers.prompt.azure_openai_chat_prompt_driver.AzureOpenAiChatPromptDriver.client)"><code class="language-python">	*AzureOpenAI*	An *openai.AzureOpenAI* client.

Source Code in griptape/drivers/prompt/azure_openai_chat_prompt_driver.py

```
@define
class AzureOpenAiChatPromptDriver(OpenAiChatPromptDriver):
    """Azure OpenAi Chat Prompt Driver.

    Attributes:
        azure_deployment: An optional Azure OpenAi deployment id. Defaults to the␣
↪model name.
        azure_endpoint: An Azure OpenAi endpoint.
        azure_ad_token: An optional Azure Active Directory token.
        azure_ad_token_provider: An optional Azure Active Directory token provider.
        api_version: An Azure OpenAi API version.
        client: An `openai.AzureOpenAI` client.
    """

    azure_deployment: str = field(
        kw_only=True,
```

(continues on next page)

(continued from previous page)

```
        default=Factory(lambda self: self.model, takes_self=True),
        metadata={"serializable": True},
    )
    azure_endpoint: str = field(kw_only=True, metadata={"serializable": True})
    azure_ad_token: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": False})
    azure_ad_token_provider: Optional[Callable[[], str]] = field(
        kw_only=True,
        default=None,
        metadata={"serializable": False},
    )
    api_version: str = field(default="2024-10-21", kw_only=True, metadata={
↪"serializable": True})
    _client: Optional[openai.AzureOpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.AzureOpenAI:
        return openai.AzureOpenAI(
            organization=self.organization,
            api_key=self.api_key,
            api_version=self.api_version,
            azure_endpoint=self.azure_endpoint,
            azure_deployment=self.azure_deployment,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        params = super()._base_params(prompt_stack)
        if self.api_version < "2024-02-01" and "seed" in params:
            del params["seed"]
        if self.api_version < "2024-10-21":
            if "stream_options" in params:
                del params["stream_options"]
            if "parallel_tool_calls" in params:
                del params["parallel_tool_calls"]

        # TODO: Add once Azure supports modalities
        if "modalities" in params:
            del params["modalities"]
        return params
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_version = field(default='2024-10-21', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_ad_token_provider = field(kw_only=True, default=None,`

`metadata={'serializable': False})` class-attribute instance-attribute

- `azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True})` class-attribute instance-attribute
- `azure_endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/azure_openai_chat_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    params = super()._base_params(prompt_stack)
    if self.api_version < "2024-02-01" and "seed" in params:
        del params["seed"]
    if self.api_version < "2024-10-21":
        if "stream_options" in params:
            del params["stream_options"]
        if "parallel_tool_calls" in params:
            del params["parallel_tool_calls"]

    # TODO: Add once Azure supports modalities
    if "modalities" in params:
        del params["modalities"]
    return params
```

client()

Source Code in griptape/drivers/prompt/azure_openai_chat_prompt_driver.py

```
@lazy_property()
def client(self) -> openai.AzureOpenAI:
    return openai.AzureOpenAI(
        organization=self.organization,
        api_key=self.api_key,
        api_version=self.api_version,
        azure_endpoint=self.azure_endpoint,
        azure_deployment=self.azure_deployment,
        azure_ad_token=self.azure_ad_token,
        azure_ad_token_provider=self.azure_ad_token_provider,
    )
```

base_prompt_driver

- `StructuredOutputStrategy = Literal['native', 'tool', 'rule']` module-attribute

Bases: *SerializableMixin* “) , *ExponentialBackoffMixin* ”) , ABC

Attributes

Name	Type	Description
<code class="language-python">	*float*	The temperature to use for the completion.
<code class="language-python">	*Optional[int]*	The maximum number of to- kens to generate. If not speci- fied, the value will be automat- ically generated based by the tokenizer.
	str	A function that converts a *PromptStack* to a string.

continues on next page

Table 194 – continued from previous page

Name	Type	Description
<code class="language-python">	*tuple[type[Exception], ...]*	A tuple of exception types to ignore.
<code class="language-python">	*str*	The model name.
<code class="language-python">	<code class="language-python">BaseTokenizer</code>	An instance of *BaseTok- enizer* to when calculating tokens.

continues on next page

Table 194 – continued from previous page

Name	Type	Description
<a href="base_prompt_drive tape.drivers.prompt.base_ PromptDriver.stream" title="stream = field(default=False, kw_only=True, meta- data={'serializable': True}) class-attribute instance-attribute (grip- tape.drivers.prompt.base_ Prompt- Driver.stream)"><code class="language-python">	*bool*	Whether to stream the comple- tion or not. *CompletionChun- kEvent`s will be published to the `Structure* if one is pro- vided.
<a href="base_prompt_drive tape.drivers.prompt.base_ PromptDriver.use_na- tive_tools" ti- tle="use_native_tools = field(default=False, kw_only=True, meta- data={'serializable': True}) class-attribute instance-attribute (grip- tape.drivers.prompt.base_ PromptDriver.use_na- tive_tools)"><code class="language-python">	*bool*	Whether to use LLM's na- tive function calling capabili- ties. Must be supported by the model.
<a href="base_prompt_drive tape.drivers.prompt.base_ PromptDriver.ex- tra_params" ti- tle="extra_params = field(factory=dict, kw_only=True, meta- data={'serializable': True}) class-attribute instance-attribute (grip- tape.drivers.prompt.base_ PromptDriver.ex- tra_params)"><code class="language-python">	*dict*	Extra parameters to pass to the model.

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
@define(kw_only=True)
class BasePromptDriver(SerializableMixin, ExponentialBackoffMixin, ABC):
    """Base class for the Prompt Drivers.
```

(continues on next page)

(continued from previous page)

```
    Attributes:
        temperature: The temperature to use for the completion.
        max_tokens: The maximum number of tokens to generate. If not specified, the␣
↪value will be automatically generated based by the tokenizer.
        prompt_stack_to_string: A function that converts a `PromptStack` to a string.
        ignored_exception_types: A tuple of exception types to ignore.
        model: The model name.
        tokenizer: An instance of `BaseTokenizer` to when calculating tokens.
        stream: Whether to stream the completion or not. `CompletionChunkEvent`s will␣
↪be published to the `Structure` if one is provided.
        use_native_tools: Whether to use LLMs native function calling capabilities.␣
↪Must be supported by the model.
        extra_params: Extra parameters to pass to the model.
    """

    temperature: float = field(default=0.1, metadata={"serializable": True})
    max_tokens: Optional[int] = field(default=None, metadata={"serializable": True})
    ignored_exception_types: tuple[type[Exception], ...] =␣
↪field(default=Factory(lambda: (ImportError, ValueError)))
    model: str = field(metadata={"serializable": True})
    tokenizer: BaseTokenizer
    stream: bool = field(default=False, kw_only=True, metadata={"serializable": True})
    use_native_tools: bool = field(default=False, kw_only=True, metadata={
↪"serializable": True})
    structured_output_strategy: StructuredOutputStrategy = field(
        default="rule", kw_only=True, metadata={"serializable": True}
    )
    extra_params: dict = field(factory=dict, kw_only=True, metadata={"serializable":␣
↪True})

    def before_run(self, prompt_stack: PromptStack) -> None:
        self._init_structured_output(prompt_stack)
        EventBus.publish_event(StartPromptEvent(model=self.model, prompt_stack=prompt_
↪stack))

    def after_run(self, result: Message) -> None:
        EventBus.publish_event(
            FinishPromptEvent(
                model=self.model,
                result=result.value,
                input_token_count=result.usage.input_tokens,
                output_token_count=result.usage.output_tokens,
            ),
        )

    @observable(tags=["PromptDriver.run()"])
    def run(self, prompt_input: PromptStack | BaseArtifact) -> Message:
        if isinstance(prompt_input, BaseArtifact):
            prompt_stack = PromptStack.from_artifact(prompt_input)
        else:
            prompt_stack = prompt_input
```

(continues on next page)

(continued from previous page)

```
        for attempt in self.retrying():
            with attempt:
                self.before_run(prompt_stack)

                result = self.__process_stream(prompt_stack) if self.stream else self.
↪__process_run(prompt_stack)

                self.after_run(result)

                return result
        raise Exception("prompt driver failed after all retry attempts")

    def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
        """Converts a Prompt Stack to a string for token counting or model prompt_
↪input.

        This base implementation is only a rough approximation, and should be␣
↪overridden by subclasses with model-specific tokens.

        Args:
            prompt_stack: The Prompt Stack to convert to a string.

        Returns:
            A single string representation of the Prompt Stack.
        """
        prompt_lines = []

        for i in prompt_stack.messages:
            content = i.to_text()
            if i.is_user():
                prompt_lines.append(f"User: {content}")
            elif i.is_assistant():
                prompt_lines.append(f"Assistant: {content}")
            else:
                prompt_lines.append(content)

        prompt_lines.append("Assistant:")

        return "\n\n".join(prompt_lines)

    @abstractmethod
    def try_run(self, prompt_stack: PromptStack) -> Message: ...

    @abstractmethod
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]: ...

    def _init_structured_output(self, prompt_stack: PromptStack) -> None:
        from griptape.tools import StructuredOutputTool

        if (output_schema := prompt_stack.output_schema) is not None:
            if self.structured_output_strategy == "tool":
```

(continues on next page)

(continued from previous page)

```
                structured_output_tool = StructuredOutputTool(output_schema=output_
↪schema)
                if structured_output_tool not in prompt_stack.tools:
                    prompt_stack.tools.append(structured_output_tool)
            elif self.structured_output_strategy == "rule":
                output_artifact = TextArtifact(JsonSchemaRule(prompt_stack.to_output_
↪json_schema()).to_text())
                system_messages = prompt_stack.system_messages
                if system_messages:
                    last_system_message = prompt_stack.system_messages[-1]
                    last_system_message.content.extend(
                        [
                            TextMessageContent(TextArtifact("\n\n")),
                            TextMessageContent(output_artifact),
                        ]
                    )
                else:
                    prompt_stack.messages.insert(
                        0,
                        Message(
                            content=[TextMessageContent(output_artifact)],
                            role=Message.SYSTEM_ROLE,
                        ),
                    )

    def __process_run(self, prompt_stack: PromptStack) -> Message:
        return self.try_run(prompt_stack)

    def __process_stream(self, prompt_stack: PromptStack) -> Message:
        delta_contents: dict[int, list[BaseDeltaMessageContent]] = {}
        usage = DeltaMessage.Usage()

        # Aggregate all content deltas from the stream
        message_deltas = self.try_stream(prompt_stack)
        for message_delta in message_deltas:
            usage += message_delta.usage
            content = message_delta.content

            if content is not None:
                if content.index in delta_contents:
                    delta_contents[content.index].append(content)
                else:
                    delta_contents[content.index] = [content]
                if isinstance(content, TextDeltaMessageContent):
                    EventBus.publish_event(TextChunkEvent(token=content.text,␣
↪index=content.index))
                elif isinstance(content, AudioDeltaMessageContent) and content.data␣
↪is not None:
                    EventBus.publish_event(AudioChunkEvent(data=content.data))
                elif isinstance(content, ActionCallDeltaMessageContent):
                    EventBus.publish_event(
                        ActionChunkEvent(
```

(continues on next page)

(continued from previous page)

```
                            partial_input=content.partial_input,
                            tag=content.tag,
                            name=content.name,
                            path=content.path,
                            index=content.index,
                        ),
                    )

        # Build a complete content from the content deltas
        return self.__build_message(list(delta_contents.values()), usage)

    def __build_message(
        self, delta_contents: list[list[BaseDeltaMessageContent]], usage:␣
↪DeltaMessage.Usage
    ) -> Message:
        content = []
        for delta_content in delta_contents:
            text_deltas = [delta for delta in delta_content if isinstance(delta,␣
↪TextDeltaMessageContent)]
            audio_deltas = [delta for delta in delta_content if isinstance(delta,␣
↪AudioDeltaMessageContent)]
            action_deltas = [delta for delta in delta_content if isinstance(delta,␣
↪ActionCallDeltaMessageContent)]

            if text_deltas:
                content.append(TextMessageContent.from_deltas(text_deltas))
            if audio_deltas:
                content.append(AudioMessageContent.from_deltas(audio_deltas))
            if action_deltas:
                content.append(ActionCallMessageContent.from_deltas(action_deltas))

        return Message(
            content=content,
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(input_tokens=usage.input_tokens, output_tokens=usage.
↪output_tokens),
        )
```

- `extra_params = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `ignored_exception_types = field(default=Factory(lambda: (ImportError, ValueError)))` class-attribute instance-attribute
- `max_tokens = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(metadata={'serializable': True})` class-attribute instance-attribute
- `stream = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='rule', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `temperature = field(default=0.1, metadata={'serializable': True})` class-attribute

instance-attribute

- `tokenizer` instance-attribute
- `use_native_tools = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__build_message(delta_contents, usage)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def __build_message(
    self, delta_contents: list[list[BaseDeltaMessageContent]], usage: DeltaMessage.
↪Usage
) -> Message:
    content = []
    for delta_content in delta_contents:
        text_deltas = [delta for delta in delta_content if isinstance(delta,␣
↪TextDeltaMessageContent)]
        audio_deltas = [delta for delta in delta_content if isinstance(delta,␣
↪AudioDeltaMessageContent)]
        action_deltas = [delta for delta in delta_content if isinstance(delta,␣
↪ActionCallDeltaMessageContent)]

        if text_deltas:
            content.append(TextMessageContent.from_deltas(text_deltas))
        if audio_deltas:
            content.append(AudioMessageContent.from_deltas(audio_deltas))
        if action_deltas:
            content.append(ActionCallMessageContent.from_deltas(action_deltas))

    return Message(
        content=content,
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(input_tokens=usage.input_tokens, output_tokens=usage.
↪output_tokens),
    )
```

__process_run(prompt_stack)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def __process_run(self, prompt_stack: PromptStack) -> Message:
    return self.try_run(prompt_stack)
```

__process_stream(prompt_stack)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def __process_stream(self, prompt_stack: PromptStack) -> Message:
    delta_contents: dict[int, list[BaseDeltaMessageContent]] = {}
    usage = DeltaMessage.Usage()

    # Aggregate all content deltas from the stream
```

(continues on next page)

(continued from previous page)

```
    message_deltas = self.try_stream(prompt_stack)
    for message_delta in message_deltas:
        usage += message_delta.usage
        content = message_delta.content

        if content is not None:
            if content.index in delta_contents:
                delta_contents[content.index].append(content)
            else:
                delta_contents[content.index] = [content]
            if isinstance(content, TextDeltaMessageContent):
                EventBus.publish_event(TextChunkEvent(token=content.text,␣
↪index=content.index))
            elif isinstance(content, AudioDeltaMessageContent) and content.data is␣
↪not None:
                EventBus.publish_event(AudioChunkEvent(data=content.data))
            elif isinstance(content, ActionCallDeltaMessageContent):
                EventBus.publish_event(
                    ActionChunkEvent(
                        partial_input=content.partial_input,
                        tag=content.tag,
                        name=content.name,
                        path=content.path,
                        index=content.index,
                    ),
                )

    # Build a complete content from the content deltas
    return self.__build_message(list(delta_contents.values()), usage)
```

_init_structured_output(prompt_stack)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def _init_structured_output(self, prompt_stack: PromptStack) -> None:
    from griptape.tools import StructuredOutputTool

    if (output_schema := prompt_stack.output_schema) is not None:
        if self.structured_output_strategy == "tool":
            structured_output_tool = StructuredOutputTool(output_schema=output_schema)
            if structured_output_tool not in prompt_stack.tools:
                prompt_stack.tools.append(structured_output_tool)
        elif self.structured_output_strategy == "rule":
            output_artifact = TextArtifact(JsonSchemaRule(prompt_stack.to_output_json_
↪schema()).to_text())
            system_messages = prompt_stack.system_messages
            if system_messages:
                last_system_message = prompt_stack.system_messages[-1]
                last_system_message.content.extend(
                    [
                        TextMessageContent(TextArtifact("\n\n")),
                        TextMessageContent(output_artifact),
```

(continues on next page)

(continued from previous page)

```
                    ]
                )
            else:
                prompt_stack.messages.insert(
                    0,
                    Message(
                        content=[TextMessageContent(output_artifact)],
                        role=Message.SYSTEM_ROLE,
                    ),
                )
```

after_run(result)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def after_run(self, result: Message) -> None:
    EventBus.publish_event(
        FinishPromptEvent(
            model=self.model,
            result=result.value,
            input_token_count=result.usage.input_tokens,
            output_token_count=result.usage.output_tokens,
        ),
    )
```

before_run(prompt_stack)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def before_run(self, prompt_stack: PromptStack) -> None:
    self._init_structured_output(prompt_stack)
    EventBus.publish_event(StartPromptEvent(model=self.model, prompt_stack=prompt_
↪stack))
```

prompt_stack_to_string(prompt_stack)

Converts a Prompt Stack to a string for token counting or model prompt_input.

This base implementation is only a rough approximation, and should be overridden by subclasses with model-specific tokens.

Parameters

Name	Type	Description	Default
prompt_stack	<a href="../../common#griptape.common.PromptStack" title="PromptStack (griptape.common.PromptStack)"><code class="language-pythc	The Prompt Stack to convert to a string. 	*required*

Returns

Type	Description
str	A single string representation of the Prompt Stack.

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
    """Converts a Prompt Stack to a string for token counting or model prompt_input.

    This base implementation is only a rough approximation, and should be overridden␣
↪by subclasses with model-specific tokens.

    Args:
        prompt_stack: The Prompt Stack to convert to a string.

    Returns:
        A single string representation of the Prompt Stack.
    """
    prompt_lines = []

    for i in prompt_stack.messages:
        content = i.to_text()
        if i.is_user():
            prompt_lines.append(f"User: {content}")
        elif i.is_assistant():
            prompt_lines.append(f"Assistant: {content}")
        else:
            prompt_lines.append(content)

    prompt_lines.append("Assistant:")

    return "\n\n".join(prompt_lines)
```

run(prompt_input)

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
@observable(tags=["PromptDriver.run()"])
def run(self, prompt_input: PromptStack | BaseArtifact) -> Message:
```

(continues on next page)

(continued from previous page)

```
    if isinstance(prompt_input, BaseArtifact):
        prompt_stack = PromptStack.from_artifact(prompt_input)
    else:
        prompt_stack = prompt_input

    for attempt in self.retrying():
        with attempt:
            self.before_run(prompt_stack)

            result = self.__process_stream(prompt_stack) if self.stream else self.__
↪process_run(prompt_stack)

            self.after_run(result)

            return result
    raise Exception("prompt driver failed after all retry attempts")
```

try_run(prompt_stack)abstractmethod

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
@abstractmethod
def try_run(self, prompt_stack: PromptStack) -> Message: ...
```

try_stream(prompt_stack)abstractmethod

Source Code in griptape/drivers/prompt/base_prompt_driver.py

```
@abstractmethod
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]: ...
```

cohere

- `__all__ = ['CoherePromptDriver']` module-attribute

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="cohere.mdx#griptape.drivers.prompt.cohere.CoherePromptDriver.api_key" title="api_key = field(default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.prompt.cohere.CoherePromptDriver.api_key)"><code class="language-python">	*str*	Cohere API key.
<a href="cohere.mdx#griptape.drivers.prompt.cohere.CoherePromptDriver.model" title="model = field(metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.cohere.CoherePromptDriver.model)"><code class="language-python">	*str*	Cohere model name.
<a href="cohere.mdx#griptape.drivers.prompt.cohere.CoherePromptDriver.client" title="client() (griptape.drivers.prompt.cohere.CoherePromptDriver.client)"><code class="language-python">	*ClientV2*	Custom *cohere.Client*.

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
@define(kw_only=True)
class CoherePromptDriver(BasePromptDriver):
    """Cohere Prompt Driver.

    Attributes:
        api_key: Cohere API key.
        model:  Cohere model name.
        client: Custom `cohere.Client`.
    """
```

(continues on next page)

(continued from previous page)

```
    api_key: str = field(default=None, metadata={"serializable": False})
    model: str = field(metadata={"serializable": True})
    force_single_step: bool = field(default=False, kw_only=True, metadata={
↪"serializable": True})
    use_native_tools: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})
    _client: Optional[ClientV2] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: CohereTokenizer(model=self.model, client=self.
↪client), takes_self=True),
    )

    @lazy_property()
    def client(self) -> ClientV2:
        return import_optional_dependency("cohere").ClientV2(self.api_key, log_
↪warning_experimental_features=False)

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        params = self._base_params(prompt_stack)

        logger.debug(params)

        result: ChatResponse = self.client.chat(**params)
        logger.debug(result.model_dump())

        return Message(
            content=self.__to_prompt_stack_message_content(result.message),
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(
                input_tokens=result.usage.tokens.input_tokens if result.usage and␣
↪result.usage.tokens else 0,
                output_tokens=result.usage.tokens.output_tokens if result.usage and␣
↪result.usage.tokens else 0,
            ),
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        params = self._base_params(prompt_stack)
        logger.debug(params)
        result: Iterator[StreamedChatResponseV2] = self.client.chat_stream(**params)

        for event in result:
            if event.type == "stream-end":
                usage = event.response.meta.tokens

                yield DeltaMessage(
                    usage=DeltaMessage.Usage(input_tokens=usage.input_tokens, output_
↪tokens=usage.output_tokens),
                )
```

(continues on next page)

(continued from previous page)

```
            elif event.type in ("tool-plan-delta", "content-delta", "tool-call-start",
↪ "tool-call-delta"):
                yield DeltaMessage(content=self.__to_prompt_stack_delta_message_
↪content(event))

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        tool_results = []

        messages = self.__to_cohere_messages(prompt_stack.messages)

        params = {
            "model": self.model,
            "messages": messages,
            "temperature": self.temperature,
            "stop_sequences": self.tokenizer.stop_sequences,
            "max_tokens": self.max_tokens,
            **({"tool_results": tool_results} if tool_results else {}),
            **self.extra_params,
        }

        if prompt_stack.output_schema is not None and self.structured_output_strategy␣
↪== "native":
            params["response_format"] = {"type": "json_object", "schema": prompt_
↪stack.to_output_json_schema()}

        if prompt_stack.tools and self.use_native_tools:
            params["tools"] = self.__to_cohere_tools(prompt_stack.tools)

        return params

    def __to_cohere_messages(self, messages: list[Message]) -> list[dict]:
        cohere_messages = []

        for message in messages:
            # If the message only contains textual content we can send it as a single␣
↪content.
            if message.is_text():
                cohere_messages.append({"role": self.__to_cohere_role(message),
↪"content": message.to_text()})
            # Action results must be sent as separate messages.
            elif message.has_any_content_type(ActionResultMessageContent):
                cohere_messages.extend(
                    {
                        "role": self.__to_cohere_role(message, action_result),
                        "content": self.__to_cohere_message_content(action_result),
                        "tool_call_id": action_result.action.tag,
                    }
                    for action_result in message.get_content_
↪type(ActionResultMessageContent)
                )

                if message.has_any_content_type(TextMessageContent):
```

(continues on next page)

(continued from previous page)

```
                    cohere_messages.append({"role": self.__to_cohere_role(message),
↪"content": message.to_text()})
            else:
                cohere_message = {
                    "role": self.__to_cohere_role(message),
                    "content": [
                        self.__to_cohere_message_content(content)
                        for content in [
                            content for content in message.content if not␣
↪isinstance(content, ActionCallMessageContent)
                        ]
                    ],
                }

                # Action calls must be attached to the message, not sent as content.
                action_call_content = [
                    content for content in message.content if isinstance(content,␣
↪ActionCallMessageContent)
                ]
                if action_call_content:
                    cohere_message["tool_calls"] = [
                        self.__to_cohere_message_content(action_call) for action_call␣
↪in action_call_content
                    ]

                cohere_messages.append(cohere_message)

        return cohere_messages

    def __to_cohere_message_content(self, content: BaseMessageContent) -> str | dict␣
↪| list[dict]:
        if isinstance(content, ActionCallMessageContent):
            action = content.artifact.value

            return {
                "type": "function",
                "id": action.tag,
                "function": {
                    "name": action.to_native_tool_name(),
                    "arguments": json.dumps(action.input),
                },
            }
        if isinstance(content, ActionResultMessageContent):
            artifact = content.artifact

            if isinstance(artifact, ListArtifact):
                message_content = [{"type": "text", "text": artifact.to_text()} for␣
↪artifact in artifact.value]
            else:
                message_content = {"type": "text", "text": artifact.to_text()}

            return message_content
```

(continues on next page)

(continued from previous page)

```
        return {"type": "text", "text": content.artifact.to_text()}

    def __to_cohere_role(self, message: Message, message_content:␣
↪Optional[BaseMessageContent] = None) -> str:
        if message.is_system():
            return "system"
        if message.is_assistant():
            return "assistant"
        if isinstance(message_content, ActionResultMessageContent):
            return "tool"
        return "user"

    def __to_cohere_tools(self, tools: list[BaseTool]) -> list[dict]:
        return [
            {
                "function": {
                    "name": tool.to_native_tool_name(activity),
                    "description": tool.activity_description(activity),
                    "parameters": tool.to_activity_json_schema(activity, "Parameters␣
↪Schema"),
                },
                "type": "function",
            }
            for tool in tools
            for activity in tool.activities()
        ]

    def __to_prompt_stack_message_content(self, response: AssistantMessageResponse) ->
↪ list[BaseMessageContent]:
        content = []
        if response.content:
            content.extend([TextMessageContent(TextArtifact(content.text)) for␣
↪content in response.content])
        if response.tool_plan:
            content.append(TextMessageContent(TextArtifact(response.tool_plan)))
        if response.tool_calls is not None:
            content.extend(
                [
                    ActionCallMessageContent(
                        ActionArtifact(
                            ToolAction(
                                tag=tool_call.id,
                                name=ToolAction.from_native_tool_name(tool_call.
↪function.name)[0],
                                path=ToolAction.from_native_tool_name(tool_call.
↪function.name)[1],
                                input=json.loads(tool_call.function.arguments),
                            ),
                        ),
                    )
                    for tool_call in response.tool_calls
                    if tool_call.id is not None
```

(continues on next page)

(continued from previous page)

```
                    and tool_call.function is not None
                    and tool_call.function.name is not None
                    and tool_call.function.arguments is not None
                ],
            )

        return content

    def __to_prompt_stack_delta_message_content(self, event: Any) ->␣
↪BaseDeltaMessageContent:
        if event.type == "content-delta":
            return TextDeltaMessageContent(event.delta.message.content.text, index=0)
        if event.type == "tool-plan-delta":
            return TextDeltaMessageContent(event.delta.message["tool_plan"])
        if event.type == "tool-call-start":
            tool_call_delta = event.delta.message["tool_calls"]
            name, path = ToolAction.from_native_tool_name(tool_call_delta["function"][
↪"name"])

            return ActionCallDeltaMessageContent(tag=tool_call_delta["id"], name=name,
↪ path=path)
        if event.type == "tool-call-delta":
            tool_call_delta = event.delta.message["tool_calls"]["function"]

            return ActionCallDeltaMessageContent(partial_input=tool_call_delta[
↪"arguments"])
        raise ValueError(f"Unsupported event type: {event.type}")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `force_single_step = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: CohereTokenizer(model=self.model, client=self.client), takes_self=True))` class-attribute instance-attribute
- `use_native_tools = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__to_cohere_message_content(content)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def __to_cohere_message_content(self, content: BaseMessageContent) -> str | dict |␣
↪list[dict]:
    if isinstance(content, ActionCallMessageContent):
        action = content.artifact.value

        return {
```

(continues on next page)

(continued from previous page)

```
            "type": "function",
            "id": action.tag,
            "function": {
                "name": action.to_native_tool_name(),
                "arguments": json.dumps(action.input),
            },
        }
    if isinstance(content, ActionResultMessageContent):
        artifact = content.artifact

        if isinstance(artifact, ListArtifact):
            message_content = [{"type": "text", "text": artifact.to_text()} for␣
↪artifact in artifact.value]
        else:
            message_content = {"type": "text", "text": artifact.to_text()}

        return message_content
    return {"type": "text", "text": content.artifact.to_text()}
```

__to_cohere_messages(messages)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def __to_cohere_messages(self, messages: list[Message]) -> list[dict]:
    cohere_messages = []

    for message in messages:
        # If the message only contains textual content we can send it as a single␣
↪content.
        if message.is_text():
            cohere_messages.append({"role": self.__to_cohere_role(message), "content
↪": message.to_text()})
        # Action results must be sent as separate messages.
        elif message.has_any_content_type(ActionResultMessageContent):
            cohere_messages.extend(
                {
                    "role": self.__to_cohere_role(message, action_result),
                    "content": self.__to_cohere_message_content(action_result),
                    "tool_call_id": action_result.action.tag,
                }
                for action_result in message.get_content_
↪type(ActionResultMessageContent)
            )

            if message.has_any_content_type(TextMessageContent):
                cohere_messages.append({"role": self.__to_cohere_role(message),
↪"content": message.to_text()})
        else:
            cohere_message = {
                "role": self.__to_cohere_role(message),
                "content": [
                    self.__to_cohere_message_content(content)
```

(continues on next page)

(continued from previous page)

```
                    for content in [
                        content for content in message.content if not␣
↪isinstance(content, ActionCallMessageContent)
                    ]
                ],
            }

            # Action calls must be attached to the message, not sent as content.
            action_call_content = [
                content for content in message.content if isinstance(content,␣
↪ActionCallMessageContent)
            ]
            if action_call_content:
                cohere_message["tool_calls"] = [
                    self.__to_cohere_message_content(action_call) for action_call in␣
↪action_call_content
                ]

            cohere_messages.append(cohere_message)

    return cohere_messages
```

__to_cohere_role(message, message_content=None)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def __to_cohere_role(self, message: Message, message_content:␣
↪Optional[BaseMessageContent] = None) -> str:
    if message.is_system():
        return "system"
    if message.is_assistant():
        return "assistant"
    if isinstance(message_content, ActionResultMessageContent):
        return "tool"
    return "user"
```

__to_cohere_tools(tools)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def __to_cohere_tools(self, tools: list[BaseTool]) -> list[dict]:
    return [
        {
            "function": {
                "name": tool.to_native_tool_name(activity),
                "description": tool.activity_description(activity),
                "parameters": tool.to_activity_json_schema(activity, "Parameters␣
↪Schema"),
            },
            "type": "function",
        }
```

(continues on next page)

(continued from previous page)

```
        for tool in tools
        for activity in tool.activities()
    ]
```

__to_prompt_stack_delta_message_content(event)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def __to_prompt_stack_delta_message_content(self, event: Any) ->␣
↪BaseDeltaMessageContent:
    if event.type == "content-delta":
        return TextDeltaMessageContent(event.delta.message.content.text, index=0)
    if event.type == "tool-plan-delta":
        return TextDeltaMessageContent(event.delta.message["tool_plan"])
    if event.type == "tool-call-start":
        tool_call_delta = event.delta.message["tool_calls"]
        name, path = ToolAction.from_native_tool_name(tool_call_delta["function"][
↪"name"])

        return ActionCallDeltaMessageContent(tag=tool_call_delta["id"], name=name,␣
↪path=path)
    if event.type == "tool-call-delta":
        tool_call_delta = event.delta.message["tool_calls"]["function"]

        return ActionCallDeltaMessageContent(partial_input=tool_call_delta["arguments
↪"])
    raise ValueError(f"Unsupported event type: {event.type}")
```

__to_prompt_stack_message_content(response)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def __to_prompt_stack_message_content(self, response: AssistantMessageResponse) ->␣
↪list[BaseMessageContent]:
    content = []
    if response.content:
        content.extend([TextMessageContent(TextArtifact(content.text)) for content in␣
↪response.content])
    if response.tool_plan:
        content.append(TextMessageContent(TextArtifact(response.tool_plan)))
    if response.tool_calls is not None:
        content.extend(
            [
                ActionCallMessageContent(
                    ActionArtifact(
                        ToolAction(
                            tag=tool_call.id,
                            name=ToolAction.from_native_tool_name(tool_call.function.
↪name)[0],
                            path=ToolAction.from_native_tool_name(tool_call.function.
↪name)[1],
```

(continues on next page)

(continued from previous page)

```
                            input=json.loads(tool_call.function.arguments),
                        ),
                    ),
                )
                for tool_call in response.tool_calls
                if tool_call.id is not None
                and tool_call.function is not None
                and tool_call.function.name is not None
                and tool_call.function.arguments is not None
            ],
        )

    return content
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    tool_results = []

    messages = self.__to_cohere_messages(prompt_stack.messages)

    params = {
        "model": self.model,
        "messages": messages,
        "temperature": self.temperature,
        "stop_sequences": self.tokenizer.stop_sequences,
        "max_tokens": self.max_tokens,
        **({"tool_results": tool_results} if tool_results else {}),
        **self.extra_params,
    }

    if prompt_stack.output_schema is not None and self.structured_output_strategy ==
↪"native":
        params["response_format"] = {"type": "json_object", "schema": prompt_stack.to_
↪output_json_schema()}

    if prompt_stack.tools and self.use_native_tools:
        params["tools"] = self.__to_cohere_tools(prompt_stack.tools)

    return params
```

client()

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
@lazy_property()
def client(self) -> ClientV2:
    return import_optional_dependency("cohere").ClientV2(self.api_key, log_warning_
↪experimental_features=False)
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    params = self._base_params(prompt_stack)

    logger.debug(params)

    result: ChatResponse = self.client.chat(**params)
    logger.debug(result.model_dump())

    return Message(
        content=self.__to_prompt_stack_message_content(result.message),
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(
            input_tokens=result.usage.tokens.input_tokens if result.usage and result.
↪usage.tokens else 0,
            output_tokens=result.usage.tokens.output_tokens if result.usage and␣
↪result.usage.tokens else 0,
        ),
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    params = self._base_params(prompt_stack)
    logger.debug(params)
    result: Iterator[StreamedChatResponseV2] = self.client.chat_stream(**params)

    for event in result:
        if event.type == "stream-end":
            usage = event.response.meta.tokens

            yield DeltaMessage(
                usage=DeltaMessage.Usage(input_tokens=usage.input_tokens, output_
↪tokens=usage.output_tokens),
            )
        elif event.type in ("tool-plan-delta", "content-delta", "tool-call-start",
↪"tool-call-delta"):
            yield DeltaMessage(content=self.__to_prompt_stack_delta_message_
↪content(event))
```

cohere_prompt_driver

- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="cohere_prompt_driver.mdx#griptape.drivers.prompt.cohere_prompt_driver.CoherePromptDriver.api_key" title="api_key = field(default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.prompt.cohere_prompt_driver.CoherePromptDriver.api_key)"><code class="language-python">	*str*	Cohere API key.
<a href="cohere_prompt_driver.mdx#griptape.drivers.prompt.cohere_prompt_driver.CoherePromptDriver.model" title="model = field(metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.cohere_prompt_driver.CoherePromptDriver.model)"><code class="language-python">	*str*	Cohere model name.
<a href="cohere_prompt_driver.mdx#griptape.drivers.prompt.cohere_prompt_driver.CoherePromptDriver.client" title="client() (griptape.drivers.prompt.cohere_prompt_driver.CoherePromptDriver.client)"><code class="language-python">	*ClientV2*	Custom *cohere.Client*.

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
@define(kw_only=True)
class CoherePromptDriver(BasePromptDriver):
```

(continues on next page)

(continued from previous page)

```
    """Cohere Prompt Driver.

    Attributes:
        api_key: Cohere API key.
        model:  Cohere model name.
        client: Custom `cohere.Client`.
    """

    api_key: str = field(default=None, metadata={"serializable": False})
    model: str = field(metadata={"serializable": True})
    force_single_step: bool = field(default=False, kw_only=True, metadata={
↪"serializable": True})
    use_native_tools: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})
    _client: Optional[ClientV2] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: CohereTokenizer(model=self.model, client=self.
↪client), takes_self=True),
    )

    @lazy_property()
    def client(self) -> ClientV2:
        return import_optional_dependency("cohere").ClientV2(self.api_key, log_
↪warning_experimental_features=False)

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        params = self._base_params(prompt_stack)

        logger.debug(params)

        result: ChatResponse = self.client.chat(**params)
        logger.debug(result.model_dump())

        return Message(
            content=self.__to_prompt_stack_message_content(result.message),
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(
                input_tokens=result.usage.tokens.input_tokens if result.usage and␣
↪result.usage.tokens else 0,
                output_tokens=result.usage.tokens.output_tokens if result.usage and␣
↪result.usage.tokens else 0,
            ),
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        params = self._base_params(prompt_stack)
        logger.debug(params)
        result: Iterator[StreamedChatResponseV2] = self.client.chat_stream(**params)
```

(continues on next page)

(continued from previous page)

```
        for event in result:
            if event.type == "stream-end":
                usage = event.response.meta.tokens

                yield DeltaMessage(
                    usage=DeltaMessage.Usage(input_tokens=usage.input_tokens, output_
↪tokens=usage.output_tokens),
                )
            elif event.type in ("tool-plan-delta", "content-delta", "tool-call-start",
↪ "tool-call-delta"):
                yield DeltaMessage(content=self.__to_prompt_stack_delta_message_
↪content(event))

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        tool_results = []

        messages = self.__to_cohere_messages(prompt_stack.messages)

        params = {
            "model": self.model,
            "messages": messages,
            "temperature": self.temperature,
            "stop_sequences": self.tokenizer.stop_sequences,
            "max_tokens": self.max_tokens,
            **({"tool_results": tool_results} if tool_results else {}),
            **self.extra_params,
        }

        if prompt_stack.output_schema is not None and self.structured_output_strategy␣
↪== "native":
            params["response_format"] = {"type": "json_object", "schema": prompt_
↪stack.to_output_json_schema()}

        if prompt_stack.tools and self.use_native_tools:
            params["tools"] = self.__to_cohere_tools(prompt_stack.tools)

        return params

    def __to_cohere_messages(self, messages: list[Message]) -> list[dict]:
        cohere_messages = []

        for message in messages:
            # If the message only contains textual content we can send it as a single␣
↪content.
            if message.is_text():
                cohere_messages.append({"role": self.__to_cohere_role(message),
↪"content": message.to_text()})
            # Action results must be sent as separate messages.
            elif message.has_any_content_type(ActionResultMessageContent):
                cohere_messages.extend(
                    {
                        "role": self.__to_cohere_role(message, action_result),
```

(continues on next page)

(continued from previous page)

```
                        "content": self.__to_cohere_message_content(action_result),
                        "tool_call_id": action_result.action.tag,
                    }
                    for action_result in message.get_content_
↪type(ActionResultMessageContent)
                )

                if message.has_any_content_type(TextMessageContent):
                    cohere_messages.append({"role": self.__to_cohere_role(message),
↪"content": message.to_text()})
            else:
                cohere_message = {
                    "role": self.__to_cohere_role(message),
                    "content": [
                        self.__to_cohere_message_content(content)
                        for content in [
                            content for content in message.content if not␣
↪isinstance(content, ActionCallMessageContent)
                        ]
                    ],
                }

                # Action calls must be attached to the message, not sent as content.
                action_call_content = [
                    content for content in message.content if isinstance(content,␣
↪ActionCallMessageContent)
                ]
                if action_call_content:
                    cohere_message["tool_calls"] = [
                        self.__to_cohere_message_content(action_call) for action_call␣
↪in action_call_content
                    ]

                cohere_messages.append(cohere_message)

        return cohere_messages

    def __to_cohere_message_content(self, content: BaseMessageContent) -> str | dict␣
↪| list[dict]:
        if isinstance(content, ActionCallMessageContent):
            action = content.artifact.value

            return {
                "type": "function",
                "id": action.tag,
                "function": {
                    "name": action.to_native_tool_name(),
                    "arguments": json.dumps(action.input),
                },
            }
        if isinstance(content, ActionResultMessageContent):
            artifact = content.artifact
```

(continues on next page)

(continued from previous page)

```
            if isinstance(artifact, ListArtifact):
                message_content = [{"type": "text", "text": artifact.to_text()} for
↪artifact in artifact.value]
            else:
                message_content = {"type": "text", "text": artifact.to_text()}

            return message_content
        return {"type": "text", "text": content.artifact.to_text()}

    def __to_cohere_role(self, message: Message, message_content:
↪Optional[BaseMessageContent] = None) -> str:
        if message.is_system():
            return "system"
        if message.is_assistant():
            return "assistant"
        if isinstance(message_content, ActionResultMessageContent):
            return "tool"
        return "user"

    def __to_cohere_tools(self, tools: list[BaseTool]) -> list[dict]:
        return [
            {
                "function": {
                    "name": tool.to_native_tool_name(activity),
                    "description": tool.activity_description(activity),
                    "parameters": tool.to_activity_json_schema(activity, "Parameters
↪Schema"),
                },
                "type": "function",
            }
            for tool in tools
            for activity in tool.activities()
        ]

    def __to_prompt_stack_message_content(self, response: AssistantMessageResponse) ->
↪ list[BaseMessageContent]:
        content = []
        if response.content:
            content.extend([TextMessageContent(TextArtifact(content.text)) for
↪content in response.content])
        if response.tool_plan:
            content.append(TextMessageContent(TextArtifact(response.tool_plan)))
        if response.tool_calls is not None:
            content.extend(
                [
                    ActionCallMessageContent(
                        ActionArtifact(
                            ToolAction(
                                tag=tool_call.id,
                                name=ToolAction.from_native_tool_name(tool_call.
↪function.name)[0],
```

(continues on next page)

(continued from previous page)

```
                                path=ToolAction.from_native_tool_name(tool_call.
↪function.name)[1],
                                input=json.loads(tool_call.function.arguments),
                            ),
                        ),
                    )
                    for tool_call in response.tool_calls
                    if tool_call.id is not None
                    and tool_call.function is not None
                    and tool_call.function.name is not None
                    and tool_call.function.arguments is not None
                ],
            )

        return content

    def __to_prompt_stack_delta_message_content(self, event: Any) ->␣
↪BaseDeltaMessageContent:
        if event.type == "content-delta":
            return TextDeltaMessageContent(event.delta.message.content.text, index=0)
        if event.type == "tool-plan-delta":
            return TextDeltaMessageContent(event.delta.message["tool_plan"])
        if event.type == "tool-call-start":
            tool_call_delta = event.delta.message["tool_calls"]
            name, path = ToolAction.from_native_tool_name(tool_call_delta["function"][
↪"name"])

            return ActionCallDeltaMessageContent(tag=tool_call_delta["id"], name=name,
↪ path=path)
        if event.type == "tool-call-delta":
            tool_call_delta = event.delta.message["tool_calls"]["function"]

            return ActionCallDeltaMessageContent(partial_input=tool_call_delta[
↪"arguments"])
        raise ValueError(f"Unsupported event type: {event.type}")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `force_single_step = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: CohereTokenizer(model=self.model, client=self.client), takes_self=True))` class-attribute instance-attribute
- `use_native_tools = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__to_cohere_message_content(content)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def __to_cohere_message_content(self, content: BaseMessageContent) -> str | dict |␣
↪list[dict]:
    if isinstance(content, ActionCallMessageContent):
        action = content.artifact.value

        return {
            "type": "function",
            "id": action.tag,
            "function": {
                "name": action.to_native_tool_name(),
                "arguments": json.dumps(action.input),
            },
        }
    if isinstance(content, ActionResultMessageContent):
        artifact = content.artifact

        if isinstance(artifact, ListArtifact):
            message_content = [{"type": "text", "text": artifact.to_text()} for␣
↪artifact in artifact.value]
        else:
            message_content = {"type": "text", "text": artifact.to_text()}

        return message_content
    return {"type": "text", "text": content.artifact.to_text()}
```

__to_cohere_messages(messages)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def __to_cohere_messages(self, messages: list[Message]) -> list[dict]:
    cohere_messages = []

    for message in messages:
        # If the message only contains textual content we can send it as a single␣
↪content.
        if message.is_text():
            cohere_messages.append({"role": self.__to_cohere_role(message), "content
↪": message.to_text()})
        # Action results must be sent as separate messages.
        elif message.has_any_content_type(ActionResultMessageContent):
            cohere_messages.extend(
                {
                    "role": self.__to_cohere_role(message, action_result),
                    "content": self.__to_cohere_message_content(action_result),
                    "tool_call_id": action_result.action.tag,
                }
                for action_result in message.get_content_
↪type(ActionResultMessageContent)
            )
```

(continues on next page)

(continued from previous page)

```
            if message.has_any_content_type(TextMessageContent):
                cohere_messages.append({"role": self.__to_cohere_role(message),
↪"content": message.to_text()})
        else:
            cohere_message = {
                "role": self.__to_cohere_role(message),
                "content": [
                    self.__to_cohere_message_content(content)
                    for content in [
                        content for content in message.content if not␣
↪isinstance(content, ActionCallMessageContent)
                    ]
                ],
            }

            # Action calls must be attached to the message, not sent as content.
            action_call_content = [
                content for content in message.content if isinstance(content,␣
↪ActionCallMessageContent)
            ]
            if action_call_content:
                cohere_message["tool_calls"] = [
                    self.__to_cohere_message_content(action_call) for action_call in␣
↪action_call_content
                ]

            cohere_messages.append(cohere_message)

    return cohere_messages
```

__to_cohere_role(message, message_content=None)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def __to_cohere_role(self, message: Message, message_content:␣
↪Optional[BaseMessageContent] = None) -> str:
    if message.is_system():
        return "system"
    if message.is_assistant():
        return "assistant"
    if isinstance(message_content, ActionResultMessageContent):
        return "tool"
    return "user"
```

__to_cohere_tools(tools)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def __to_cohere_tools(self, tools: list[BaseTool]) -> list[dict]:
    return [
```

(continues on next page)

(continued from previous page)

```
        {
            "function": {
                "name": tool.to_native_tool_name(activity),
                "description": tool.activity_description(activity),
                "parameters": tool.to_activity_json_schema(activity, "Parameters␣
↪Schema"),
            },
            "type": "function",
        }
        for tool in tools
        for activity in tool.activities()
    ]
```

__to_prompt_stack_delta_message_content(event)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def __to_prompt_stack_delta_message_content(self, event: Any) ->␣
↪BaseDeltaMessageContent:
    if event.type == "content-delta":
        return TextDeltaMessageContent(event.delta.message.content.text, index=0)
    if event.type == "tool-plan-delta":
        return TextDeltaMessageContent(event.delta.message["tool_plan"])
    if event.type == "tool-call-start":
        tool_call_delta = event.delta.message["tool_calls"]
        name, path = ToolAction.from_native_tool_name(tool_call_delta["function"][
↪"name"])

        return ActionCallDeltaMessageContent(tag=tool_call_delta["id"], name=name,␣
↪path=path)
    if event.type == "tool-call-delta":
        tool_call_delta = event.delta.message["tool_calls"]["function"]

        return ActionCallDeltaMessageContent(partial_input=tool_call_delta["arguments
↪"])
    raise ValueError(f"Unsupported event type: {event.type}")
```

__to_prompt_stack_message_content(response)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def __to_prompt_stack_message_content(self, response: AssistantMessageResponse) ->␣
↪list[BaseMessageContent]:
    content = []
    if response.content:
        content.extend([TextMessageContent(TextArtifact(content.text)) for content in␣
↪response.content])
    if response.tool_plan:
        content.append(TextMessageContent(TextArtifact(response.tool_plan)))
    if response.tool_calls is not None:
        content.extend(
```

(continues on next page)

(continued from previous page)

```
            [
                ActionCallMessageContent(
                    ActionArtifact(
                        ToolAction(
                            tag=tool_call.id,
                            name=ToolAction.from_native_tool_name(tool_call.function.
↪name)[0],
                            path=ToolAction.from_native_tool_name(tool_call.function.
↪name)[1],
                            input=json.loads(tool_call.function.arguments),
                        ),
                    ),
                )
                for tool_call in response.tool_calls
                if tool_call.id is not None
                and tool_call.function is not None
                and tool_call.function.name is not None
                and tool_call.function.arguments is not None
            ],
        )

    return content
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    tool_results = []

    messages = self.__to_cohere_messages(prompt_stack.messages)

    params = {
        "model": self.model,
        "messages": messages,
        "temperature": self.temperature,
        "stop_sequences": self.tokenizer.stop_sequences,
        "max_tokens": self.max_tokens,
        **({"tool_results": tool_results} if tool_results else {}),
        **self.extra_params,
    }

    if prompt_stack.output_schema is not None and self.structured_output_strategy ==
↪"native":
        params["response_format"] = {"type": "json_object", "schema": prompt_stack.to_
↪output_json_schema()}

    if prompt_stack.tools and self.use_native_tools:
        params["tools"] = self.__to_cohere_tools(prompt_stack.tools)

    return params
```

client()

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
@lazy_property()
def client(self) -> ClientV2:
    return import_optional_dependency("cohere").ClientV2(self.api_key, log_warning_
↪experimental_features=False)
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    params = self._base_params(prompt_stack)

    logger.debug(params)

    result: ChatResponse = self.client.chat(**params)
    logger.debug(result.model_dump())

    return Message(
        content=self.__to_prompt_stack_message_content(result.message),
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(
            input_tokens=result.usage.tokens.input_tokens if result.usage and result.
↪usage.tokens else 0,
            output_tokens=result.usage.tokens.output_tokens if result.usage and␣
↪result.usage.tokens else 0,
        ),
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/cohere_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    params = self._base_params(prompt_stack)
    logger.debug(params)
    result: Iterator[StreamedChatResponseV2] = self.client.chat_stream(**params)

    for event in result:
        if event.type == "stream-end":
            usage = event.response.meta.tokens

            yield DeltaMessage(
                usage=DeltaMessage.Usage(input_tokens=usage.input_tokens, output_
↪tokens=usage.output_tokens),
            )
        elif event.type in ("tool-plan-delta", "content-delta", "tool-call-start",
↪"tool-call-delta"):
```

(continues on next page)

(continued from previous page)

```
            yield DeltaMessage(content=self.__to_prompt_stack_delta_message_
↪content(event))
```

dummy

- `__all__ = ['DummyPromptDriver']` module-attribute

Bases: *BasePromptDriver* “)

Source Code in griptape/drivers/prompt/dummy_prompt_driver.py

```
@define
class DummyPromptDriver(BasePromptDriver):
    model: None = field(init=False, default=None, kw_only=True)
    tokenizer: DummyTokenizer = field(default=Factory(lambda: DummyTokenizer()), kw_
↪only=True)

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        raise DummyError(__class__.__name__, "try_run")

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        raise DummyError(__class__.__name__, "try_stream")
```

- `model = field(init=False, default=None, kw_only=True)` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda: DummyTokenizer()), kw_only=True)` class-attribute instance-attribute

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/dummy_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    raise DummyError(__class__.__name__, "try_run")
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/dummy_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    raise DummyError(__class__.__name__, "try_stream")
```

dummy_prompt_driver

Bases: *BasePromptDriver* “)

Source Code in griptape/drivers/prompt/dummy_prompt_driver.py

```
@define
class DummyPromptDriver(BasePromptDriver):
    model: None = field(init=False, default=None, kw_only=True)
    tokenizer: DummyTokenizer = field(default=Factory(lambda: DummyTokenizer()), kw_
↪only=True)

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        raise DummyError(__class__.__name__, "try_run")

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        raise DummyError(__class__.__name__, "try_stream")
```

- `model = field(init=False, default=None, kw_only=True)` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda: DummyTokenizer()), kw_only=True)` class-attribute instance-attribute

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/dummy_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    raise DummyError(__class__.__name__, "try_run")
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/dummy_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    raise DummyError(__class__.__name__, "try_stream")
```

google

- `__all__ = ['GooglePromptDriver']` module-attribute

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="google.mdx#grip-tape.drivers.prompt.googl Driver.api_key" title="api_key = field(default=None, kw_only=True, meta-data={'serializable': False}) class-attribute instance-attribute (grip-tape.drivers.prompt.googl Driver.api_key)"><code class="language-python">	*Optional[str]*	Google API key.
<a href="google.mdx#grip-tape.drivers.prompt.googl Driver.model" title="model = field(kw_only=True, metadata={'seri-alizable': True}) class-attribute instance-attribute (grip-tape.drivers.prompt.googl Driver.model)"><code class="language-python">	*str*	Google model name.
<a href="google.mdx#grip-tape.drivers.prompt.googl Driver.client" ti-tle="client() (grip-tape.drivers.prompt.googl Driver.client)"><code class="language-python">	*GenerativeModel*	Custom *GenerativeModel* client.
<a href="google.mdx#grip-tape.drivers.prompt.googl Driver.top_p" title="top_p = field(default=None, kw_only=True, meta-data={'serializable': True}) class-attribute instance-attribute (grip-tape.drivers.prompt.googl Driver.top_p)"><code class="language-python">	*Optional[float]*	Optional value for top_p.

continues on next page

Table 199 – continued from previous page

Name	Type	Description
<a href="google.mdx#griptape.drivers.prompt.google_prompt_driver.GooglePromptDriver.top_k" title="top_k = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.google_prompt_driver.GooglePromptDriver.top_k)"><code class="language-python">	*Optional[int]*	Optional value for top_k.

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
@define
class GooglePromptDriver(BasePromptDriver):
    """Google Prompt Driver.

    Attributes:
        api_key: Google API key.
        model: Google model name.
        client: Custom `GenerativeModel` client.
        top_p: Optional value for top_p.
        top_k: Optional value for top_k.
    """

    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    model: str = field(kw_only=True, metadata={"serializable": True})
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: GoogleTokenizer(api_key=self.api_key, model=self.
↪model), takes_self=True),
        kw_only=True,
    )
    top_p: Optional[float] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    top_k: Optional[int] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    use_native_tools: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})
    structured_output_strategy: StructuredOutputStrategy = field(
        default="tool", kw_only=True, metadata={"serializable": True}
    )
    tool_choice: str = field(default="auto", kw_only=True, metadata={"serializable":␣
↪True})
    _client: Optional[GenerativeModel] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )
```

(continues on next page)

(continued from previous page)

```
    @structured_output_strategy.validator  # pyright:␣
↪ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
        if value == "native":
            raise ValueError(f"{__class__.__name__} does not support `{value}`␣
↪structured output strategy.")

        return value

    @lazy_property()
    def client(self) -> GenerativeModel:
        genai = import_optional_dependency("google.generativeai")
        genai.configure(api_key=self.api_key)

        return genai.GenerativeModel(self.model)

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        messages = self.__to_google_messages(prompt_stack)
        params = self._base_params(prompt_stack)
        logger.debug((messages, params["generation_config"].__dict__))
        response: GenerateContentResponse = self.client.generate_content(messages,␣
↪**params)
        logger.debug(response.to_dict())

        usage_metadata = response.usage_metadata

        return Message(
            content=[self.__to_prompt_stack_message_content(part) for part in␣
↪response.parts],
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(
                input_tokens=usage_metadata.prompt_token_count,
                output_tokens=usage_metadata.candidates_token_count,
            ),
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        messages = self.__to_google_messages(prompt_stack)
        params = {**self._base_params(prompt_stack), "stream": True}
        logger.debug((messages, params))
        response: GenerateContentResponse = self.client.generate_content(
            messages,
            **params,
        )

        prompt_token_count = None
        for chunk in response:
            logger.debug(chunk.to_dict())
            usage_metadata = chunk.usage_metadata
```

(continues on next page)

(continued from previous page)

```
            content = self.__to_prompt_stack_delta_message_content(chunk.parts[0]) if␣
↪chunk.parts else None
            # Only want to output the prompt token count once since it is static each␣
↪chunk
            if prompt_token_count is None:
                prompt_token_count = usage_metadata.prompt_token_count
                yield DeltaMessage(
                    content=content,
                    usage=DeltaMessage.Usage(
                        input_tokens=usage_metadata.prompt_token_count,
                        output_tokens=usage_metadata.candidates_token_count,
                    ),
                )
            else:
                yield DeltaMessage(
                    content=content,
                    usage=DeltaMessage.Usage(output_tokens=usage_metadata.candidates_
↪token_count),
                )

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        types = import_optional_dependency("google.generativeai.types")
        protos = import_optional_dependency("google.generativeai.protos")

        system_messages = prompt_stack.system_messages
        if system_messages:
            self.client._system_instruction = types.ContentDict(
                role="system",
                parts=[protos.Part(text=system_message.to_text()) for system_message␣
↪in system_messages],
            )

        params = {
            "generation_config": types.GenerationConfig(
                **{
                    # For some reason, providing stop sequences when streaming breaks␣
↪native functions
                    # https://github.com/google-gemini/generative-ai-python/issues/446
                    "stop_sequences": [] if self.stream and self.use_native_tools␣
↪else self.tokenizer.stop_sequences,
                    "max_output_tokens": self.max_tokens,
                    "temperature": self.temperature,
                    "top_p": self.top_p,
                    "top_k": self.top_k,
                    **self.extra_params,
                },
            ),
        }

        if prompt_stack.tools and self.use_native_tools:
            params["tool_config"] = {"function_calling_config": {"mode": self.tool_
```

(continues on next page)

(continued from previous page)

```
↪choice}}

            if prompt_stack.output_schema is not None and self.structured_output_
↪strategy == "tool":
                params["tool_config"]["function_calling_config"]["mode"] = "auto"

            params["tools"] = self.__to_google_tools(prompt_stack.tools)

        return params

    def __to_google_messages(self, prompt_stack: PromptStack) -> ContentsType:
        types = import_optional_dependency("google.generativeai.types")

        return [
            types.ContentDict(
                {
                    "role": self.__to_google_role(message),
                    "parts": [self.__to_google_message_content(content) for content␣
↪in message.content],
                },
            )
            for message in prompt_stack.messages
            if not message.is_system()
        ]

    def __to_google_role(self, message: Message) -> str:
        if message.is_assistant():
            return "model"
        return "user"

    def __to_google_tools(self, tools: list[BaseTool]) -> list[dict]:
        types = import_optional_dependency("google.generativeai.types")

        tool_declarations = []
        for tool in tools:
            for activity in tool.activities():
                schema = tool.to_activity_json_schema(activity, "Parameters Schema")

                if "values" in schema["properties"]:
                    schema = schema["properties"]["values"]

                schema = remove_key_in_dict_recursively(schema, "additionalProperties
↪")
                tool_declaration = types.FunctionDeclaration(
                    name=tool.to_native_tool_name(activity),
                    description=tool.activity_description(activity),
                    **(
                        {
                            "parameters": {
                                "type": schema["type"],
                                "properties": schema["properties"],
                                "required": schema.get("required", []),
```

(continues on next page)

(continued from previous page)

```
                            }
                        }
                        if schema.get("properties")
                        else {}
                    ),
                )

                tool_declarations.append(tool_declaration)

        return tool_declarations

    def __to_google_message_content(self, content: BaseMessageContent) -> ContentDict
↪| Part | str:
        types = import_optional_dependency("google.generativeai.types")
        protos = import_optional_dependency("google.generativeai.protos")

        if isinstance(content, TextMessageContent):
            return content.artifact.to_text()
        if isinstance(content, ImageMessageContent):
            if isinstance(content.artifact, ImageArtifact):
                return types.ContentDict(mime_type=content.artifact.mime_type,
↪data=content.artifact.value)
            # TODO: Google requires uploading to the files endpoint: https://ai.
↪google.dev/gemini-api/docs/image-understanding#upload-image
            # Can be worked around by using GenericMessageContent, similar to videos.
            raise ValueError(f"Unsupported image artifact type: {type(content.
↪artifact)}")
        if isinstance(content, ActionCallMessageContent):
            action = content.artifact.value

            return protos.Part(function_call=protos.FunctionCall(name=action.tag,
↪args=action.input))
        if isinstance(content, ActionResultMessageContent):
            artifact = content.artifact

            return protos.Part(
                function_response=protos.FunctionResponse(
                    name=content.action.to_native_tool_name(),
                    response=artifact.to_dict(),
                ),
            )
        if isinstance(content, GenericMessageContent):
            return content.artifact.value
        raise ValueError(f"Unsupported prompt stack content type: {type(content)}")

    def __to_prompt_stack_message_content(self, content: Part) -> BaseMessageContent:
        json_format = import_optional_dependency("google.protobuf.json_format")

        if content.text:
            return TextMessageContent(TextArtifact(content.text))
        if content.function_call:
            function_call = content.function_call
```

(continues on next page)

(continued from previous page)

```
            name, path = ToolAction.from_native_tool_name(function_call.name)

            args = json_format.MessageToDict(function_call._pb).get("args", {})
            return ActionCallMessageContent(
                artifact=ActionArtifact(value=ToolAction(tag=function_call.name,␣
↪name=name, path=path, input=args)),
            )
        raise ValueError(f"Unsupported message content type {content}")

    def __to_prompt_stack_delta_message_content(self, content: Part) ->␣
↪BaseDeltaMessageContent:
        json_format = import_optional_dependency("google.protobuf.json_format")

        if content.text:
            return TextDeltaMessageContent(content.text)
        if content.function_call:
            function_call = content.function_call

            name, path = ToolAction.from_native_tool_name(function_call.name)

            args = json_format.MessageToDict(function_call._pb).get("args", {})
            return ActionCallDeltaMessageContent(
                tag=function_call.name,
                name=name,
                path=path,
                partial_input=json.dumps(args),
            )
        raise ValueError(f"Unsupported message content type {content}")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='tool', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: GoogleTokenizer(api_key=self.api_key, model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `tool_choice = field(default='auto', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `top_k = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `top_p = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `use_native_tools = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__to_google_message_content(content)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def __to_google_message_content(self, content: BaseMessageContent) -> ContentDict |␣
↪Part | str:
    types = import_optional_dependency("google.generativeai.types")
    protos = import_optional_dependency("google.generativeai.protos")

    if isinstance(content, TextMessageContent):
        return content.artifact.to_text()
    if isinstance(content, ImageMessageContent):
        if isinstance(content.artifact, ImageArtifact):
            return types.ContentDict(mime_type=content.artifact.mime_type,␣
↪data=content.artifact.value)
        # TODO: Google requires uploading to the files endpoint: https://ai.google.
↪dev/gemini-api/docs/image-understanding#upload-image
        # Can be worked around by using GenericMessageContent, similar to videos.
        raise ValueError(f"Unsupported image artifact type: {type(content.artifact)}")
    if isinstance(content, ActionCallMessageContent):
        action = content.artifact.value

        return protos.Part(function_call=protos.FunctionCall(name=action.tag,␣
↪args=action.input))
    if isinstance(content, ActionResultMessageContent):
        artifact = content.artifact

        return protos.Part(
            function_response=protos.FunctionResponse(
                name=content.action.to_native_tool_name(),
                response=artifact.to_dict(),
            ),
        )
    if isinstance(content, GenericMessageContent):
        return content.artifact.value
    raise ValueError(f"Unsupported prompt stack content type: {type(content)}")
```

__to_google_messages(prompt_stack)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def __to_google_messages(self, prompt_stack: PromptStack) -> ContentsType:
    types = import_optional_dependency("google.generativeai.types")

    return [
        types.ContentDict(
            {
                "role": self.__to_google_role(message),
                "parts": [self.__to_google_message_content(content) for content in␣
↪message.content],
            },
        )
        for message in prompt_stack.messages
```

(continues on next page)

(continued from previous page)

```
        if not message.is_system()
    ]
```

__to_google_role(message)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def __to_google_role(self, message: Message) -> str:
    if message.is_assistant():
        return "model"
    return "user"
```

__to_google_tools(tools)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def __to_google_tools(self, tools: list[BaseTool]) -> list[dict]:
    types = import_optional_dependency("google.generativeai.types")

    tool_declarations = []
    for tool in tools:
        for activity in tool.activities():
            schema = tool.to_activity_json_schema(activity, "Parameters Schema")

            if "values" in schema["properties"]:
                schema = schema["properties"]["values"]

            schema = remove_key_in_dict_recursively(schema, "additionalProperties")
            tool_declaration = types.FunctionDeclaration(
                name=tool.to_native_tool_name(activity),
                description=tool.activity_description(activity),
                **(
                    {
                        "parameters": {
                            "type": schema["type"],
                            "properties": schema["properties"],
                            "required": schema.get("required", []),
                        }
                    }
                    if schema.get("properties")
                    else {}
                ),
            )

            tool_declarations.append(tool_declaration)

    return tool_declarations
```

__to_prompt_stack_delta_message_content(content)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def __to_prompt_stack_delta_message_content(self, content: Part) ->␣
↪BaseDeltaMessageContent:
    json_format = import_optional_dependency("google.protobuf.json_format")

    if content.text:
        return TextDeltaMessageContent(content.text)
    if content.function_call:
        function_call = content.function_call

        name, path = ToolAction.from_native_tool_name(function_call.name)

        args = json_format.MessageToDict(function_call._pb).get("args", {})
        return ActionCallDeltaMessageContent(
            tag=function_call.name,
            name=name,
            path=path,
            partial_input=json.dumps(args),
        )
    raise ValueError(f"Unsupported message content type {content}")
```

__to_prompt_stack_message_content(content)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def __to_prompt_stack_message_content(self, content: Part) -> BaseMessageContent:
    json_format = import_optional_dependency("google.protobuf.json_format")

    if content.text:
        return TextMessageContent(TextArtifact(content.text))
    if content.function_call:
        function_call = content.function_call

        name, path = ToolAction.from_native_tool_name(function_call.name)

        args = json_format.MessageToDict(function_call._pb).get("args", {})
        return ActionCallMessageContent(
            artifact=ActionArtifact(value=ToolAction(tag=function_call.name,␣
↪name=name, path=path, input=args)),
        )
    raise ValueError(f"Unsupported message content type {content}")
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    types = import_optional_dependency("google.generativeai.types")
    protos = import_optional_dependency("google.generativeai.protos")
```

(continues on next page)

(continued from previous page)

```
    system_messages = prompt_stack.system_messages
    if system_messages:
        self.client._system_instruction = types.ContentDict(
            role="system",
            parts=[protos.Part(text=system_message.to_text()) for system_message in␣
↪system_messages],
        )

    params = {
        "generation_config": types.GenerationConfig(
            **{
                # For some reason, providing stop sequences when streaming breaks␣
↪native functions
                # https://github.com/google-gemini/generative-ai-python/issues/446
                "stop_sequences": [] if self.stream and self.use_native_tools else␣
↪self.tokenizer.stop_sequences,
                "max_output_tokens": self.max_tokens,
                "temperature": self.temperature,
                "top_p": self.top_p,
                "top_k": self.top_k,
                **self.extra_params,
            },
        ),
    }

    if prompt_stack.tools and self.use_native_tools:
        params["tool_config"] = {"function_calling_config": {"mode": self.tool_choice}
↪}

        if prompt_stack.output_schema is not None and self.structured_output_strategy␣
↪== "tool":
            params["tool_config"]["function_calling_config"]["mode"] = "auto"

        params["tools"] = self.__to_google_tools(prompt_stack.tools)

    return params
```

client()

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
@lazy_property()
def client(self) -> GenerativeModel:
    genai = import_optional_dependency("google.generativeai")
    genai.configure(api_key=self.api_key)

    return genai.GenerativeModel(self.model)
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    messages = self.__to_google_messages(prompt_stack)
    params = self._base_params(prompt_stack)
    logger.debug((messages, params["generation_config"].__dict__))
    response: GenerateContentResponse = self.client.generate_content(messages,␣
↪**params)
    logger.debug(response.to_dict())

    usage_metadata = response.usage_metadata

    return Message(
        content=[self.__to_prompt_stack_message_content(part) for part in response.
↪parts],
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(
            input_tokens=usage_metadata.prompt_token_count,
            output_tokens=usage_metadata.candidates_token_count,
        ),
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    messages = self.__to_google_messages(prompt_stack)
    params = {**self._base_params(prompt_stack), "stream": True}
    logger.debug((messages, params))
    response: GenerateContentResponse = self.client.generate_content(
        messages,
        **params,
    )

    prompt_token_count = None
    for chunk in response:
        logger.debug(chunk.to_dict())
        usage_metadata = chunk.usage_metadata

        content = self.__to_prompt_stack_delta_message_content(chunk.parts[0]) if␣
↪chunk.parts else None
        # Only want to output the prompt token count once since it is static each␣
↪chunk
        if prompt_token_count is None:
            prompt_token_count = usage_metadata.prompt_token_count
            yield DeltaMessage(
                content=content,
                usage=DeltaMessage.Usage(
```

(continues on next page)

(continued from previous page)

```
                    input_tokens=usage_metadata.prompt_token_count,
                    output_tokens=usage_metadata.candidates_token_count,
                ),
            )
        else:
            yield DeltaMessage(
                content=content,
                usage=DeltaMessage.Usage(output_tokens=usage_metadata.candidates_
↪token_count),
            )
```

validate*structured_output_strategy*(, value)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
@structured_output_strategy.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
    if value == "native":
        raise ValueError(f"{__class__.__name__} does not support `{value}` structured␣
↪output strategy.")

    return value
```

google_prompt_driver

- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="google_prompt_dri tape.drivers.prompt.googl Driver.api_key" title="api_key = field(default=None, kw_only=True, meta- data={'serializable': False}) class-attribute instance-attribute (grip- tape.drivers.prompt.googl Driver.api_key)"><code class="language-python">	*Optional[str]*	Google API key.

continues on next page

Table 200 – continued from previous page

Name	Type	Description
<a href="google_prompt_driver.html#griptape.drivers.prompt.google_prompt_driver.GooglePromptDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.google_prompt_driver.GooglePromptDriver.model)"><code class="language-python">	*str*	Google model name.
<a href="google_prompt_driver.html#griptape.drivers.prompt.google_prompt_driver.GooglePromptDriver.client" title="client() (griptape.drivers.prompt.google_prompt_driver.GooglePromptDriver.client)"><code class="language-python">	*GenerativeModel*	Custom *GenerativeModel* client.
<a href="google_prompt_driver.html#griptape.drivers.prompt.google_prompt_driver.GooglePromptDriver.top_p" title="top_p = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.google_prompt_driver.GooglePromptDriver.top_p)"><code class="language-python">	*Optional[float]*	Optional value for top_p.
<a href="google_prompt_driver.html#griptape.drivers.prompt.google_prompt_driver.GooglePromptDriver.top_k" title="top_k = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.google_prompt_driver.GooglePromptDriver.top_k)"><code class="language-python">	*Optional[int]*	Optional value for top_k.

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
@define
class GooglePromptDriver(BasePromptDriver):
    """Google Prompt Driver.

    Attributes:
        api_key: Google API key.
        model: Google model name.
        client: Custom `GenerativeModel` client.
        top_p: Optional value for top_p.
        top_k: Optional value for top_k.
    """

    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    model: str = field(kw_only=True, metadata={"serializable": True})
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: GoogleTokenizer(api_key=self.api_key, model=self.
↪model), takes_self=True),
        kw_only=True,
    )
    top_p: Optional[float] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    top_k: Optional[int] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    use_native_tools: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})
    structured_output_strategy: StructuredOutputStrategy = field(
        default="tool", kw_only=True, metadata={"serializable": True}
    )
    tool_choice: str = field(default="auto", kw_only=True, metadata={"serializable":␣
↪True})
    _client: Optional[GenerativeModel] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @structured_output_strategy.validator  # pyright:␣
↪ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
        if value == "native":
            raise ValueError(f"{__class__.__name__} does not support `{value}`␣
↪structured output strategy.")

        return value

    @lazy_property()
    def client(self) -> GenerativeModel:
        genai = import_optional_dependency("google.generativeai")
        genai.configure(api_key=self.api_key)

        return genai.GenerativeModel(self.model)

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
```

(continues on next page)

(continued from previous page)

```
        messages = self.__to_google_messages(prompt_stack)
        params = self._base_params(prompt_stack)
        logger.debug((messages, params["generation_config"].__dict__))
        response: GenerateContentResponse = self.client.generate_content(messages,␣
↪**params)
        logger.debug(response.to_dict())

        usage_metadata = response.usage_metadata

        return Message(
            content=[self.__to_prompt_stack_message_content(part) for part in␣
↪response.parts],
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(
                input_tokens=usage_metadata.prompt_token_count,
                output_tokens=usage_metadata.candidates_token_count,
            ),
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        messages = self.__to_google_messages(prompt_stack)
        params = {**self._base_params(prompt_stack), "stream": True}
        logger.debug((messages, params))
        response: GenerateContentResponse = self.client.generate_content(
            messages,
            **params,
        )

        prompt_token_count = None
        for chunk in response:
            logger.debug(chunk.to_dict())
            usage_metadata = chunk.usage_metadata

            content = self.__to_prompt_stack_delta_message_content(chunk.parts[0]) if␣
↪chunk.parts else None
            # Only want to output the prompt token count once since it is static each␣
↪chunk
            if prompt_token_count is None:
                prompt_token_count = usage_metadata.prompt_token_count
                yield DeltaMessage(
                    content=content,
                    usage=DeltaMessage.Usage(
                        input_tokens=usage_metadata.prompt_token_count,
                        output_tokens=usage_metadata.candidates_token_count,
                    ),
                )
            else:
                yield DeltaMessage(
                    content=content,
                    usage=DeltaMessage.Usage(output_tokens=usage_metadata.candidates_
↪token_count),
```

(continues on next page)

(continued from previous page)

```
                )

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        types = import_optional_dependency("google.generativeai.types")
        protos = import_optional_dependency("google.generativeai.protos")

        system_messages = prompt_stack.system_messages
        if system_messages:
            self.client._system_instruction = types.ContentDict(
                role="system",
                parts=[protos.Part(text=system_message.to_text()) for system_message␣
↪in system_messages],
            )

        params = {
            "generation_config": types.GenerationConfig(
                **{
                    # For some reason, providing stop sequences when streaming breaks␣
↪native functions
                    # https://github.com/google-gemini/generative-ai-python/issues/446
                    "stop_sequences": [] if self.stream and self.use_native_tools␣
↪else self.tokenizer.stop_sequences,
                    "max_output_tokens": self.max_tokens,
                    "temperature": self.temperature,
                    "top_p": self.top_p,
                    "top_k": self.top_k,
                    **self.extra_params,
                },
            ),
        }

        if prompt_stack.tools and self.use_native_tools:
            params["tool_config"] = {"function_calling_config": {"mode": self.tool_
↪choice}}

            if prompt_stack.output_schema is not None and self.structured_output_
↪strategy == "tool":
                params["tool_config"]["function_calling_config"]["mode"] = "auto"

            params["tools"] = self.__to_google_tools(prompt_stack.tools)

        return params

    def __to_google_messages(self, prompt_stack: PromptStack) -> ContentsType:
        types = import_optional_dependency("google.generativeai.types")

        return [
            types.ContentDict(
                {
                    "role": self.__to_google_role(message),
                    "parts": [self.__to_google_message_content(content) for content␣
↪in message.content],
```

(continues on next page)

(continued from previous page)

```
                },
            )
            for message in prompt_stack.messages
            if not message.is_system()
        ]

    def __to_google_role(self, message: Message) -> str:
        if message.is_assistant():
            return "model"
        return "user"

    def __to_google_tools(self, tools: list[BaseTool]) -> list[dict]:
        types = import_optional_dependency("google.generativeai.types")

        tool_declarations = []
        for tool in tools:
            for activity in tool.activities():
                schema = tool.to_activity_json_schema(activity, "Parameters Schema")

                if "values" in schema["properties"]:
                    schema = schema["properties"]["values"]

                schema = remove_key_in_dict_recursively(schema, "additionalProperties
↪")
                tool_declaration = types.FunctionDeclaration(
                    name=tool.to_native_tool_name(activity),
                    description=tool.activity_description(activity),
                    **(
                        {
                            "parameters": {
                                "type": schema["type"],
                                "properties": schema["properties"],
                                "required": schema.get("required", []),
                            }
                        }
                        if schema.get("properties")
                        else {}
                    ),
                )

                tool_declarations.append(tool_declaration)

        return tool_declarations

    def __to_google_message_content(self, content: BaseMessageContent) -> ContentDict␣
↪| Part | str:
        types = import_optional_dependency("google.generativeai.types")
        protos = import_optional_dependency("google.generativeai.protos")

        if isinstance(content, TextMessageContent):
            return content.artifact.to_text()
        if isinstance(content, ImageMessageContent):
```

(continues on next page)

(continued from previous page)

```
            if isinstance(content.artifact, ImageArtifact):
                return types.ContentDict(mime_type=content.artifact.mime_type,␣
↪data=content.artifact.value)
            # TODO: Google requires uploading to the files endpoint: https://ai.
↪google.dev/gemini-api/docs/image-understanding#upload-image
            # Can be worked around by using GenericMessageContent, similar to videos.
            raise ValueError(f"Unsupported image artifact type: {type(content.
↪artifact)}")
        if isinstance(content, ActionCallMessageContent):
            action = content.artifact.value

            return protos.Part(function_call=protos.FunctionCall(name=action.tag,␣
↪args=action.input))
        if isinstance(content, ActionResultMessageContent):
            artifact = content.artifact

            return protos.Part(
                function_response=protos.FunctionResponse(
                    name=content.action.to_native_tool_name(),
                    response=artifact.to_dict(),
                ),
            )
        if isinstance(content, GenericMessageContent):
            return content.artifact.value
        raise ValueError(f"Unsupported prompt stack content type: {type(content)}")

    def __to_prompt_stack_message_content(self, content: Part) -> BaseMessageContent:
        json_format = import_optional_dependency("google.protobuf.json_format")

        if content.text:
            return TextMessageContent(TextArtifact(content.text))
        if content.function_call:
            function_call = content.function_call

            name, path = ToolAction.from_native_tool_name(function_call.name)

            args = json_format.MessageToDict(function_call._pb).get("args", {})
            return ActionCallMessageContent(
                artifact=ActionArtifact(value=ToolAction(tag=function_call.name,␣
↪name=name, path=path, input=args)),
            )
        raise ValueError(f"Unsupported message content type {content}")

    def __to_prompt_stack_delta_message_content(self, content: Part) ->␣
↪BaseDeltaMessageContent:
        json_format = import_optional_dependency("google.protobuf.json_format")

        if content.text:
            return TextDeltaMessageContent(content.text)
        if content.function_call:
            function_call = content.function_call
```

(continues on next page)

(continued from previous page)

```
            name, path = ToolAction.from_native_tool_name(function_call.name)

            args = json_format.MessageToDict(function_call._pb).get("args", {})
            return ActionCallDeltaMessageContent(
                tag=function_call.name,
                name=name,
                path=path,
                partial_input=json.dumps(args),
            )
        raise ValueError(f"Unsupported message content type {content}")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='tool', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: GoogleTokenizer(api_key=self.api_key, model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `tool_choice = field(default='auto', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `top_k = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `top_p = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `use_native_tools = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__to_google_message_content(content)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def __to_google_message_content(self, content: BaseMessageContent) -> ContentDict |␣
↪Part | str:
    types = import_optional_dependency("google.generativeai.types")
    protos = import_optional_dependency("google.generativeai.protos")

    if isinstance(content, TextMessageContent):
        return content.artifact.to_text()
    if isinstance(content, ImageMessageContent):
        if isinstance(content.artifact, ImageArtifact):
            return types.ContentDict(mime_type=content.artifact.mime_type,␣
↪data=content.artifact.value)
        # TODO: Google requires uploading to the files endpoint: https://ai.google.
↪dev/gemini-api/docs/image-understanding#upload-image
```

(continues on next page)

(continued from previous page)

```
        # Can be worked around by using GenericMessageContent, similar to videos.
        raise ValueError(f"Unsupported image artifact type: {type(content.artifact)}")
    if isinstance(content, ActionCallMessageContent):
        action = content.artifact.value

        return protos.Part(function_call=protos.FunctionCall(name=action.tag,␣
↪args=action.input))
    if isinstance(content, ActionResultMessageContent):
        artifact = content.artifact

        return protos.Part(
            function_response=protos.FunctionResponse(
                name=content.action.to_native_tool_name(),
                response=artifact.to_dict(),
            ),
        )
    if isinstance(content, GenericMessageContent):
        return content.artifact.value
    raise ValueError(f"Unsupported prompt stack content type: {type(content)}")
```

__to_google_messages(prompt_stack)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def __to_google_messages(self, prompt_stack: PromptStack) -> ContentsType:
    types = import_optional_dependency("google.generativeai.types")

    return [
        types.ContentDict(
            {
                "role": self.__to_google_role(message),
                "parts": [self.__to_google_message_content(content) for content in␣
↪message.content],
            },
        )
        for message in prompt_stack.messages
        if not message.is_system()
    ]
```

__to_google_role(message)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def __to_google_role(self, message: Message) -> str:
    if message.is_assistant():
        return "model"
    return "user"
```

__to_google_tools(tools)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def __to_google_tools(self, tools: list[BaseTool]) -> list[dict]:
    types = import_optional_dependency("google.generativeai.types")

    tool_declarations = []
    for tool in tools:
        for activity in tool.activities():
            schema = tool.to_activity_json_schema(activity, "Parameters Schema")

            if "values" in schema["properties"]:
                schema = schema["properties"]["values"]

            schema = remove_key_in_dict_recursively(schema, "additionalProperties")
            tool_declaration = types.FunctionDeclaration(
                name=tool.to_native_tool_name(activity),
                description=tool.activity_description(activity),
                **(
                    {
                        "parameters": {
                            "type": schema["type"],
                            "properties": schema["properties"],
                            "required": schema.get("required", []),
                        }
                    }
                    if schema.get("properties")
                    else {}
                ),
            )

            tool_declarations.append(tool_declaration)

    return tool_declarations
```

__to_prompt_stack_delta_message_content(content)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def __to_prompt_stack_delta_message_content(self, content: Part) ->␣
↪BaseDeltaMessageContent:
    json_format = import_optional_dependency("google.protobuf.json_format")

    if content.text:
        return TextDeltaMessageContent(content.text)
    if content.function_call:
        function_call = content.function_call

        name, path = ToolAction.from_native_tool_name(function_call.name)

        args = json_format.MessageToDict(function_call._pb).get("args", {})
        return ActionCallDeltaMessageContent(
```

(continues on next page)

(continued from previous page)

```
            tag=function_call.name,
            name=name,
            path=path,
            partial_input=json.dumps(args),
        )
    raise ValueError(f"Unsupported message content type {content}")
```

__to_prompt_stack_message_content(content)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def __to_prompt_stack_message_content(self, content: Part) -> BaseMessageContent:
    json_format = import_optional_dependency("google.protobuf.json_format")

    if content.text:
        return TextMessageContent(TextArtifact(content.text))
    if content.function_call:
        function_call = content.function_call

        name, path = ToolAction.from_native_tool_name(function_call.name)

        args = json_format.MessageToDict(function_call._pb).get("args", {})
        return ActionCallMessageContent(
            artifact=ActionArtifact(value=ToolAction(tag=function_call.name,␣
↪name=name, path=path, input=args)),
        )
    raise ValueError(f"Unsupported message content type {content}")
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    types = import_optional_dependency("google.generativeai.types")
    protos = import_optional_dependency("google.generativeai.protos")

    system_messages = prompt_stack.system_messages
    if system_messages:
        self.client._system_instruction = types.ContentDict(
            role="system",
            parts=[protos.Part(text=system_message.to_text()) for system_message in␣
↪system_messages],
        )

    params = {
        "generation_config": types.GenerationConfig(
            **{
                # For some reason, providing stop sequences when streaming breaks␣
↪native functions
                # https://github.com/google-gemini/generative-ai-python/issues/446
                "stop_sequences": [] if self.stream and self.use_native_tools else␣
```

(continues on next page)

(continued from previous page)

```
↪self.tokenizer.stop_sequences,
                "max_output_tokens": self.max_tokens,
                "temperature": self.temperature,
                "top_p": self.top_p,
                "top_k": self.top_k,
                **self.extra_params,
            },
        ),
    }

    if prompt_stack.tools and self.use_native_tools:
        params["tool_config"] = {"function_calling_config": {"mode": self.tool_choice}
↪}

        if prompt_stack.output_schema is not None and self.structured_output_strategy␣
↪== "tool":
            params["tool_config"]["function_calling_config"]["mode"] = "auto"

        params["tools"] = self.__to_google_tools(prompt_stack.tools)

    return params
```

client()

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
@lazy_property()
def client(self) -> GenerativeModel:
    genai = import_optional_dependency("google.generativeai")
    genai.configure(api_key=self.api_key)

    return genai.GenerativeModel(self.model)
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    messages = self.__to_google_messages(prompt_stack)
    params = self._base_params(prompt_stack)
    logger.debug((messages, params["generation_config"].__dict__))
    response: GenerateContentResponse = self.client.generate_content(messages,␣
↪**params)
    logger.debug(response.to_dict())

    usage_metadata = response.usage_metadata

    return Message(
        content=[self.__to_prompt_stack_message_content(part) for part in response.
↪parts],
```

(continues on next page)

(continued from previous page)

```
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(
            input_tokens=usage_metadata.prompt_token_count,
            output_tokens=usage_metadata.candidates_token_count,
        ),
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    messages = self.__to_google_messages(prompt_stack)
    params = {**self._base_params(prompt_stack), "stream": True}
    logger.debug((messages, params))
    response: GenerateContentResponse = self.client.generate_content(
        messages,
        **params,
    )

    prompt_token_count = None
    for chunk in response:
        logger.debug(chunk.to_dict())
        usage_metadata = chunk.usage_metadata

        content = self.__to_prompt_stack_delta_message_content(chunk.parts[0]) if␣
↪chunk.parts else None
        # Only want to output the prompt token count once since it is static each␣
↪chunk
        if prompt_token_count is None:
            prompt_token_count = usage_metadata.prompt_token_count
            yield DeltaMessage(
                content=content,
                usage=DeltaMessage.Usage(
                    input_tokens=usage_metadata.prompt_token_count,
                    output_tokens=usage_metadata.candidates_token_count,
                ),
            )
        else:
            yield DeltaMessage(
                content=content,
                usage=DeltaMessage.Usage(output_tokens=usage_metadata.candidates_
↪token_count),
            )
```

validate*structured_output_strategy(*, value)

Source Code in griptape/drivers/prompt/google_prompt_driver.py

```
@structured_output_strategy.validator  # pyright: ignore[reportAttributeAccessIssue,
↪reportOptionalMemberAccess]
def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
    if value == "native":
        raise ValueError(f"{__class__.__name__} does not support `{value}` structured
↪output strategy.")

    return value
```

griptape_cloud

- `__all__ = ['GriptapeCloudPromptDriver']` module-attribute

Bases: *BasePromptDriver* “)

Source Code in griptape/drivers/prompt/griptape_cloud_prompt_driver.py

```
@define
class GriptapeCloudPromptDriver(BasePromptDriver):
    model: Optional[str] = field(default=None, kw_only=True)
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]))
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},
↪takes_self=True), kw_only=True
    )
    tokenizer: BaseTokenizer = field(
        default=Factory(
            lambda self: SimpleTokenizer(
                characters_per_token=4,
                max_input_tokens=2000,
                max_output_tokens=self.max_tokens,
            ),
            takes_self=True,
        ),
        kw_only=True,
    )
    use_native_tools: bool = field(default=True, kw_only=True)
    structured_output_strategy: StructuredOutputStrategy = field(
        default="native", kw_only=True, metadata={"serializable": True}
    )

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        url = griptape_cloud_url(self.base_url, "api/chat/messages")

        params = self._base_params(prompt_stack)
        logger.debug(params)
        response = requests.post(url, headers=self.headers, json=params)
        response.raise_for_status()
```

(continues on next page)

(continued from previous page)

```
        response_json = response.json()
        logger.debug(response_json)

        return Message.from_dict(response_json)

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        url = griptape_cloud_url(self.base_url, "api/chat/messages/stream")
        params = self._base_params(prompt_stack)
        logger.debug(params)
        with requests.post(url, headers=self.headers, json=params, stream=True) as␣
↪response:
            response.raise_for_status()
            for line in response.iter_lines():
                if line:
                    decoded_line = line.decode("utf-8")
                    if decoded_line.startswith("data:"):
                        delta_message_payload = decoded_line.removeprefix("data:").
↪strip()
                        logger.debug(delta_message_payload)
                        yield DeltaMessage.from_json(delta_message_payload)

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        return {
            "messages": prompt_stack.to_dict()["messages"],
            "tools": self.__to_griptape_tools(prompt_stack.tools),
            **({"output_schema": prompt_stack.to_output_json_schema()} if prompt_
↪stack.output_schema else {}),
            "driver_configuration": {
                **({"model": self.model} if self.model else {}),
                "max_tokens": self.max_tokens,
                "use_native_tools": self.use_native_tools,
                "temperature": self.temperature,
                "structured_output_strategy": self.structured_output_strategy,
                "extra_params": self.extra_params,
            },
        }

    def __to_griptape_tools(self, tools: list[BaseTool]) -> list[dict]:
        return [
            {
                "name": tool.name,
                "activities": [
                    {
                        "name": activity.__name__,
                        "description": tool.activity_description(activity),
                        "json_schema": tool.to_activity_json_schema(activity, "Schema
↪"),
                    }
                    for activity in tool.activities()
                ],
            }
```

(continues on next page)

(continued from previous page)

```
            for tool in tools
        ]
```

- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']))` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')))` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `model = field(default=None, kw_only=True)` class-attribute instance-attribute
- `structured_output_strategy = field(default='native', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: SimpleTokenizer(characters_per_token=4, max_input_tokens=2000, max_output_tokens=self.max_tokens), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `use_native_tools = field(default=True, kw_only=True)` class-attribute instance-attribute

__to_griptape_tools(tools)

Source Code in griptape/drivers/prompt/griptape_cloud_prompt_driver.py

```
def __to_griptape_tools(self, tools: list[BaseTool]) -> list[dict]:
    return [
        {
            "name": tool.name,
            "activities": [
                {
                    "name": activity.__name__,
                    "description": tool.activity_description(activity),
                    "json_schema": tool.to_activity_json_schema(activity, "Schema"),
                }
                for activity in tool.activities()
            ],
        }
        for tool in tools
    ]
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/griptape_cloud_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    return {
        "messages": prompt_stack.to_dict()["messages"],
        "tools": self.__to_griptape_tools(prompt_stack.tools),
        **({"output_schema": prompt_stack.to_output_json_schema()} if prompt_stack.
↪output_schema else {}),
        "driver_configuration": {
            **({"model": self.model} if self.model else {}),
```

(continues on next page)

(continued from previous page)

```
            "max_tokens": self.max_tokens,
            "use_native_tools": self.use_native_tools,
            "temperature": self.temperature,
            "structured_output_strategy": self.structured_output_strategy,
            "extra_params": self.extra_params,
        },
    }
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/griptape_cloud_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    url = griptape_cloud_url(self.base_url, "api/chat/messages")

    params = self._base_params(prompt_stack)
    logger.debug(params)
    response = requests.post(url, headers=self.headers, json=params)
    response.raise_for_status()
    response_json = response.json()
    logger.debug(response_json)

    return Message.from_dict(response_json)
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/griptape_cloud_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    url = griptape_cloud_url(self.base_url, "api/chat/messages/stream")
    params = self._base_params(prompt_stack)
    logger.debug(params)
    with requests.post(url, headers=self.headers, json=params, stream=True) as␣
↪response:
        response.raise_for_status()
        for line in response.iter_lines():
            if line:
                decoded_line = line.decode("utf-8")
                if decoded_line.startswith("data:"):
                    delta_message_payload = decoded_line.removeprefix("data:").strip()
                    logger.debug(delta_message_payload)
                    yield DeltaMessage.from_json(delta_message_payload)
```

griptape_cloud_prompt_driver

- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *BasePromptDriver* “)

Source Code in griptape/drivers/prompt/griptape_cloud_prompt_driver.py

```
@define
class GriptapeCloudPromptDriver(BasePromptDriver):
    model: Optional[str] = field(default=None, kw_only=True)
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]))
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True), kw_only=True
    )
    tokenizer: BaseTokenizer = field(
        default=Factory(
            lambda self: SimpleTokenizer(
                characters_per_token=4,
                max_input_tokens=2000,
                max_output_tokens=self.max_tokens,
            ),
            takes_self=True,
        ),
        kw_only=True,
    )
    use_native_tools: bool = field(default=True, kw_only=True)
    structured_output_strategy: StructuredOutputStrategy = field(
        default="native", kw_only=True, metadata={"serializable": True}
    )

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        url = griptape_cloud_url(self.base_url, "api/chat/messages")

        params = self._base_params(prompt_stack)
        logger.debug(params)
        response = requests.post(url, headers=self.headers, json=params)
        response.raise_for_status()
        response_json = response.json()
        logger.debug(response_json)

        return Message.from_dict(response_json)

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        url = griptape_cloud_url(self.base_url, "api/chat/messages/stream")
        params = self._base_params(prompt_stack)
        logger.debug(params)
        with requests.post(url, headers=self.headers, json=params, stream=True) as␣
↪response:
            response.raise_for_status()
            for line in response.iter_lines():
                if line:
                    decoded_line = line.decode("utf-8")
                    if decoded_line.startswith("data:"):
```

(continues on next page)

(continued from previous page)

```
                        delta_message_payload = decoded_line.removeprefix("data:").
↪strip()
                        logger.debug(delta_message_payload)
                        yield DeltaMessage.from_json(delta_message_payload)

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        return {
            "messages": prompt_stack.to_dict()["messages"],
            "tools": self.__to_griptape_tools(prompt_stack.tools),
            **({"output_schema": prompt_stack.to_output_json_schema()} if prompt_
↪stack.output_schema else {}),
            "driver_configuration": {
                **({"model": self.model} if self.model else {}),
                "max_tokens": self.max_tokens,
                "use_native_tools": self.use_native_tools,
                "temperature": self.temperature,
                "structured_output_strategy": self.structured_output_strategy,
                "extra_params": self.extra_params,
            },
        }

    def __to_griptape_tools(self, tools: list[BaseTool]) -> list[dict]:
        return [
            {
                "name": tool.name,
                "activities": [
                    {
                        "name": activity.__name__,
                        "description": tool.activity_description(activity),
                        "json_schema": tool.to_activity_json_schema(activity, "Schema
↪"),
                    }
                    for activity in tool.activities()
                ],
            }
            for tool in tools
        ]
```

- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']))` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')))` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `model = field(default=None, kw_only=True)` class-attribute instance-attribute
- `structured_output_strategy = field(default='native', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: SimpleTokenizer(characters_per_token=4, max_input_tokens=2000, max_output_tokens=self.max_tokens), takes_self=True), kw_only=True)` class-attribute instance-attribute

- `use_native_tools = field(default=True, kw_only=True)` class-attribute instance-attribute

__to_griptape_tools(tools)

Source Code in griptape/drivers/prompt/griptape_cloud_prompt_driver.py

```
def __to_griptape_tools(self, tools: list[BaseTool]) -> list[dict]:
    return [
        {
            "name": tool.name,
            "activities": [
                {
                    "name": activity.__name__,
                    "description": tool.activity_description(activity),
                    "json_schema": tool.to_activity_json_schema(activity, "Schema"),
                }
                for activity in tool.activities()
            ],
        }
        for tool in tools
    ]
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/griptape_cloud_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    return {
        "messages": prompt_stack.to_dict()["messages"],
        "tools": self.__to_griptape_tools(prompt_stack.tools),
        **({"output_schema": prompt_stack.to_output_json_schema()} if prompt_stack.
↪output_schema else {}),
        "driver_configuration": {
            **({"model": self.model} if self.model else {}),
            "max_tokens": self.max_tokens,
            "use_native_tools": self.use_native_tools,
            "temperature": self.temperature,
            "structured_output_strategy": self.structured_output_strategy,
            "extra_params": self.extra_params,
        },
    }
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/griptape_cloud_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    url = griptape_cloud_url(self.base_url, "api/chat/messages")

    params = self._base_params(prompt_stack)
    logger.debug(params)
    response = requests.post(url, headers=self.headers, json=params)
```

(continues on next page)

(continued from previous page)

```
    response.raise_for_status()
    response_json = response.json()
    logger.debug(response_json)

    return Message.from_dict(response_json)
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/griptape_cloud_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    url = griptape_cloud_url(self.base_url, "api/chat/messages/stream")
    params = self._base_params(prompt_stack)
    logger.debug(params)
    with requests.post(url, headers=self.headers, json=params, stream=True) as␣
↪response:
        response.raise_for_status()
        for line in response.iter_lines():
            if line:
                decoded_line = line.decode("utf-8")
                if decoded_line.startswith("data:"):
                    delta_message_payload = decoded_line.removeprefix("data:").strip()
                    logger.debug(delta_message_payload)
                    yield DeltaMessage.from_json(delta_message_payload)
```

grok

- `__all__ = ['GrokPromptDriver']` module-attribute

Bases: *AzureOpenAiChatPromptDriver* “)

Source Code in griptape/drivers/prompt/grok_prompt_driver.py

```
@define
class GrokPromptDriver(OpenAiChatPromptDriver):
    base_url: str = field(default="https://api.x.ai/v1", kw_only=True, metadata={
↪"serializable": True})
    tokenizer: GrokTokenizer = field(
        default=Factory(
            lambda self: GrokTokenizer(base_url=self.base_url, api_key=self.api_key,␣
↪model=self.model), takes_self=True
        ),
        kw_only=True,
        metadata={"serializable": True},
    )
```

- `base_url = field(default='https://api.x.ai/v1', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: GrokTokenizer(base_url=self.base_url, api_key=self.api_key, model=self.model), takes_self=True), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

grok_prompt_driver

Bases: *AzureOpenAiChatPromptDriver* “)

Source Code in griptape/drivers/prompt/grok_prompt_driver.py

```
@define
class GrokPromptDriver(OpenAiChatPromptDriver):
    base_url: str = field(default="https://api.x.ai/v1", kw_only=True, metadata={
↪"serializable": True})
    tokenizer: GrokTokenizer = field(
        default=Factory(
            lambda self: GrokTokenizer(base_url=self.base_url, api_key=self.api_key,␣
↪model=self.model), takes_self=True
        ),
        kw_only=True,
        metadata={"serializable": True},
    )
```

- `base_url = field(default='https://api.x.ai/v1', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: GrokTokenizer(base_url=self.base_url, api_key=self.api_key, model=self.model), takes_self=True), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

huggingface_hub

- `__all__ = ['HuggingFaceHubPromptDriver']` module-attribute

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="huggingface_hub.mdx#griptape.drivers.prompt.huggingface_hub.HuggingFaceHubPromptDriver.api_token" title="api_token = field(kw_only=True, metadata={'serializable': True})">class-attribute instance-attribute (griptape.drivers.prompt.huggingface_hub.HuggingFaceHubPromptDriver.api_token)"><code class="language-python">	*str*	Hugging Face Hub API token.

continues on next page

Table 201 – continued from previous page

Name	Type	Description
use_gpu	*str*	Use GPU during model run.
<a href="hugging-face_hub.mdx#grip-tape.drivers.prompt.hug-gingface_hub.Hugging-FaceHubPrompt-Driver.model" title="model = field(kw_only=True, metadata={'seri-alizable': True}) class-attribute instance-attribute (grip-tape.drivers.prompt.hug-gingface_hub.Hugging-FaceHubPrompt-Driver.model)"><code class="language-python">	*str*	Hugging Face Hub model name.
<a href="hugging-face_hub.mdx#grip-tape.drivers.prompt.hug-gingface_hub.Hugging-FaceHubPrompt-Driver.client" ti-tle="client() (grip-tape.drivers.prompt.hug-gingface_hub.Hugging-FaceHubPrompt-Driver.client)"><code class="language-python">	*InferenceClient*	Custom *InferenceApi*.

continues on next page

Table 201 – continued from previous page

Name	Type	Description
<a href="huggingface_hub.mdx#griptape.drivers.prompt.huggingface_hub.HuggingFaceHubPromptDriver.tokenizer" title="tokenizer = field(default=Factory(lambda self: HuggingFaceTokenizer(model=self.model, max_output_tokens=self.max_tokens), takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.drivers.prompt.huggingface_hub.HuggingFaceHubPromptDriver.tokenizer)"><code class="language-python">	<a href="../../tokenizers#griptape.tokenizers.HuggingFaceTokenizer" title="HuggingFaceTokenizer (griptape.tokenizers.HuggingFaceTokenizer)"><code class="language-python">HuggingFaceTokenizer</coc	Custom *HuggingFaceTokenizer*.

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
@define
class HuggingFaceHubPromptDriver(BasePromptDriver):
    """Hugging Face Hub Prompt Driver.

    Attributes:
        api_token: Hugging Face Hub API token.
        use_gpu: Use GPU during model run.
        model: Hugging Face Hub model name.
        client: Custom `InferenceApi`.
        tokenizer: Custom `HuggingFaceTokenizer`.
    """

    api_token: str = field(kw_only=True, metadata={"serializable": True})
    max_tokens: int = field(default=250, kw_only=True, metadata={"serializable": True}
↪)
    model: str = field(kw_only=True, metadata={"serializable": True})
    structured_output_strategy: StructuredOutputStrategy = field(
        default="native", kw_only=True, metadata={"serializable": True}
    )
    tokenizer: HuggingFaceTokenizer = field(
        default=Factory(
            lambda self: HuggingFaceTokenizer(model=self.model, max_output_
↪tokens=self.max_tokens),
            takes_self=True,
```

(continues on next page)

(continued from previous page)

```
        ),
        kw_only=True,
    )
    _client: Optional[InferenceClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> InferenceClient:
        return import_optional_dependency("huggingface_hub").InferenceClient(
            model=self.model,
            token=self.api_token,
        )

    @structured_output_strategy.validator  # pyright:␣
↪ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
        if value == "tool":
            raise ValueError(f"{__class__.__name__} does not support `{value}`␣
↪structured output strategy.")

        return value

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        prompt = self.prompt_stack_to_string(prompt_stack)
        full_params = self._base_params(prompt_stack)
        logger.debug(
            {
                "prompt": prompt,
                **full_params,
            }
        )

        response = self.client.text_generation(
            prompt,
            **full_params,
        )
        logger.debug(response)
        input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))
        output_tokens = len(self.tokenizer.tokenizer.encode(response))  # pyright:␣
↪ignore[reportArgumentType]

        return Message(
            content=response,
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(input_tokens=input_tokens, output_tokens=output_
↪tokens),
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
```

(continues on next page)

(continued from previous page)

```
        prompt = self.prompt_stack_to_string(prompt_stack)
        full_params = {**self._base_params(prompt_stack), "stream": True}
        logger.debug(
            {
                "prompt": prompt,
                **full_params,
            }
        )

        response = self.client.text_generation(prompt, **full_params)

        input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))

        full_text = ""
        for token in response:
            logger.debug(token)
            full_text += token
            yield DeltaMessage(content=TextDeltaMessageContent(token, index=0))

        output_tokens = len(self.tokenizer.tokenizer.encode(full_text))  # pyright:␣
↪ignore[reportArgumentType]
        yield DeltaMessage(usage=DeltaMessage.Usage(input_tokens=input_tokens, output_
↪tokens=output_tokens))

    def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
        return self.tokenizer.tokenizer.decode(self.__prompt_stack_to_tokens(prompt_
↪stack))  # pyright: ignore[reportArgumentType]

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        params = {
            "return_full_text": False,
            "max_new_tokens": self.max_tokens,
            **self.extra_params,
        }

        if prompt_stack.output_schema and self.structured_output_strategy == "native":
            # https://huggingface.co/learn/cookbook/en/structured_generation#-
↪constrained-decoding
            output_schema = prompt_stack.to_output_json_schema()
            # Grammar does not support $schema and $id
            del output_schema["$schema"]
            del output_schema["$id"]
            params["grammar"] = {"type": "json", "value": output_schema}

        return params

    def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
        messages = []
        for message in prompt_stack.messages:
            if len(message.content) == 1:
                messages.append({"role": message.role, "content": message.to_text()})
            else:
```

(continues on next page)

(continued from previous page)

```
                raise ValueError("Invalid input content length.")

        return messages

    def __prompt_stack_to_tokens(self, prompt_stack: PromptStack) -> list[int]:
        messages = self._prompt_stack_to_messages(prompt_stack)
        tokens = self.tokenizer.tokenizer.apply_chat_template(messages, add_
↪generation_prompt=True, tokenize=True)

        if isinstance(tokens, list):
            return tokens  # pyright: ignore[reportReturnType] According to the␣
↪[docs](https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils
↪#transformers.PreTrainedTokenizerBase.apply_chat_template), the return type is␣
↪List[int].
        raise ValueError("Invalid output type.")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_token = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `max_tokens = field(default=250, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='native', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: HuggingFaceTokenizer(model=self.model, max_output_tokens=self.max_tokens), takes_self=True), kw_only=True)` class-attribute instance-attribute

__prompt_stack_to_tokens(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
def __prompt_stack_to_tokens(self, prompt_stack: PromptStack) -> list[int]:
    messages = self._prompt_stack_to_messages(prompt_stack)
    tokens = self.tokenizer.tokenizer.apply_chat_template(messages, add_generation_
↪prompt=True, tokenize=True)

    if isinstance(tokens, list):
        return tokens  # pyright: ignore[reportReturnType] According to the␣
↪[docs](https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils
↪#transformers.PreTrainedTokenizerBase.apply_chat_template), the return type is␣
↪List[int].
    raise ValueError("Invalid output type.")
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    params = {
        "return_full_text": False,
        "max_new_tokens": self.max_tokens,
        **self.extra_params,
    }

    if prompt_stack.output_schema and self.structured_output_strategy == "native":
        # https://huggingface.co/learn/cookbook/en/structured_generation#-constrained-
↪decoding
        output_schema = prompt_stack.to_output_json_schema()
        # Grammar does not support $schema and $id
        del output_schema["$schema"]
        del output_schema["$id"]
        params["grammar"] = {"type": "json", "value": output_schema}

    return params
```

_prompt_stack_to_messages(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
    messages = []
    for message in prompt_stack.messages:
        if len(message.content) == 1:
            messages.append({"role": message.role, "content": message.to_text()})
        else:
            raise ValueError("Invalid input content length.")

    return messages
```

client()

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
@lazy_property()
def client(self) -> InferenceClient:
    return import_optional_dependency("huggingface_hub").InferenceClient(
        model=self.model,
        token=self.api_token,
    )
```

prompt_stack_to_string(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
    return self.tokenizer.tokenizer.decode(self.__prompt_stack_to_tokens(prompt_
↪stack))  # pyright: ignore[reportArgumentType]
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    prompt = self.prompt_stack_to_string(prompt_stack)
    full_params = self._base_params(prompt_stack)
    logger.debug(
        {
            "prompt": prompt,
            **full_params,
        }
    )

    response = self.client.text_generation(
        prompt,
        **full_params,
    )
    logger.debug(response)
    input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))
    output_tokens = len(self.tokenizer.tokenizer.encode(response))  # pyright:␣
↪ignore[reportArgumentType]

    return Message(
        content=response,
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(input_tokens=input_tokens, output_tokens=output_tokens),
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    prompt = self.prompt_stack_to_string(prompt_stack)
    full_params = {**self._base_params(prompt_stack), "stream": True}
    logger.debug(
        {
            "prompt": prompt,
            **full_params,
        }
    )

    response = self.client.text_generation(prompt, **full_params)

    input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))
```

(continues on next page)

(continued from previous page)

```
    full_text = ""
    for token in response:
        logger.debug(token)
        full_text += token
        yield DeltaMessage(content=TextDeltaMessageContent(token, index=0))

    output_tokens = len(self.tokenizer.tokenizer.encode(full_text))  # pyright:␣
↪ignore[reportArgumentType]
    yield DeltaMessage(usage=DeltaMessage.Usage(input_tokens=input_tokens, output_
↪tokens=output_tokens))
```

validate*structured_output_strategy(*, value)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
@structured_output_strategy.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
    if value == "tool":
        raise ValueError(f"{__class__.__name__} does not support `{value}` structured␣
↪output strategy.")

    return value
```

huggingface_hub_prompt_driver

- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="hugging-face_hub_prompt_driver.tape.drivers.prompt.hugging-face_hub_prompt_driver.gingFaceHubPrompt-Driver.api_token" title="api_token = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.hugging-face_hub_prompt_driver.gingFaceHubPrompt-Driver.api_token)"><code class="language-python">	*str*	Hugging Face Hub API token.
use_gpu	*str*	Use GPU during model run.
<a href="hugging-face_hub_prompt_driver.tape.drivers.prompt.hugging-face_hub_prompt_driver.gingFaceHubPrompt-Driver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.hugging-face_hub_prompt_driver.gingFaceHubPrompt-Driver.model)"><code class="language-python">	*str*	Hugging Face Hub model name.

continues on next page

Table 202 – continued from previous page

Name	Type	Description
<a href="hugging-face_hub_prompt_driver.tape.drivers.prompt.hug-ging-face_hub_prompt_driver.gingFaceHubPrompt-Driver.client" title="client() (grip-tape.drivers.prompt.hug-ging-face_hub_prompt_driver.gingFaceHubPrompt-Driver.client)"><code class="language-python">	*InferenceClient*	Custom *InferenceApi*.
<a href="hugging-face_hub_prompt_driver.tape.drivers.prompt.hug-ging-face_hub_prompt_driver.gingFaceHubPrompt-Driver.tokenizer" title="tokenizer = field(default=Fac-tory(lambda self: HuggingFaceTok-enizer(model=self.model, max_output_to-kens=self.max_tokens), takes_self=True), kw_only=True) class-attribute instance-attribute (grip-tape.drivers.prompt.hug-ging-face_hub_prompt_driver.gingFaceHubPrompt-Driver.tok-enizer)"><code class="language-python">	<a href="../../tokenizers#griptape.tokenizers.Hug-gingFaceTokenizer" title="HuggingFaceTokenizer (griptape.tokenizers.HuggingFaceTokenizer)"><code class="language-python">HuggingFaceTokenizer</co	Custom *HuggingFaceTok-enizer*.

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
@define
class HuggingFaceHubPromptDriver(BasePromptDriver):
    """Hugging Face Hub Prompt Driver.

    Attributes:
        api_token: Hugging Face Hub API token.
        use_gpu: Use GPU during model run.
        model: Hugging Face Hub model name.
```

(continues on next page)

(continued from previous page)

```
        client: Custom `InferenceApi`.
        tokenizer: Custom `HuggingFaceTokenizer`.
    """

    api_token: str = field(kw_only=True, metadata={"serializable": True})
    max_tokens: int = field(default=250, kw_only=True, metadata={"serializable": True}
↪)
    model: str = field(kw_only=True, metadata={"serializable": True})
    structured_output_strategy: StructuredOutputStrategy = field(
        default="native", kw_only=True, metadata={"serializable": True}
    )
    tokenizer: HuggingFaceTokenizer = field(
        default=Factory(
            lambda self: HuggingFaceTokenizer(model=self.model, max_output_
↪tokens=self.max_tokens),
            takes_self=True,
        ),
        kw_only=True,
    )
    _client: Optional[InferenceClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> InferenceClient:
        return import_optional_dependency("huggingface_hub").InferenceClient(
            model=self.model,
            token=self.api_token,
        )

    @structured_output_strategy.validator  # pyright:␣
↪ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
        if value == "tool":
            raise ValueError(f"{__class__.__name__} does not support `{value}`␣
↪structured output strategy.")

        return value

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        prompt = self.prompt_stack_to_string(prompt_stack)
        full_params = self._base_params(prompt_stack)
        logger.debug(
            {
                "prompt": prompt,
                **full_params,
            }
        )

        response = self.client.text_generation(
            prompt,
```

(continues on next page)

(continued from previous page)

```
            **full_params,
        )
        logger.debug(response)
        input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))
        output_tokens = len(self.tokenizer.tokenizer.encode(response))  # pyright:␣
↪ignore[reportArgumentType]

        return Message(
            content=response,
            role=Message.ASSISTANT_ROLE,
            usage=Message.Usage(input_tokens=input_tokens, output_tokens=output_
↪tokens),
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        prompt = self.prompt_stack_to_string(prompt_stack)
        full_params = {**self._base_params(prompt_stack), "stream": True}
        logger.debug(
            {
                "prompt": prompt,
                **full_params,
            }
        )

        response = self.client.text_generation(prompt, **full_params)

        input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))

        full_text = ""
        for token in response:
            logger.debug(token)
            full_text += token
            yield DeltaMessage(content=TextDeltaMessageContent(token, index=0))

        output_tokens = len(self.tokenizer.tokenizer.encode(full_text))  # pyright:␣
↪ignore[reportArgumentType]
        yield DeltaMessage(usage=DeltaMessage.Usage(input_tokens=input_tokens, output_
↪tokens=output_tokens))

    def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
        return self.tokenizer.tokenizer.decode(self.__prompt_stack_to_tokens(prompt_
↪stack))  # pyright: ignore[reportArgumentType]

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        params = {
            "return_full_text": False,
            "max_new_tokens": self.max_tokens,
            **self.extra_params,
        }

        if prompt_stack.output_schema and self.structured_output_strategy == "native":
```

(continues on next page)

(continued from previous page)

```
            # https://huggingface.co/learn/cookbook/en/structured_generation#-
↪constrained-decoding
            output_schema = prompt_stack.to_output_json_schema()
            # Grammar does not support $schema and $id
            del output_schema["$schema"]
            del output_schema["$id"]
            params["grammar"] = {"type": "json", "value": output_schema}

        return params

    def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
        messages = []
        for message in prompt_stack.messages:
            if len(message.content) == 1:
                messages.append({"role": message.role, "content": message.to_text()})
            else:
                raise ValueError("Invalid input content length.")

        return messages

    def __prompt_stack_to_tokens(self, prompt_stack: PromptStack) -> list[int]:
        messages = self._prompt_stack_to_messages(prompt_stack)
        tokens = self.tokenizer.tokenizer.apply_chat_template(messages, add_
↪generation_prompt=True, tokenize=True)

        if isinstance(tokens, list):
            return tokens  # pyright: ignore[reportReturnType] According to the␣
↪[docs](https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils
↪#transformers.PreTrainedTokenizerBase.apply_chat_template), the return type is␣
↪List[int].
        raise ValueError("Invalid output type.")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_token = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `max_tokens = field(default=250, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='native', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: HuggingFaceTokenizer(model=self.model, max_output_tokens=self.max_tokens), takes_self=True), kw_only=True)` class-attribute instance-attribute

__prompt_stack_to_tokens(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
def __prompt_stack_to_tokens(self, prompt_stack: PromptStack) -> list[int]:
    messages = self._prompt_stack_to_messages(prompt_stack)
    tokens = self.tokenizer.tokenizer.apply_chat_template(messages, add_generation_
↪prompt=True, tokenize=True)

    if isinstance(tokens, list):
        return tokens  # pyright: ignore[reportReturnType] According to the␣
↪[docs](https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils
↪#transformers.PreTrainedTokenizerBase.apply_chat_template), the return type is␣
↪List[int].
    raise ValueError("Invalid output type.")
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    params = {
        "return_full_text": False,
        "max_new_tokens": self.max_tokens,
        **self.extra_params,
    }

    if prompt_stack.output_schema and self.structured_output_strategy == "native":
        # https://huggingface.co/learn/cookbook/en/structured_generation#-constrained-
↪decoding
        output_schema = prompt_stack.to_output_json_schema()
        # Grammar does not support $schema and $id
        del output_schema["$schema"]
        del output_schema["$id"]
        params["grammar"] = {"type": "json", "value": output_schema}

    return params
```

_prompt_stack_to_messages(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
    messages = []
    for message in prompt_stack.messages:
        if len(message.content) == 1:
            messages.append({"role": message.role, "content": message.to_text()})
        else:
            raise ValueError("Invalid input content length.")

    return messages
```

client()

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
@lazy_property()
def client(self) -> InferenceClient:
    return import_optional_dependency("huggingface_hub").InferenceClient(
        model=self.model,
        token=self.api_token,
    )
```

prompt_stack_to_string(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
    return self.tokenizer.tokenizer.decode(self.__prompt_stack_to_tokens(prompt_
↪stack))  # pyright: ignore[reportArgumentType]
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    prompt = self.prompt_stack_to_string(prompt_stack)
    full_params = self._base_params(prompt_stack)
    logger.debug(
        {
            "prompt": prompt,
            **full_params,
        }
    )

    response = self.client.text_generation(
        prompt,
        **full_params,
    )
    logger.debug(response)
    input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))
    output_tokens = len(self.tokenizer.tokenizer.encode(response))  # pyright:␣
↪ignore[reportArgumentType]

    return Message(
        content=response,
        role=Message.ASSISTANT_ROLE,
        usage=Message.Usage(input_tokens=input_tokens, output_tokens=output_tokens),
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    prompt = self.prompt_stack_to_string(prompt_stack)
    full_params = {**self._base_params(prompt_stack), "stream": True}
    logger.debug(
        {
            "prompt": prompt,
            **full_params,
        }
    )

    response = self.client.text_generation(prompt, **full_params)

    input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))

    full_text = ""
    for token in response:
        logger.debug(token)
        full_text += token
        yield DeltaMessage(content=TextDeltaMessageContent(token, index=0))

    output_tokens = len(self.tokenizer.tokenizer.encode(full_text))  # pyright:␣
↪ignore[reportArgumentType]
    yield DeltaMessage(usage=DeltaMessage.Usage(input_tokens=input_tokens, output_
↪tokens=output_tokens))
```

validate*structured_output_strategy(*, value)

Source Code in griptape/drivers/prompt/huggingface_hub_prompt_driver.py

```
@structured_output_strategy.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
    if value == "tool":
        raise ValueError(f"{__class__.__name__} does not support `{value}` structured␣
↪output strategy.")

    return value
```

huggingface_pipeline

- `__all__ = ['HuggingFacePipelinePromptDriver']` module-attribute

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="huggingface_pipeline.mdx#griptape.drivers.prompt.huggingface_pipeline.HuggingFacePipelinePromptDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.huggingface_pipeline.HuggingFacePipelinePromptDriver.model)"><code class="language-python">	*str*	Hugging Face Hub model name.

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
@define
class HuggingFacePipelinePromptDriver(BasePromptDriver):
    """Hugging Face Pipeline Prompt Driver.

    Attributes:
        model: Hugging Face Hub model name.
    """

    max_tokens: int = field(default=250, kw_only=True, metadata={"serializable": True}
↪)
    model: str = field(kw_only=True, metadata={"serializable": True})
    tokenizer: HuggingFaceTokenizer = field(
        default=Factory(
            lambda self: HuggingFaceTokenizer(model=self.model, max_output_
↪tokens=self.max_tokens),
            takes_self=True,
        ),
        kw_only=True,
    )
    structured_output_strategy: StructuredOutputStrategy = field(
        default="rule", kw_only=True, metadata={"serializable": True}
    )
    _pipeline: Optional[TextGenerationPipeline] = field(
        default=None, kw_only=True, alias="pipeline", metadata={"serializable": False}
    )

    @structured_output_strategy.validator  # pyright:␣
↪ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
```

(continues on next page)

(continued from previous page)

```
    def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
        if value in ("native", "tool"):
            raise ValueError(f"{__class__.__name__} does not support `{value}`␣
↪structured output strategy.")

        return value

    @lazy_property()
    def pipeline(self) -> TextGenerationPipeline:
        return import_optional_dependency("transformers").pipeline(
            task="text-generation",
            model=self.model,
            max_new_tokens=self.max_tokens,
            tokenizer=self.tokenizer.tokenizer,
        )

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        messages = self._prompt_stack_to_messages(prompt_stack)
        full_params = self._base_params(prompt_stack)
        logger.debug(
            (
                messages,
                full_params,
            )
        )

        result = self.pipeline(messages, **full_params)
        logger.debug(result)

        if isinstance(result, list):
            if len(result) == 1:
                generated_text = result[0]["generated_text"][-1]["content"]

                input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))
                output_tokens = len(self.tokenizer.tokenizer.encode(generated_text))
↪# pyright: ignore[reportArgumentType]

                return Message(
                    content=[TextMessageContent(TextArtifact(generated_text))],
                    role=Message.ASSISTANT_ROLE,
                    usage=Message.Usage(input_tokens=input_tokens, output_
↪tokens=output_tokens),
                )
            raise Exception("completion with more than one choice is not supported yet
↪")
        raise Exception("invalid output format")

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        raise NotImplementedError("streaming is not supported")
```

(continues on next page)

(continued from previous page)

```
    def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
        return self.tokenizer.tokenizer.decode(self.__prompt_stack_to_tokens(prompt_
↪stack))  # pyright: ignore[reportArgumentType]

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        return {
            "max_new_tokens": self.max_tokens,
            "temperature": self.temperature,
            "do_sample": True,
            **self.extra_params,
        }

    def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
        messages = []

        for message in prompt_stack.messages:
            messages.append({"role": message.role, "content": message.to_text()})

        return messages

    def __prompt_stack_to_tokens(self, prompt_stack: PromptStack) -> list[int]:
        messages = self._prompt_stack_to_messages(prompt_stack)
        tokens = self.tokenizer.tokenizer.apply_chat_template(messages, add_
↪generation_prompt=True, tokenize=True)

        if isinstance(tokens, list):
            return tokens  # pyright: ignore[reportReturnType] According to the␣
↪[docs](https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils
↪#transformers.PreTrainedTokenizerBase.apply_chat_template), the return type is␣
↪List[int].
        raise ValueError("Invalid output type.")
```

- `_pipeline = field(default=None, kw_only=True, alias='pipeline', metadata={'serializable': False})` class-attribute instance-attribute
- `max_tokens = field(default=250, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='rule', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: HuggingFaceTokenizer(model=self.model, max_output_tokens=self.max_tokens), takes_self=True), kw_only=True)` class-attribute instance-attribute

__prompt_stack_to_tokens(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
def __prompt_stack_to_tokens(self, prompt_stack: PromptStack) -> list[int]:
    messages = self._prompt_stack_to_messages(prompt_stack)
```

(continues on next page)

(continued from previous page)

```
    tokens = self.tokenizer.tokenizer.apply_chat_template(messages, add_generation_
↪prompt=True, tokenize=True)

    if isinstance(tokens, list):
        return tokens  # pyright: ignore[reportReturnType] According to the␣
↪[docs](https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils
↪#transformers.PreTrainedTokenizerBase.apply_chat_template), the return type is␣
↪List[int].
    raise ValueError("Invalid output type.")
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    return {
        "max_new_tokens": self.max_tokens,
        "temperature": self.temperature,
        "do_sample": True,
        **self.extra_params,
    }
```

_prompt_stack_to_messages(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
    messages = []

    for message in prompt_stack.messages:
        messages.append({"role": message.role, "content": message.to_text()})

    return messages
```

pipeline()

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
@lazy_property()
def pipeline(self) -> TextGenerationPipeline:
    return import_optional_dependency("transformers").pipeline(
        task="text-generation",
        model=self.model,
        max_new_tokens=self.max_tokens,
        tokenizer=self.tokenizer.tokenizer,
    )
```

prompt_stack_to_string(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
    return self.tokenizer.tokenizer.decode(self.__prompt_stack_to_tokens(prompt_
↪stack))  # pyright: ignore[reportArgumentType]
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    messages = self._prompt_stack_to_messages(prompt_stack)
    full_params = self._base_params(prompt_stack)
    logger.debug(
        (
            messages,
            full_params,
        )
    )

    result = self.pipeline(messages, **full_params)
    logger.debug(result)

    if isinstance(result, list):
        if len(result) == 1:
            generated_text = result[0]["generated_text"][-1]["content"]

            input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))
            output_tokens = len(self.tokenizer.tokenizer.encode(generated_text))  #␣
↪pyright: ignore[reportArgumentType]

            return Message(
                content=[TextMessageContent(TextArtifact(generated_text))],
                role=Message.ASSISTANT_ROLE,
                usage=Message.Usage(input_tokens=input_tokens, output_tokens=output_
↪tokens),
            )
        raise Exception("completion with more than one choice is not supported yet")
    raise Exception("invalid output format")
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    raise NotImplementedError("streaming is not supported")
```

validate*structured_output_strategy*(, value)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
@structured_output_strategy.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
    if value in ("native", "tool"):
        raise ValueError(f"{__class__.__name__} does not support `{value}` structured␣
↪output strategy.")

    return value
```

huggingface_pipeline_prompt_driver

- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="hugging-face_pipeline_prompt_dri tape.drivers.prompt.hug-ging-face_pipeline_prompt_dri gingFacePipeline-PromptDriver.model" title="model = field(kw_only=True, metadata={'seri-alizable': True}) class-attribute instance-attribute (grip-tape.drivers.prompt.hug-ging-face_pipeline_prompt_dri gingFacePipeline-Prompt-Driver.model)"><code class="language-python">	*str*	Hugging Face Hub model name.

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
@define
class HuggingFacePipelinePromptDriver(BasePromptDriver):
    """Hugging Face Pipeline Prompt Driver.

    Attributes:
        model: Hugging Face Hub model name.
    """
```

(continues on next page)

(continued from previous page)

```
    max_tokens: int = field(default=250, kw_only=True, metadata={"serializable": True}
↪)
    model: str = field(kw_only=True, metadata={"serializable": True})
    tokenizer: HuggingFaceTokenizer = field(
        default=Factory(
            lambda self: HuggingFaceTokenizer(model=self.model, max_output_
↪tokens=self.max_tokens),
            takes_self=True,
        ),
        kw_only=True,
    )
    structured_output_strategy: StructuredOutputStrategy = field(
        default="rule", kw_only=True, metadata={"serializable": True}
    )
    _pipeline: Optional[TextGenerationPipeline] = field(
        default=None, kw_only=True, alias="pipeline", metadata={"serializable": False}
    )

    @structured_output_strategy.validator  # pyright:␣
↪ignore[reportAttributeAccessIssue, reportOptionalMemberAccess]
    def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
        if value in ("native", "tool"):
            raise ValueError(f"{__class__.__name__} does not support `{value}`␣
↪structured output strategy.")

        return value

    @lazy_property()
    def pipeline(self) -> TextGenerationPipeline:
        return import_optional_dependency("transformers").pipeline(
            task="text-generation",
            model=self.model,
            max_new_tokens=self.max_tokens,
            tokenizer=self.tokenizer.tokenizer,
        )

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        messages = self._prompt_stack_to_messages(prompt_stack)
        full_params = self._base_params(prompt_stack)
        logger.debug(
            (
                messages,
                full_params,
            )
        )

        result = self.pipeline(messages, **full_params)
        logger.debug(result)

        if isinstance(result, list):
```

(continues on next page)

(continued from previous page)

```
            if len(result) == 1:
                generated_text = result[0]["generated_text"][-1]["content"]

                input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))
                output_tokens = len(self.tokenizer.tokenizer.encode(generated_text))
↪# pyright: ignore[reportArgumentType]

                return Message(
                    content=[TextMessageContent(TextArtifact(generated_text))],
                    role=Message.ASSISTANT_ROLE,
                    usage=Message.Usage(input_tokens=input_tokens, output_
↪tokens=output_tokens),
                )
            raise Exception("completion with more than one choice is not supported yet
↪")
        raise Exception("invalid output format")

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        raise NotImplementedError("streaming is not supported")

    def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
        return self.tokenizer.tokenizer.decode(self.__prompt_stack_to_tokens(prompt_
↪stack))  # pyright: ignore[reportArgumentType]

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        return {
            "max_new_tokens": self.max_tokens,
            "temperature": self.temperature,
            "do_sample": True,
            **self.extra_params,
        }

    def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
        messages = []

        for message in prompt_stack.messages:
            messages.append({"role": message.role, "content": message.to_text()})

        return messages

    def __prompt_stack_to_tokens(self, prompt_stack: PromptStack) -> list[int]:
        messages = self._prompt_stack_to_messages(prompt_stack)
        tokens = self.tokenizer.tokenizer.apply_chat_template(messages, add_
↪generation_prompt=True, tokenize=True)

        if isinstance(tokens, list):
            return tokens  # pyright: ignore[reportReturnType] According to the␣
↪[docs](https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils
↪#transformers.PreTrainedTokenizerBase.apply_chat_template), the return type is␣
↪List[int].
        raise ValueError("Invalid output type.")
```

- `_pipeline = field(default=None, kw_only=True, alias='pipeline', metadata={'serializable': False})` class-attribute instance-attribute
- `max_tokens = field(default=250, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='rule', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: HuggingFaceTokenizer(model=self.model, max_output_tokens=self.max_tokens), takes_self=True), kw_only=True)` class-attribute instance-attribute

__prompt_stack_to_tokens(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
def __prompt_stack_to_tokens(self, prompt_stack: PromptStack) -> list[int]:
    messages = self._prompt_stack_to_messages(prompt_stack)
    tokens = self.tokenizer.tokenizer.apply_chat_template(messages, add_generation_
↪prompt=True, tokenize=True)

    if isinstance(tokens, list):
        return tokens  # pyright: ignore[reportReturnType] According to the␣
↪[docs](https://huggingface.co/docs/transformers/main/en/internal/tokenization_utils
↪#transformers.PreTrainedTokenizerBase.apply_chat_template), the return type is␣
↪List[int].
    raise ValueError("Invalid output type.")
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    return {
        "max_new_tokens": self.max_tokens,
        "temperature": self.temperature,
        "do_sample": True,
        **self.extra_params,
    }
```

_prompt_stack_to_messages(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
    messages = []

    for message in prompt_stack.messages:
        messages.append({"role": message.role, "content": message.to_text()})

    return messages
```

pipeline()

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
@lazy_property()
def pipeline(self) -> TextGenerationPipeline:
    return import_optional_dependency("transformers").pipeline(
        task="text-generation",
        model=self.model,
        max_new_tokens=self.max_tokens,
        tokenizer=self.tokenizer.tokenizer,
    )
```

prompt_stack_to_string(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
def prompt_stack_to_string(self, prompt_stack: PromptStack) -> str:
    return self.tokenizer.tokenizer.decode(self.__prompt_stack_to_tokens(prompt_
↪stack))  # pyright: ignore[reportArgumentType]
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    messages = self._prompt_stack_to_messages(prompt_stack)
    full_params = self._base_params(prompt_stack)
    logger.debug(
        (
            messages,
            full_params,
        )
    )

    result = self.pipeline(messages, **full_params)
    logger.debug(result)

    if isinstance(result, list):
        if len(result) == 1:
            generated_text = result[0]["generated_text"][-1]["content"]

            input_tokens = len(self.__prompt_stack_to_tokens(prompt_stack))
            output_tokens = len(self.tokenizer.tokenizer.encode(generated_text))  #␣
↪pyright: ignore[reportArgumentType]

            return Message(
                content=[TextMessageContent(TextArtifact(generated_text))],
                role=Message.ASSISTANT_ROLE,
                usage=Message.Usage(input_tokens=input_tokens, output_tokens=output_
↪tokens),
            )
```

(continues on next page)

(continued from previous page)

```
        raise Exception("completion with more than one choice is not supported yet")
    raise Exception("invalid output format")
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    raise NotImplementedError("streaming is not supported")
```

validate*structured_output_strategy(*, value)

Source Code in griptape/drivers/prompt/huggingface_pipeline_prompt_driver.py

```
@structured_output_strategy.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_structured_output_strategy(self, _: Attribute, value: str) -> str:
    if value in ("native", "tool"):
        raise ValueError(f"{__class__.__name__} does not support `{value}` structured␣
↪output strategy.")

    return value
```

ollama

- `__all__ = ['OllamaPromptDriver']` module-attribute

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="ollama.mdx#griptape.drivers.prompt.ollama.OllamaPromptDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.ollama.OllamaPromptDriver.model)"><code class="language-python">	*str*	Model name.

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
@define
class OllamaPromptDriver(BasePromptDriver):
    """Ollama Prompt Driver.

    Attributes:
        model: Model name.
    """

    model: str = field(kw_only=True, metadata={"serializable": True})
    host: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    tokenizer: BaseTokenizer = field(
        default=Factory(
            lambda self: SimpleTokenizer(
                characters_per_token=4,
                max_input_tokens=2000,
                max_output_tokens=self.max_tokens,
            ),
            takes_self=True,
        ),
        kw_only=True,
    )
    options: dict = field(
        default=Factory(
            lambda self: {
                "temperature": self.temperature,
                "stop": self.tokenizer.stop_sequences,
                "num_predict": self.max_tokens,
            },
            takes_self=True,
        ),
        kw_only=True,
    )
    use_native_tools: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})
    _client: Optional[Client] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> Client:
        return import_optional_dependency("ollama").Client(host=self.host)

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        params = self._base_params(prompt_stack)
        logger.debug(params)
        response = self.client.chat(**params)
        logger.debug(response.model_dump())

        return Message(
            content=self.__to_prompt_stack_message_content(response),
            role=Message.ASSISTANT_ROLE,
        )
```

(continues on next page)

(continued from previous page)

```
    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        params = {**self._base_params(prompt_stack), "stream": True}
        logger.debug(params)
        stream: Iterator = self.client.chat(**params)

        tool_index = 0
        for chunk in stream:
            logger.debug(chunk)
            message_content = self.__to_prompt_stack_delta_message_content(chunk)
            # Ollama provides multiple Tool calls as separate chunks but with no␣
↪index to differentiate them.
            # So we must keep track of the index ourselves.
            if isinstance(message_content, ActionCallDeltaMessageContent):
                message_content.index = tool_index
                tool_index += 1
            yield DeltaMessage(content=message_content)

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        messages = self._prompt_stack_to_messages(prompt_stack)

        params = {
            "messages": messages,
            "model": self.model,
            "options": self.options,
            **self.extra_params,
        }

        if prompt_stack.output_schema is not None and self.structured_output_strategy␣
↪== "native":
            params["format"] = prompt_stack.to_output_json_schema()

        # Tool calling is only supported when not streaming
        if prompt_stack.tools and self.use_native_tools:
            params["tools"] = self.__to_ollama_tools(prompt_stack.tools)

        return params

    def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
        ollama_messages = []
        for message in prompt_stack.messages:
            action_result_contents = message.get_content_
↪type(ActionResultMessageContent)

            # Function calls need to be handled separately from the rest of the␣
↪message content
            if action_result_contents:
                ollama_messages.extend(
                    [
                        {
                            "role": self.__to_ollama_role(message, action_result_
```

(continues on next page)

(continued from previous page)

```
↪content),
                            "content": self.__to_ollama_message_content(action_result_
↪content),
                        }
                        for action_result_content in action_result_contents
                    ],
                )

                text_contents = message.get_content_type(TextMessageContent)
                if text_contents:
                    ollama_messages.append({"role": self.__to_ollama_role(message),
↪"content": message.to_text()})
            else:
                ollama_message: dict[str, Any] = {
                    "role": self.__to_ollama_role(message),
                    "content": message.to_text(),
                }

                action_call_contents = message.get_content_
↪type(ActionCallMessageContent)
                if action_call_contents:
                    ollama_message["tool_calls"] = [
                        self.__to_ollama_message_content(action_call_content)
                        for action_call_content in action_call_contents
                    ]

                image_contents = message.get_content_type(ImageMessageContent)
                if image_contents:
                    ollama_message["images"] = [
                        self.__to_ollama_message_content(image_content) for image_
↪content in image_contents
                    ]

                ollama_messages.append(ollama_message)

        return ollama_messages

    def __to_ollama_message_content(self, content: BaseMessageContent) -> str | dict:
        if isinstance(content, TextMessageContent):
            return content.artifact.to_text()
        if isinstance(content, ImageMessageContent):
            if isinstance(content.artifact, ImageArtifact):
                return content.artifact.base64
            # TODO: add support for image urls once resolved https://github.com/
↪ollama/ollama/issues/4474
            raise ValueError(f"Unsupported image artifact type: {type(content.
↪artifact)}")
        if isinstance(content, ActionCallMessageContent):
            action = content.artifact.value

            return {
                "type": "function",
```

(continues on next page)

(continued from previous page)

```
                "id": action.tag,
                "function": {"name": action.to_native_tool_name(), "arguments":␣
↪action.input},
            }
        if isinstance(content, ActionResultMessageContent):
            return content.artifact.to_text()
        raise ValueError(f"Unsupported content type: {type(content)}")

    def __to_ollama_tools(self, tools: list[BaseTool]) -> list[dict]:
        ollama_tools = []

        for tool in tools:
            for activity in tool.activities():
                ollama_tool = {
                    "function": {
                        "name": tool.to_native_tool_name(activity),
                        "description": tool.activity_description(activity),
                    },
                    "type": "function",
                }

                activity_schema = tool.activity_schema(activity)
                if activity_schema is not None:
                    ollama_tool["function"]["parameters"] = tool.to_activity_json_
↪schema(activity, "Parameters Schema")[
                        "properties"
                    ]["values"]

                ollama_tools.append(ollama_tool)
        return ollama_tools

    def __to_ollama_role(self, message: Message, message_content:␣
↪Optional[BaseMessageContent] = None) -> str:
        if message.is_system():
            return "system"
        if message.is_assistant():
            return "assistant"
        if isinstance(message_content, ActionResultMessageContent):
            return "tool"
        return "user"

    def __to_prompt_stack_message_content(self, response: ChatResponse) ->␣
↪list[BaseMessageContent]:
        content = []
        message = response["message"]

        if message.get("content"):
            content.append(TextMessageContent(TextArtifact(response["message"][
↪"content"])))
        if "tool_calls" in message:
            content.extend(
                [
```

(continues on next page)

(continued from previous page)

```
                    ActionCallMessageContent(
                        ActionArtifact(
                            ToolAction(
                                tag=tool_call["function"]["name"],
                                name=ToolAction.from_native_tool_name(tool_call[
↪"function"]["name"])[0],
                                path=ToolAction.from_native_tool_name(tool_call[
↪"function"]["name"])[1],
                                input=tool_call["function"]["arguments"],
                            ),
                        ),
                    )
                    for tool_call in message["tool_calls"]
                ],
            )

        return content

    def __to_prompt_stack_delta_message_content(self, content_delta: ChatResponse) ->␣
↪BaseDeltaMessageContent:
        message = content_delta["message"]
        if message.get("content"):
            return TextDeltaMessageContent(message["content"])
        if "tool_calls" in message and len(message["tool_calls"]):
            tool_calls = message["tool_calls"]

            # Ollama doesnt _really_ support Tool streaming. They provide the full␣
↪tool call at once.
            # Multiple, parallel, Tool calls are provided as multiple content deltas.
            # Tracking here: https://github.com/ollama/ollama/issues/7886
            tool_call = tool_calls[0]

            return ActionCallDeltaMessageContent(
                tag=tool_call["function"]["name"],
                name=ToolAction.from_native_tool_name(tool_call["function"]["name
↪"])[0],
                path=ToolAction.from_native_tool_name(tool_call["function"]["name
↪"])[1],
                partial_input=json.dumps(tool_call["function"]["arguments"]),
            )
        return TextDeltaMessageContent("")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `host = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `options = field(default=Factory(lambda self: {'temperature': self.temperature, 'stop': self.tokenizer.stop_sequences, 'num_predict': self.max_tokens}, takes_self=True), kw_only=True)` class-attribute instance-attribute

- `tokenizer = field(default=Factory(lambda self: SimpleTokenizer(characters_per_token=4, max_input_tokens=2000, max_output_tokens=self.max_tokens), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `use_native_tools = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__to_ollama_message_content(content)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def __to_ollama_message_content(self, content: BaseMessageContent) -> str | dict:
    if isinstance(content, TextMessageContent):
        return content.artifact.to_text()
    if isinstance(content, ImageMessageContent):
        if isinstance(content.artifact, ImageArtifact):
            return content.artifact.base64
        # TODO: add support for image urls once resolved https://github.com/ollama/
↪ollama/issues/4474
        raise ValueError(f"Unsupported image artifact type: {type(content.artifact)}")
    if isinstance(content, ActionCallMessageContent):
        action = content.artifact.value

        return {
            "type": "function",
            "id": action.tag,
            "function": {"name": action.to_native_tool_name(), "arguments": action.
↪input},
        }
    if isinstance(content, ActionResultMessageContent):
        return content.artifact.to_text()
    raise ValueError(f"Unsupported content type: {type(content)}")
```

__to_ollama_role(message, message_content=None)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def __to_ollama_role(self, message: Message, message_content:␣
↪Optional[BaseMessageContent] = None) -> str:
    if message.is_system():
        return "system"
    if message.is_assistant():
        return "assistant"
    if isinstance(message_content, ActionResultMessageContent):
        return "tool"
    return "user"
```

__to_ollama_tools(tools)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def __to_ollama_tools(self, tools: list[BaseTool]) -> list[dict]:
    ollama_tools = []
```

(continues on next page)

(continued from previous page)

```
    for tool in tools:
        for activity in tool.activities():
            ollama_tool = {
                "function": {
                    "name": tool.to_native_tool_name(activity),
                    "description": tool.activity_description(activity),
                },
                "type": "function",
            }

            activity_schema = tool.activity_schema(activity)
            if activity_schema is not None:
                ollama_tool["function"]["parameters"] = tool.to_activity_json_
↪schema(activity, "Parameters Schema")[
                    "properties"
                ]["values"]

            ollama_tools.append(ollama_tool)
    return ollama_tools
```

__to_prompt_stack_delta_message_content(content_delta)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def __to_prompt_stack_delta_message_content(self, content_delta: ChatResponse) ->␣
↪BaseDeltaMessageContent:
    message = content_delta["message"]
    if message.get("content"):
        return TextDeltaMessageContent(message["content"])
    if "tool_calls" in message and len(message["tool_calls"]):
        tool_calls = message["tool_calls"]

        # Ollama doesnt _really_ support Tool streaming. They provide the full tool␣
↪call at once.
        # Multiple, parallel, Tool calls are provided as multiple content deltas.
        # Tracking here: https://github.com/ollama/ollama/issues/7886
        tool_call = tool_calls[0]

        return ActionCallDeltaMessageContent(
            tag=tool_call["function"]["name"],
            name=ToolAction.from_native_tool_name(tool_call["function"]["name"])[0],
            path=ToolAction.from_native_tool_name(tool_call["function"]["name"])[1],
            partial_input=json.dumps(tool_call["function"]["arguments"]),
        )
    return TextDeltaMessageContent("")
```

__to_prompt_stack_message_content(response)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def __to_prompt_stack_message_content(self, response: ChatResponse) ->␣
↪list[BaseMessageContent]:
    content = []
    message = response["message"]

    if message.get("content"):
        content.append(TextMessageContent(TextArtifact(response["message"]["content
↪"])))
    if "tool_calls" in message:
        content.extend(
            [
                ActionCallMessageContent(
                    ActionArtifact(
                        ToolAction(
                            tag=tool_call["function"]["name"],
                            name=ToolAction.from_native_tool_name(tool_call["function
↪"]["name"])[0],
                            path=ToolAction.from_native_tool_name(tool_call["function
↪"]["name"])[1],
                            input=tool_call["function"]["arguments"],
                        ),
                    ),
                )
                for tool_call in message["tool_calls"]
            ],
        )

    return content
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    messages = self._prompt_stack_to_messages(prompt_stack)

    params = {
        "messages": messages,
        "model": self.model,
        "options": self.options,
        **self.extra_params,
    }

    if prompt_stack.output_schema is not None and self.structured_output_strategy ==
↪"native":
        params["format"] = prompt_stack.to_output_json_schema()

    # Tool calling is only supported when not streaming
    if prompt_stack.tools and self.use_native_tools:
```

(continues on next page)

(continued from previous page)

```
        params["tools"] = self.__to_ollama_tools(prompt_stack.tools)

    return params
```

_prompt_stack_to_messages(prompt_stack)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
    ollama_messages = []
    for message in prompt_stack.messages:
        action_result_contents = message.get_content_type(ActionResultMessageContent)

        # Function calls need to be handled separately from the rest of the message␣
↪content
        if action_result_contents:
            ollama_messages.extend(
                [
                    {
                        "role": self.__to_ollama_role(message, action_result_content),
                        "content": self.__to_ollama_message_content(action_result_
↪content),
                    }
                    for action_result_content in action_result_contents
                ],
            )

            text_contents = message.get_content_type(TextMessageContent)
            if text_contents:
                ollama_messages.append({"role": self.__to_ollama_role(message),
↪"content": message.to_text()})
        else:
            ollama_message: dict[str, Any] = {
                "role": self.__to_ollama_role(message),
                "content": message.to_text(),
            }

            action_call_contents = message.get_content_type(ActionCallMessageContent)
            if action_call_contents:
                ollama_message["tool_calls"] = [
                    self.__to_ollama_message_content(action_call_content)
                    for action_call_content in action_call_contents
                ]

            image_contents = message.get_content_type(ImageMessageContent)
            if image_contents:
                ollama_message["images"] = [
                    self.__to_ollama_message_content(image_content) for image_content␣
↪in image_contents
                ]

            ollama_messages.append(ollama_message)
```

(continues on next page)

(continued from previous page)

```
    return ollama_messages
```

client()

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
@lazy_property()
def client(self) -> Client:
    return import_optional_dependency("ollama").Client(host=self.host)
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    params = self._base_params(prompt_stack)
    logger.debug(params)
    response = self.client.chat(**params)
    logger.debug(response.model_dump())

    return Message(
        content=self.__to_prompt_stack_message_content(response),
        role=Message.ASSISTANT_ROLE,
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    params = {**self._base_params(prompt_stack), "stream": True}
    logger.debug(params)
    stream: Iterator = self.client.chat(**params)

    tool_index = 0
    for chunk in stream:
        logger.debug(chunk)
        message_content = self.__to_prompt_stack_delta_message_content(chunk)
        # Ollama provides multiple Tool calls as separate chunks but with no index to␣
↪differentiate them.
        # So we must keep track of the index ourselves.
        if isinstance(message_content, ActionCallDeltaMessageContent):
            message_content.index = tool_index
            tool_index += 1
        yield DeltaMessage(content=message_content)
```

ollama_prompt_driver

- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="ollama_prompt_driver.mdx griptape.drivers.prompt.ollama_prompt_driver.OllamaPromptDriver.model" title="model = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.ollama_prompt_driver.OllamaPromptDriver.model)"><code class="language-python">	*str*	Model name.

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
@define
class OllamaPromptDriver(BasePromptDriver):
    """Ollama Prompt Driver.

    Attributes:
        model: Model name.
    """

    model: str = field(kw_only=True, metadata={"serializable": True})
    host: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    tokenizer: BaseTokenizer = field(
        default=Factory(
            lambda self: SimpleTokenizer(
                characters_per_token=4,
                max_input_tokens=2000,
                max_output_tokens=self.max_tokens,
            ),
            takes_self=True,
        ),
        kw_only=True,
    )
    options: dict = field(
```

(continues on next page)

(continued from previous page)

```
        default=Factory(
            lambda self: {
                "temperature": self.temperature,
                "stop": self.tokenizer.stop_sequences,
                "num_predict": self.max_tokens,
            },
            takes_self=True,
        ),
        kw_only=True,
    )
    use_native_tools: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})
    _client: Optional[Client] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> Client:
        return import_optional_dependency("ollama").Client(host=self.host)

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        params = self._base_params(prompt_stack)
        logger.debug(params)
        response = self.client.chat(**params)
        logger.debug(response.model_dump())

        return Message(
            content=self.__to_prompt_stack_message_content(response),
            role=Message.ASSISTANT_ROLE,
        )

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        params = {**self._base_params(prompt_stack), "stream": True}
        logger.debug(params)
        stream: Iterator = self.client.chat(**params)

        tool_index = 0
        for chunk in stream:
            logger.debug(chunk)
            message_content = self.__to_prompt_stack_delta_message_content(chunk)
            # Ollama provides multiple Tool calls as separate chunks but with no␣
↪index to differentiate them.
            # So we must keep track of the index ourselves.
            if isinstance(message_content, ActionCallDeltaMessageContent):
                message_content.index = tool_index
                tool_index += 1
            yield DeltaMessage(content=message_content)

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        messages = self._prompt_stack_to_messages(prompt_stack)
```

(continues on next page)

(continued from previous page)

```
        params = {
            "messages": messages,
            "model": self.model,
            "options": self.options,
            **self.extra_params,
        }

        if prompt_stack.output_schema is not None and self.structured_output_strategy␣
↪== "native":
            params["format"] = prompt_stack.to_output_json_schema()

        # Tool calling is only supported when not streaming
        if prompt_stack.tools and self.use_native_tools:
            params["tools"] = self.__to_ollama_tools(prompt_stack.tools)

        return params

    def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
        ollama_messages = []
        for message in prompt_stack.messages:
            action_result_contents = message.get_content_
↪type(ActionResultMessageContent)

            # Function calls need to be handled separately from the rest of the␣
↪message content
            if action_result_contents:
                ollama_messages.extend(
                    [
                        {
                            "role": self.__to_ollama_role(message, action_result_
↪content),
                            "content": self.__to_ollama_message_content(action_result_
↪content),
                        }
                        for action_result_content in action_result_contents
                    ],
                )

                text_contents = message.get_content_type(TextMessageContent)
                if text_contents:
                    ollama_messages.append({"role": self.__to_ollama_role(message),
↪"content": message.to_text()})
            else:
                ollama_message: dict[str, Any] = {
                    "role": self.__to_ollama_role(message),
                    "content": message.to_text(),
                }

                action_call_contents = message.get_content_
↪type(ActionCallMessageContent)
                if action_call_contents:
                    ollama_message["tool_calls"] = [
```

(continues on next page)

(continued from previous page)

```
                        self.__to_ollama_message_content(action_call_content)
                        for action_call_content in action_call_contents
                    ]

                image_contents = message.get_content_type(ImageMessageContent)
                if image_contents:
                    ollama_message["images"] = [
                        self.__to_ollama_message_content(image_content) for image_
↪content in image_contents
                    ]

                ollama_messages.append(ollama_message)

        return ollama_messages

    def __to_ollama_message_content(self, content: BaseMessageContent) -> str | dict:
        if isinstance(content, TextMessageContent):
            return content.artifact.to_text()
        if isinstance(content, ImageMessageContent):
            if isinstance(content.artifact, ImageArtifact):
                return content.artifact.base64
            # TODO: add support for image urls once resolved https://github.com/
↪ollama/ollama/issues/4474
            raise ValueError(f"Unsupported image artifact type: {type(content.
↪artifact)}")
        if isinstance(content, ActionCallMessageContent):
            action = content.artifact.value

            return {
                "type": "function",
                "id": action.tag,
                "function": {"name": action.to_native_tool_name(), "arguments":␣
↪action.input},
            }
        if isinstance(content, ActionResultMessageContent):
            return content.artifact.to_text()
        raise ValueError(f"Unsupported content type: {type(content)}")

    def __to_ollama_tools(self, tools: list[BaseTool]) -> list[dict]:
        ollama_tools = []

        for tool in tools:
            for activity in tool.activities():
                ollama_tool = {
                    "function": {
                        "name": tool.to_native_tool_name(activity),
                        "description": tool.activity_description(activity),
                    },
                    "type": "function",
                }

                activity_schema = tool.activity_schema(activity)
```

(continues on next page)

(continued from previous page)

```
                if activity_schema is not None:
                    ollama_tool["function"]["parameters"] = tool.to_activity_json_
↪schema(activity, "Parameters Schema")[
                        "properties"
                    ]["values"]

                ollama_tools.append(ollama_tool)
        return ollama_tools

    def __to_ollama_role(self, message: Message, message_content:␣
↪Optional[BaseMessageContent] = None) -> str:
        if message.is_system():
            return "system"
        if message.is_assistant():
            return "assistant"
        if isinstance(message_content, ActionResultMessageContent):
            return "tool"
        return "user"

    def __to_prompt_stack_message_content(self, response: ChatResponse) ->␣
↪list[BaseMessageContent]:
        content = []
        message = response["message"]

        if message.get("content"):
            content.append(TextMessageContent(TextArtifact(response["message"][
↪"content"])))
        if "tool_calls" in message:
            content.extend(
                [
                    ActionCallMessageContent(
                        ActionArtifact(
                            ToolAction(
                                tag=tool_call["function"]["name"],
                                name=ToolAction.from_native_tool_name(tool_call[
↪"function"]["name"])[0],
                                path=ToolAction.from_native_tool_name(tool_call[
↪"function"]["name"])[1],
                                input=tool_call["function"]["arguments"],
                            ),
                        ),
                    )
                    for tool_call in message["tool_calls"]
                ],
            )

        return content

    def __to_prompt_stack_delta_message_content(self, content_delta: ChatResponse) ->␣
↪BaseDeltaMessageContent:
        message = content_delta["message"]
        if message.get("content"):
```

(continues on next page)

(continued from previous page)

```
            return TextDeltaMessageContent(message["content"])
        if "tool_calls" in message and len(message["tool_calls"]):
            tool_calls = message["tool_calls"]

            # Ollama doesnt _really_ support Tool streaming. They provide the full␣
↪tool call at once.
            # Multiple, parallel, Tool calls are provided as multiple content deltas.
            # Tracking here: https://github.com/ollama/ollama/issues/7886
            tool_call = tool_calls[0]

            return ActionCallDeltaMessageContent(
                tag=tool_call["function"]["name"],
                name=ToolAction.from_native_tool_name(tool_call["function"]["name
↪"])[0],
                path=ToolAction.from_native_tool_name(tool_call["function"]["name
↪"])[1],
                partial_input=json.dumps(tool_call["function"]["arguments"]),
            )
        return TextDeltaMessageContent("")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `host = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `options = field(default=Factory(lambda self: {'temperature': self.temperature, 'stop': self.tokenizer.stop_sequences, 'num_predict': self.max_tokens}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: SimpleTokenizer(characters_per_token=4, max_input_tokens=2000, max_output_tokens=self.max_tokens), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `use_native_tools = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__to_ollama_message_content(content)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def __to_ollama_message_content(self, content: BaseMessageContent) -> str | dict:
    if isinstance(content, TextMessageContent):
        return content.artifact.to_text()
    if isinstance(content, ImageMessageContent):
        if isinstance(content.artifact, ImageArtifact):
            return content.artifact.base64
        # TODO: add support for image urls once resolved https://github.com/ollama/
↪ollama/issues/4474
        raise ValueError(f"Unsupported image artifact type: {type(content.artifact)}")
    if isinstance(content, ActionCallMessageContent):
        action = content.artifact.value
```

(continues on next page)

(continued from previous page)

```
        return {
            "type": "function",
            "id": action.tag,
            "function": {"name": action.to_native_tool_name(), "arguments": action.
↪input},
        }
    if isinstance(content, ActionResultMessageContent):
        return content.artifact.to_text()
    raise ValueError(f"Unsupported content type: {type(content)}")
```

__to_ollama_role(message, message_content=None)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def __to_ollama_role(self, message: Message, message_content:␣
↪Optional[BaseMessageContent] = None) -> str:
    if message.is_system():
        return "system"
    if message.is_assistant():
        return "assistant"
    if isinstance(message_content, ActionResultMessageContent):
        return "tool"
    return "user"
```

__to_ollama_tools(tools)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def __to_ollama_tools(self, tools: list[BaseTool]) -> list[dict]:
    ollama_tools = []

    for tool in tools:
        for activity in tool.activities():
            ollama_tool = {
                "function": {
                    "name": tool.to_native_tool_name(activity),
                    "description": tool.activity_description(activity),
                },
                "type": "function",
            }

            activity_schema = tool.activity_schema(activity)
            if activity_schema is not None:
                ollama_tool["function"]["parameters"] = tool.to_activity_json_
↪schema(activity, "Parameters Schema")[
                    "properties"
                ]["values"]

            ollama_tools.append(ollama_tool)
    return ollama_tools
```

__to_prompt_stack_delta_message_content(content_delta)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def __to_prompt_stack_delta_message_content(self, content_delta: ChatResponse) ->␣
↪BaseDeltaMessageContent:
    message = content_delta["message"]
    if message.get("content"):
        return TextDeltaMessageContent(message["content"])
    if "tool_calls" in message and len(message["tool_calls"]):
        tool_calls = message["tool_calls"]

        # Ollama doesnt _really_ support Tool streaming. They provide the full tool␣
↪call at once.
        # Multiple, parallel, Tool calls are provided as multiple content deltas.
        # Tracking here: https://github.com/ollama/ollama/issues/7886
        tool_call = tool_calls[0]

        return ActionCallDeltaMessageContent(
            tag=tool_call["function"]["name"],
            name=ToolAction.from_native_tool_name(tool_call["function"]["name"])[0],
            path=ToolAction.from_native_tool_name(tool_call["function"]["name"])[1],
            partial_input=json.dumps(tool_call["function"]["arguments"]),
        )
    return TextDeltaMessageContent("")
```

__to_prompt_stack_message_content(response)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def __to_prompt_stack_message_content(self, response: ChatResponse) ->␣
↪list[BaseMessageContent]:
    content = []
    message = response["message"]

    if message.get("content"):
        content.append(TextMessageContent(TextArtifact(response["message"]["content
↪"])))
    if "tool_calls" in message:
        content.extend(
            [
                ActionCallMessageContent(
                    ActionArtifact(
                        ToolAction(
                            tag=tool_call["function"]["name"],
                            name=ToolAction.from_native_tool_name(tool_call["function
↪"]["name"])[0],
                            path=ToolAction.from_native_tool_name(tool_call["function
↪"]["name"])[1],
                            input=tool_call["function"]["arguments"],
                        ),
                    ),
                )
```

(continues on next page)

(continued from previous page)

```
                for tool_call in message["tool_calls"]
            ],
        )

    return content
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    messages = self._prompt_stack_to_messages(prompt_stack)

    params = {
        "messages": messages,
        "model": self.model,
        "options": self.options,
        **self.extra_params,
    }

    if prompt_stack.output_schema is not None and self.structured_output_strategy ==
↪"native":
        params["format"] = prompt_stack.to_output_json_schema()

    # Tool calling is only supported when not streaming
    if prompt_stack.tools and self.use_native_tools:
        params["tools"] = self.__to_ollama_tools(prompt_stack.tools)

    return params
```

_prompt_stack_to_messages(prompt_stack)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
def _prompt_stack_to_messages(self, prompt_stack: PromptStack) -> list[dict]:
    ollama_messages = []
    for message in prompt_stack.messages:
        action_result_contents = message.get_content_type(ActionResultMessageContent)

        # Function calls need to be handled separately from the rest of the message␣
↪content
        if action_result_contents:
            ollama_messages.extend(
                [
                    {
                        "role": self.__to_ollama_role(message, action_result_content),
                        "content": self.__to_ollama_message_content(action_result_
↪content),
                    }
                    for action_result_content in action_result_contents
                ],
```

(continues on next page)

(continued from previous page)

```
            )

            text_contents = message.get_content_type(TextMessageContent)
            if text_contents:
                ollama_messages.append({"role": self.__to_ollama_role(message),
↪"content": message.to_text()})
        else:
            ollama_message: dict[str, Any] = {
                "role": self.__to_ollama_role(message),
                "content": message.to_text(),
            }

            action_call_contents = message.get_content_type(ActionCallMessageContent)
            if action_call_contents:
                ollama_message["tool_calls"] = [
                    self.__to_ollama_message_content(action_call_content)
                    for action_call_content in action_call_contents
                ]

            image_contents = message.get_content_type(ImageMessageContent)
            if image_contents:
                ollama_message["images"] = [
                    self.__to_ollama_message_content(image_content) for image_content␣
↪in image_contents
                ]

            ollama_messages.append(ollama_message)

    return ollama_messages
```

client()

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
@lazy_property()
def client(self) -> Client:
    return import_optional_dependency("ollama").Client(host=self.host)
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    params = self._base_params(prompt_stack)
    logger.debug(params)
    response = self.client.chat(**params)
    logger.debug(response.model_dump())

    return Message(
        content=self.__to_prompt_stack_message_content(response),
```

(continues on next page)

(continued from previous page)

```
        role=Message.ASSISTANT_ROLE,
    )
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/ollama_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    params = {**self._base_params(prompt_stack), "stream": True}
    logger.debug(params)
    stream: Iterator = self.client.chat(**params)

    tool_index = 0
    for chunk in stream:
        logger.debug(chunk)
        message_content = self.__to_prompt_stack_delta_message_content(chunk)
        # Ollama provides multiple Tool calls as separate chunks but with no index to␣
↪differentiate them.
        # So we must keep track of the index ourselves.
        if isinstance(message_content, ActionCallDeltaMessageContent):
            message_content.index = tool_index
            tool_index += 1
        yield DeltaMessage(content=message_content)
```

openai

- `__all__ = ['AzureOpenAiChatPromptDriver', 'OpenAiChatPromptDriver']` module-attribute

Bases: *AzureOpenAiChatPromptDriver* “)

Attributes

Name	Type	Description
<a href="openai.mdx#griptape.drivers.prompt.openai.AzureOpenAiChatPromptDriver.azure_deployment" title="azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.openai.AzureOpenAiChatPromptDriver.azure_deployment)"><code class="language-python">	*str*	An optional Azure OpenAi deployment id. Defaults to the model name.
<a href="openai.mdx#griptape.drivers.prompt.openai.AzureOpenAiChatPromptDriver.azure_endpoint" title="azure_endpoint = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.openai.AzureOpenAiChatPromptDriver.azure_endpoint)"><code class="language-python">	*str*	An Azure OpenAi endpoint.

continues on next page

Table 207 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.prompt.openai.AzureOpenAiChatPromptDriver.azure_ad_token" title="azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.prompt.openai.AzureOpenAiChatPromptDriver.azure_ad_token)"><code class="language-python">	*Optional[str]*	An optional Azure Active Directory token.
<a href="openai.mdx#griptape.drivers.prompt.openai.AzureOpenAiChatPromptDriver.azure_ad_token_provider" title="azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.prompt.openai.AzureOpenAiChatPromptDriver.azure_ad_token_provider)"><code class="language-python">	*Optional[Callable[[], str]]*	An optional Azure Active Directory token provider.

continues on next page

Table 207 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.prompt.openai.AzureOpenAiChatPromptDriver.api_version" title="api_version = field(default='2024-10-21', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.openai.AzureOpenAiChatPromptDriver.api_version)"><code class="language-python">	*str*	An Azure OpenAi API version.
<a href="openai.mdx#griptape.drivers.prompt.openai.AzureOpenAiChatPromptDriver.client" title="client() (griptape.drivers.prompt.openai.AzureOpenAiChatPromptDriver.client)"><code class="language-python">	*AzureOpenAI*	An *openai.AzureOpenAI* client.

Source Code in griptape/drivers/prompt/azure_openai_chat_prompt_driver.py

```
@define
class AzureOpenAiChatPromptDriver(OpenAiChatPromptDriver):
    """Azure OpenAi Chat Prompt Driver.

    Attributes:
        azure_deployment: An optional Azure OpenAi deployment id. Defaults to the␣
↪model name.
        azure_endpoint: An Azure OpenAi endpoint.
        azure_ad_token: An optional Azure Active Directory token.
        azure_ad_token_provider: An optional Azure Active Directory token provider.
        api_version: An Azure OpenAi API version.
        client: An `openai.AzureOpenAI` client.
    """

    azure_deployment: str = field(
        kw_only=True,
        default=Factory(lambda self: self.model, takes_self=True),
        metadata={"serializable": True},
    )
```

(continues on next page)

(continued from previous page)

```
    azure_endpoint: str = field(kw_only=True, metadata={"serializable": True})
    azure_ad_token: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": False})
    azure_ad_token_provider: Optional[Callable[[], str]] = field(
        kw_only=True,
        default=None,
        metadata={"serializable": False},
    )
    api_version: str = field(default="2024-10-21", kw_only=True, metadata={
↪"serializable": True})
    _client: Optional[openai.AzureOpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.AzureOpenAI:
        return openai.AzureOpenAI(
            organization=self.organization,
            api_key=self.api_key,
            api_version=self.api_version,
            azure_endpoint=self.azure_endpoint,
            azure_deployment=self.azure_deployment,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        params = super()._base_params(prompt_stack)
        if self.api_version < "2024-02-01" and "seed" in params:
            del params["seed"]
        if self.api_version < "2024-10-21":
            if "stream_options" in params:
                del params["stream_options"]
            if "parallel_tool_calls" in params:
                del params["parallel_tool_calls"]

        # TODO: Add once Azure supports modalities
        if "modalities" in params:
            del params["modalities"]
        return params
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_version = field(default='2024-10-21', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True})` class-attribute instance-attribute

- `azure_endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/azure_openai_chat_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    params = super()._base_params(prompt_stack)
    if self.api_version < "2024-02-01" and "seed" in params:
        del params["seed"]
    if self.api_version < "2024-10-21":
        if "stream_options" in params:
            del params["stream_options"]
        if "parallel_tool_calls" in params:
            del params["parallel_tool_calls"]

    # TODO: Add once Azure supports modalities
    if "modalities" in params:
        del params["modalities"]
    return params
```

client()

Source Code in griptape/drivers/prompt/azure_openai_chat_prompt_driver.py

```
@lazy_property()
def client(self) -> openai.AzureOpenAI:
    return openai.AzureOpenAI(
        organization=self.organization,
        api_key=self.api_key,
        api_version=self.api_version,
        azure_endpoint=self.azure_endpoint,
        azure_deployment=self.azure_deployment,
        azure_ad_token=self.azure_ad_token,
        azure_ad_token_provider=self.azure_ad_token_provider,
    )
```

OpenAiChatPromptDriver

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="openai.mdx#griptape.drivers.prompt.openai.OpenAiChatPromptDriver.base_url" title="base_url = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.openai.OpenAiChatPromptDriver.base_url)"><code class="language-python">	*Optional[str]*	An optional OpenAi API URL.
<code class="language-python">	*Optional[str]*	An optional OpenAi API key. If not provided, the *OPENAI_API_KEY* environment variable will be used.
<code class="language-python">	*Optional[str]*	An optional OpenAI organization. If not provided, the *OPENAI_ORG_ID* environment variable will be used.

continues on next page

Table 208 – continued from previous page

Name	Type	Description
<code class="language-python">	*OpenAI*	An *openai.OpenAI* client.
<code class="language-python">	*str*	An OpenAI model name.
<code class="language-python">	<code class="language-python">BaseTokenizer</code>	An *OpenAiTokenizer*.

continues on next page

Table 208 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.prompt.openai.OpenAiChatPromptDriver.user" title="user = field(default='', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.openai.OpenAiChatPromptDriver.user)"><code class="language-python">	*str*	A user id. Can be used to track requests by user.
<a href="openai.mdx#griptape.drivers.prompt.openai.OpenAiChatPromptDriver.response_format" title="response_format = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.openai.OpenAiChatPromptDriver.response_format)"><code class="language-python">	*Optional[dict]*	An optional OpenAi Chat Completion response format. Currently only supports *json_object* which will enable OpenAi's JSON mode.
<a href="openai.mdx#griptape.drivers.prompt.openai.OpenAiChatPromptDriver.seed" title="seed = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.openai.OpenAiChatPromptDriver.seed)"><code class="language-python">	*Optional[int]*	An optional OpenAi Chat Completion seed.

continues on next page

Table 208 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.prompt.openai.OpenAiChatPromptDriver.ignored_exception_types" title="ignored_exception_types = field(default=Factory(lambda: (openai.BadRequestError, openai.AuthenticationError, openai.PermissionDeniedError, openai.NotFoundError, openai.ConflictError, openai.UnprocessableEntityError)), kw_only=True) class-attribute instance-attribute (griptape.drivers.prompt.openai.OpenAiChatPromptDriver.ignored_exception_types)"><code class="language-python">	*tuple[type[Exception], ...]*	An optional tuple of exception types to ignore. Defaults to OpenAI's known exception types.
<a href="openai.mdx#griptape.drivers.prompt.openai.OpenAiChatPromptDriver.parallel_tool_calls" title="parallel_tool_calls = field(default=True, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.openai.OpenAiChatPromptDriver.parallel_tool_calls)"><code class="language-python">	*bool*	A flag to enable parallel tool calls. Defaults to *True*.

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
@define
class OpenAiChatPromptDriver(BasePromptDriver):
    """OpenAI Chat Prompt Driver.
```

(continues on next page)

(continued from previous page)

```
    Attributes:
        base_url: An optional OpenAi API URL.
        api_key: An optional OpenAi API key. If not provided, the `OPENAI_API_KEY`␣
↪environment variable will be used.
        organization: An optional OpenAI organization. If not provided, the `OPENAI_
↪ORG_ID` environment variable will be used.
        client: An `openai.OpenAI` client.
        model: An OpenAI model name.
        tokenizer: An `OpenAiTokenizer`.
        user: A user id. Can be used to track requests by user.
        response_format: An optional OpenAi Chat Completion response format.␣
↪Currently only supports `json_object` which will enable OpenAis JSON mode.
        seed: An optional OpenAi Chat Completion seed.
        ignored_exception_types: An optional tuple of exception types to ignore.␣
↪Defaults to OpenAIs known exception types.
        parallel_tool_calls: A flag to enable parallel tool calls. Defaults to `True`.
    """

    base_url: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    organization: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    model: str = field(kw_only=True, metadata={"serializable": True})
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
    user: str = field(default="", kw_only=True, metadata={"serializable": True})
    response_format: Optional[dict] = field(
        default=None,
        kw_only=True,
        metadata={"serializable": True},
    )
    seed: Optional[int] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    tool_choice: str = field(default="auto", kw_only=True, metadata={"serializable":␣
↪False})
    reasoning_effort: Literal["low", "medium", "high"] = field(
        default="medium", kw_only=True, metadata={"serializable": True}
    )
    use_native_tools: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})
    structured_output_strategy: StructuredOutputStrategy = field(
        default="native", kw_only=True, metadata={"serializable": True}
    )
    parallel_tool_calls: bool = field(default=True, kw_only=True, metadata={
↪"serializable": True})
    ignored_exception_types: tuple[type[Exception], ...] = field(
```

(continues on next page)

(continued from previous page)

```
        default=Factory(
            lambda: (
                openai.BadRequestError,
                openai.AuthenticationError,
                openai.PermissionDeniedError,
                openai.NotFoundError,
                openai.ConflictError,
                openai.UnprocessableEntityError,
            ),
        ),
        kw_only=True,
    )
    modalities: list[str] = field(factory=list, kw_only=True, metadata={"serializable
↪": True})
    audio: dict = field(
        default=Factory(lambda: {"voice": "alloy", "format": "pcm16"}), kw_only=True,␣
↪metadata={"serializable": True}
    )
    _client: Optional[openai.OpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.OpenAI:
        return openai.OpenAI(
            base_url=self.base_url,
            api_key=self.api_key,
            organization=self.organization,
        )

    @property
    def is_reasoning_model(self) -> bool:
        return self.model.startswith("o")

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        params = self._base_params(prompt_stack)
        logger.debug(params)
        result = self.client.chat.completions.create(**params)

        logger.debug(result.model_dump())
        return self._to_message(result)

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        params = self._base_params(prompt_stack)
        logger.debug({"stream": True, **params})
        result = self.client.chat.completions.create(**params, stream=True)

        return self._to_delta_message_stream(result)

    def _to_message(self, result: ChatCompletion) -> Message:
```

(continues on next page)

(continued from previous page)

```
        if len(result.choices) == 1:
            choice_message = result.choices[0].message

            message = Message(
                content=self.__to_prompt_stack_message_content(choice_message),
                role=Message.ASSISTANT_ROLE,
            )
            if result.usage is not None:
                message.usage = Message.Usage(
                    input_tokens=result.usage.prompt_tokens,
                    output_tokens=result.usage.completion_tokens,
                )

            return message
        raise Exception("Completion with more than one choice is not supported yet.")

    def _to_delta_message_stream(self, result: Stream[ChatCompletionChunk]) ->␣
↪Iterator[DeltaMessage]:
        for message in result:
            if message.usage is not None:
                yield DeltaMessage(
                    usage=DeltaMessage.Usage(
                        input_tokens=message.usage.prompt_tokens,
                        output_tokens=message.usage.completion_tokens,
                    ),
                )
            if message.choices:
                choice = message.choices[0]
                delta = choice.delta

                content = self.__to_prompt_stack_delta_message_content(delta)

                if content is not None:
                    yield DeltaMessage(content=content)

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        params = {
            "model": self.model,
            **({"user": self.user} if self.user else {}),
            **({"seed": self.seed} if self.seed is not None else {}),
            **({"modalities": self.modalities} if self.modalities and not self.is_
↪reasoning_model else {}),
            **(
                {"reasoning_effort": self.reasoning_effort}
                if self.is_reasoning_model and self.model != "o1-mini"
                else {}
            ),
            **({"temperature": self.temperature} if not self.is_reasoning_model else
↪{}),
            **({"audio": self.audio} if "audio" in self.modalities else {}),
            **(
                {"stop": self.tokenizer.stop_sequences}
```

(continues on next page)

(continued from previous page)

```
                if not self.is_reasoning_model and self.tokenizer.stop_sequences
                else {}
            ),
            **({"max_tokens": self.max_tokens} if self.max_tokens is not None else {}
↪),
            **({"stream_options": {"include_usage": True}} if self.stream else {}),
            **self.extra_params,
        }

        if prompt_stack.tools and self.use_native_tools:
            params["tool_choice"] = self.tool_choice
            params["parallel_tool_calls"] = self.parallel_tool_calls

        if prompt_stack.output_schema is not None:
            if self.structured_output_strategy == "native":
                params["response_format"] = {
                    "type": "json_schema",
                    "json_schema": {
                        "name": "Output",
                        "schema": prompt_stack.to_output_json_schema(),
                        "strict": True,
                    },
                }
            elif self.structured_output_strategy == "tool" and self.use_native_tools:
                params["tool_choice"] = "required"

        if self.response_format is not None:
            if self.response_format == {"type": "json_object"}:
                params["response_format"] = self.response_format
                # JSON mode still requires a system message instructing the LLM to␣
↪output JSON.
                prompt_stack.add_system_message("Provide your response as a valid␣
↪JSON object.")
            else:
                params["response_format"] = self.response_format

        if prompt_stack.tools and self.use_native_tools:
            params["tools"] = self.__to_openai_tools(prompt_stack.tools)

        messages = self.__to_openai_messages(prompt_stack.messages)

        params["messages"] = messages

        return params

    def __to_openai_messages(self, messages: list[Message]) -> list[dict]:
        openai_messages = []

        for message in messages:
            # If the message only contains textual content we can send it as a single␣
↪content.
            if message.has_all_content_type(TextMessageContent):
```

(continues on next page)

(continued from previous page)

```
                openai_messages.append({"role": self.__to_openai_role(message),
↪"content": message.to_text()})
            # Action results must be sent as separate messages.
            elif action_result_contents := message.get_content_
↪type(ActionResultMessageContent):
                openai_messages.extend(
                    {
                        "role": self.__to_openai_role(message, action_result_content),
                        "content": self.__to_openai_message_content(action_result_
↪content),
                        "tool_call_id": action_result_content.action.tag,
                    }
                    for action_result_content in action_result_contents
                )

                if message.has_any_content_type(TextMessageContent):
                    openai_messages.append({"role": self.__to_openai_role(message),
↪"content": message.to_text()})
            else:
                openai_message = {
                    "role": self.__to_openai_role(message),
                    "content": [],
                }

                for content in message.content:
                    if isinstance(content, ActionCallMessageContent):
                        if "tool_calls" not in openai_message:
                            openai_message["tool_calls"] = []
                        openai_message["tool_calls"].append(self.__to_openai_message_
↪content(content))
                    elif (
                        isinstance(content, AudioMessageContent)
                        and message.is_assistant()
                        and time.time() < content.artifact.meta["expires_at"]
                    ):
                        # For assistant audio messages, we reference the audio id␣
↪instead of sending audio message content.
                        openai_message["audio"] = {
                            "id": content.artifact.meta["audio_id"],
                        }
                    else:
                        openai_message["content"].append(self.__to_openai_message_
↪content(content))

                # Some OpenAi-compatible services dont accept an empty array for␣
↪content
                if not openai_message["content"]:
                    del openai_message["content"]

                openai_messages.append(openai_message)

        return openai_messages
```

(continues on next page)

(continued from previous page)

```
    def __to_openai_role(self, message: Message, message_content:␣
↪Optional[BaseMessageContent] = None) -> str:
        if message.is_system():
            if self.is_reasoning_model:
                return "developer"
            return "system"
        if message.is_assistant():
            return "assistant"
        if isinstance(message_content, ActionResultMessageContent):
            return "tool"
        return "user"

    def __to_openai_tools(self, tools: list[BaseTool]) -> list[dict]:
        return [
            {
                "function": {
                    "name": tool.to_native_tool_name(activity),
                    "description": tool.activity_description(activity),
                    "parameters": tool.to_activity_json_schema(activity, "Parameters␣
↪Schema"),
                },
                "type": "function",
            }
            for tool in tools
            for activity in tool.activities()
        ]

    def __to_openai_message_content(self, content: BaseMessageContent) -> str | dict:
        if isinstance(content, TextMessageContent):
            return {"type": "text", "text": content.artifact.to_text()}
        if isinstance(content, ImageMessageContent):
            if isinstance(content.artifact, ImageArtifact):
                return {
                    "type": "image_url",
                    "image_url": {"url": f"data:{content.artifact.mime_type}; base64,
↪{content.artifact.base64}"},
                }
            if isinstance(content.artifact, ImageUrlArtifact):
                return {
                    "type": "image_url",
                    "image_url": {"url": content.artifact.value},
                }
            raise ValueError(f"Unsupported image artifact type: {type(content.
↪artifact)}")
        if isinstance(content, AudioMessageContent):
            artifact = content.artifact
            metadata = artifact.meta

            # If theres an expiration date, we can assume its an assistant message.
            if "expires_at" in metadata:
                # If its expired, we send the transcript instead.
```

(continues on next page)

(continued from previous page)

```
                if time.time() >= metadata["expires_at"]:
                    return {
                        "type": "text",
                        "text": artifact.meta.get("transcript"),
                    }
                # This should never occur, since a non-expired audio content
                # should have already been referenced by the audio id.
                raise ValueError("Assistant audio messages should be sent as audio␣
↪ids.")
            # If theres no expiration date, we can assume its a user message where we␣
↪send the audio every time.
            return {
                "type": "input_audio",
                "input_audio": {
                    "data": base64.b64encode(artifact.value).decode("utf-8"),
                    "format": artifact.format,
                },
            }
        if isinstance(content, ActionCallMessageContent):
            action = content.artifact.value

            return {
                "type": "function",
                "id": action.tag,
                "function": {"name": action.to_native_tool_name(), "arguments": json.
↪dumps(action.input)},
            }
        if isinstance(content, ActionResultMessageContent):
            return content.artifact.to_text()
        return {"type": "text", "text": content.artifact.to_text()}

    def __to_prompt_stack_message_content(self, response: ChatCompletionMessage) ->␣
↪list[BaseMessageContent]:
        content = []

        if response.content is not None:
            content.append(TextMessageContent(TextArtifact(response.content)))
        if hasattr(response, "audio") and response.audio is not None:
            content.append(
                AudioMessageContent(
                    AudioArtifact(
                        value=base64.b64decode(response.audio.data),
                        format="wav",
                        meta={
                            "audio_id": response.audio.id,
                            "transcript": response.audio.transcript,
                            "expires_at": response.audio.expires_at,
                        },
                    )
                )
            )
        if response.tool_calls is not None:
```

(continues on next page)

(continued from previous page)

```
            content.extend(
                [
                    ActionCallMessageContent(
                        ActionArtifact(
                            ToolAction(
                                tag=tool_call.id,
                                name=ToolAction.from_native_tool_name(tool_call.
↪function.name)[0],
                                path=ToolAction.from_native_tool_name(tool_call.
↪function.name)[1],
                                input=json.loads(tool_call.function.arguments),
                            ),
                        ),
                    )
                    for tool_call in response.tool_calls
                ],
            )

        return content

    def __to_prompt_stack_delta_message_content(self, content_delta: ChoiceDelta) ->␣
↪Optional[BaseDeltaMessageContent]:
        if content_delta.content is not None:
            return TextDeltaMessageContent(content_delta.content)
        if content_delta.tool_calls is not None:
            tool_calls = content_delta.tool_calls

            if len(tool_calls) == 1:
                tool_call = tool_calls[0]
                index = tool_call.index

                if tool_call.function is not None:
                    function_name = tool_call.function.name
                    return ActionCallDeltaMessageContent(
                        index=index,
                        tag=tool_call.id,
                        name=ToolAction.from_native_tool_name(function_name)[0] if␣
↪function_name else None,
                        path=ToolAction.from_native_tool_name(function_name)[1] if␣
↪function_name else None,
                        partial_input=tool_call.function.arguments,
                    )
                raise ValueError(f"Unsupported tool call delta: {tool_call}")
            raise ValueError(f"Unsupported tool call delta length: {len(tool_calls)}")
        # OpenAi doesnt have types for audio deltas so we need to use hasattr and␣
↪getattr.
        if hasattr(content_delta, "audio") and getattr(content_delta, "audio") is not␣
↪None:
            audio_chunk: dict = getattr(content_delta, "audio")
            return AudioDeltaMessageContent(
                id=audio_chunk.get("id"),
                data=audio_chunk.get("data"),
```

(continues on next page)

(continued from previous page)

```
                expires_at=audio_chunk.get("expires_at"),
                transcript=audio_chunk.get("transcript"),
            )
        return None
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `audio = field(default=Factory(lambda: {'voice': 'alloy', 'format': 'pcm16'}), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `base_url = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `ignored_exception_types = field(default=Factory(lambda: (openai.BadRequestError, openai.AuthenticationError, openai.PermissionDeniedError, openai.NotFoundError, openai.ConflictError, openai.UnprocessableEntityError)), kw_only=True)` class-attribute instance-attribute
- `is_reasoning_model` property
- `modalities = field(factory=list, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `organization = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `parallel_tool_calls = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `reasoning_effort = field(default='medium', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `response_format = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `seed = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='native', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `tool_choice = field(default='auto', kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `use_native_tools = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `user = field(default='', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__to_openai_message_content(content)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def __to_openai_message_content(self, content: BaseMessageContent) -> str | dict:
    if isinstance(content, TextMessageContent):
        return {"type": "text", "text": content.artifact.to_text()}
    if isinstance(content, ImageMessageContent):
        if isinstance(content.artifact, ImageArtifact):
            return {
                "type": "image_url",
                "image_url": {"url": f"data:{content.artifact.mime_type}; base64,
↪{content.artifact.base64}"},
            }
        if isinstance(content.artifact, ImageUrlArtifact):
            return {
                "type": "image_url",
                "image_url": {"url": content.artifact.value},
            }
        raise ValueError(f"Unsupported image artifact type: {type(content.artifact)}")
    if isinstance(content, AudioMessageContent):
        artifact = content.artifact
        metadata = artifact.meta

        # If theres an expiration date, we can assume its an assistant message.
        if "expires_at" in metadata:
            # If its expired, we send the transcript instead.
            if time.time() >= metadata["expires_at"]:
                return {
                    "type": "text",
                    "text": artifact.meta.get("transcript"),
                }
            # This should never occur, since a non-expired audio content
            # should have already been referenced by the audio id.
            raise ValueError("Assistant audio messages should be sent as audio ids.")
        # If theres no expiration date, we can assume its a user message where we␣
↪send the audio every time.
        return {
            "type": "input_audio",
            "input_audio": {
                "data": base64.b64encode(artifact.value).decode("utf-8"),
                "format": artifact.format,
            },
        }
    if isinstance(content, ActionCallMessageContent):
        action = content.artifact.value

        return {
            "type": "function",
            "id": action.tag,
            "function": {"name": action.to_native_tool_name(), "arguments": json.
↪dumps(action.input)},
        }
    if isinstance(content, ActionResultMessageContent):
```

(continues on next page)

(continued from previous page)

```
        return content.artifact.to_text()
    return {"type": "text", "text": content.artifact.to_text()}
```

__to_openai_messages(messages)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def __to_openai_messages(self, messages: list[Message]) -> list[dict]:
    openai_messages = []

    for message in messages:
        # If the message only contains textual content we can send it as a single␣
↪content.
        if message.has_all_content_type(TextMessageContent):
            openai_messages.append({"role": self.__to_openai_role(message), "content
↪": message.to_text()})
        # Action results must be sent as separate messages.
        elif action_result_contents := message.get_content_
↪type(ActionResultMessageContent):
            openai_messages.extend(
                {
                    "role": self.__to_openai_role(message, action_result_content),
                    "content": self.__to_openai_message_content(action_result_
↪content),
                    "tool_call_id": action_result_content.action.tag,
                }
                for action_result_content in action_result_contents
            )

            if message.has_any_content_type(TextMessageContent):
                openai_messages.append({"role": self.__to_openai_role(message),
↪"content": message.to_text()})
        else:
            openai_message = {
                "role": self.__to_openai_role(message),
                "content": [],
            }

            for content in message.content:
                if isinstance(content, ActionCallMessageContent):
                    if "tool_calls" not in openai_message:
                        openai_message["tool_calls"] = []
                    openai_message["tool_calls"].append(self.__to_openai_message_
↪content(content))
                elif (
                    isinstance(content, AudioMessageContent)
                    and message.is_assistant()
                    and time.time() < content.artifact.meta["expires_at"]
                ):
                    # For assistant audio messages, we reference the audio id instead␣
↪of sending audio message content.
                    openai_message["audio"] = {
```

(continues on next page)

(continued from previous page)

```
                        "id": content.artifact.meta["audio_id"],
                    }
                else:
                    openai_message["content"].append(self.__to_openai_message_
↪content(content))

            # Some OpenAi-compatible services dont accept an empty array for content
            if not openai_message["content"]:
                del openai_message["content"]

            openai_messages.append(openai_message)

    return openai_messages
```

__to_openai_role(message, message_content=None)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def __to_openai_role(self, message: Message, message_content:␣
↪Optional[BaseMessageContent] = None) -> str:
    if message.is_system():
        if self.is_reasoning_model:
            return "developer"
        return "system"
    if message.is_assistant():
        return "assistant"
    if isinstance(message_content, ActionResultMessageContent):
        return "tool"
    return "user"
```

__to_openai_tools(tools)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def __to_openai_tools(self, tools: list[BaseTool]) -> list[dict]:
    return [
        {
            "function": {
                "name": tool.to_native_tool_name(activity),
                "description": tool.activity_description(activity),
                "parameters": tool.to_activity_json_schema(activity, "Parameters␣
↪Schema"),
            },
            "type": "function",
        }
        for tool in tools
        for activity in tool.activities()
    ]
```

__to_prompt_stack_delta_message_content(content_delta)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def __to_prompt_stack_delta_message_content(self, content_delta: ChoiceDelta) ->␣
↪Optional[BaseDeltaMessageContent]:
    if content_delta.content is not None:
        return TextDeltaMessageContent(content_delta.content)
    if content_delta.tool_calls is not None:
        tool_calls = content_delta.tool_calls

        if len(tool_calls) == 1:
            tool_call = tool_calls[0]
            index = tool_call.index

            if tool_call.function is not None:
                function_name = tool_call.function.name
                return ActionCallDeltaMessageContent(
                    index=index,
                    tag=tool_call.id,
                    name=ToolAction.from_native_tool_name(function_name)[0] if␣
↪function_name else None,
                    path=ToolAction.from_native_tool_name(function_name)[1] if␣
↪function_name else None,
                    partial_input=tool_call.function.arguments,
                )
            raise ValueError(f"Unsupported tool call delta: {tool_call}")
        raise ValueError(f"Unsupported tool call delta length: {len(tool_calls)}")
    # OpenAi doesnt have types for audio deltas so we need to use hasattr and getattr.
    if hasattr(content_delta, "audio") and getattr(content_delta, "audio") is not␣
↪None:
        audio_chunk: dict = getattr(content_delta, "audio")
        return AudioDeltaMessageContent(
            id=audio_chunk.get("id"),
            data=audio_chunk.get("data"),
            expires_at=audio_chunk.get("expires_at"),
            transcript=audio_chunk.get("transcript"),
        )
    return None
```

__to_prompt_stack_message_content(response)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def __to_prompt_stack_message_content(self, response: ChatCompletionMessage) ->␣
↪list[BaseMessageContent]:
    content = []

    if response.content is not None:
        content.append(TextMessageContent(TextArtifact(response.content)))
    if hasattr(response, "audio") and response.audio is not None:
        content.append(
            AudioMessageContent(
```

(continues on next page)

(continued from previous page)

```
                AudioArtifact(
                    value=base64.b64decode(response.audio.data),
                    format="wav",
                    meta={
                        "audio_id": response.audio.id,
                        "transcript": response.audio.transcript,
                        "expires_at": response.audio.expires_at,
                    },
                )
            )
        )
    if response.tool_calls is not None:
        content.extend(
            [
                ActionCallMessageContent(
                    ActionArtifact(
                        ToolAction(
                            tag=tool_call.id,
                            name=ToolAction.from_native_tool_name(tool_call.function.
↪name)[0],
                            path=ToolAction.from_native_tool_name(tool_call.function.
↪name)[1],
                            input=json.loads(tool_call.function.arguments),
                        ),
                    ),
                )
                for tool_call in response.tool_calls
            ],
        )

    return content
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    params = {
        "model": self.model,
        **({"user": self.user} if self.user else {}),
        **({"seed": self.seed} if self.seed is not None else {}),
        **({"modalities": self.modalities} if self.modalities and not self.is_
↪reasoning_model else {}),
        **(
            {"reasoning_effort": self.reasoning_effort}
            if self.is_reasoning_model and self.model != "o1-mini"
            else {}
        ),
        **({"temperature": self.temperature} if not self.is_reasoning_model else {}),
        **({"audio": self.audio} if "audio" in self.modalities else {}),
        **(
            {"stop": self.tokenizer.stop_sequences}
```

(continues on next page)

(continued from previous page)

```
            if not self.is_reasoning_model and self.tokenizer.stop_sequences
            else {}
        ),
        **({"max_tokens": self.max_tokens} if self.max_tokens is not None else {}),
        **({"stream_options": {"include_usage": True}} if self.stream else {}),
        **self.extra_params,
    }

    if prompt_stack.tools and self.use_native_tools:
        params["tool_choice"] = self.tool_choice
        params["parallel_tool_calls"] = self.parallel_tool_calls

    if prompt_stack.output_schema is not None:
        if self.structured_output_strategy == "native":
            params["response_format"] = {
                "type": "json_schema",
                "json_schema": {
                    "name": "Output",
                    "schema": prompt_stack.to_output_json_schema(),
                    "strict": True,
                },
            }
        elif self.structured_output_strategy == "tool" and self.use_native_tools:
            params["tool_choice"] = "required"

    if self.response_format is not None:
        if self.response_format == {"type": "json_object"}:
            params["response_format"] = self.response_format
            # JSON mode still requires a system message instructing the LLM to output␣
↪JSON.
            prompt_stack.add_system_message("Provide your response as a valid JSON␣
↪object.")
        else:
            params["response_format"] = self.response_format

    if prompt_stack.tools and self.use_native_tools:
        params["tools"] = self.__to_openai_tools(prompt_stack.tools)

    messages = self.__to_openai_messages(prompt_stack.messages)

    params["messages"] = messages

    return params
```

_to_delta_message_stream(result)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def _to_delta_message_stream(self, result: Stream[ChatCompletionChunk]) ->␣
↪Iterator[DeltaMessage]:
    for message in result:
        if message.usage is not None:
```

(continues on next page)

(continued from previous page)

```
            yield DeltaMessage(
                usage=DeltaMessage.Usage(
                    input_tokens=message.usage.prompt_tokens,
                    output_tokens=message.usage.completion_tokens,
                ),
            )
        if message.choices:
            choice = message.choices[0]
            delta = choice.delta

            content = self.__to_prompt_stack_delta_message_content(delta)

            if content is not None:
                yield DeltaMessage(content=content)
```

_to_message(result)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def _to_message(self, result: ChatCompletion) -> Message:
    if len(result.choices) == 1:
        choice_message = result.choices[0].message

        message = Message(
            content=self.__to_prompt_stack_message_content(choice_message),
            role=Message.ASSISTANT_ROLE,
        )
        if result.usage is not None:
            message.usage = Message.Usage(
                input_tokens=result.usage.prompt_tokens,
                output_tokens=result.usage.completion_tokens,
            )

        return message
    raise Exception("Completion with more than one choice is not supported yet.")
```

client()

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
@lazy_property()
def client(self) -> openai.OpenAI:
    return openai.OpenAI(
        base_url=self.base_url,
        api_key=self.api_key,
        organization=self.organization,
    )
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    params = self._base_params(prompt_stack)
    logger.debug(params)
    result = self.client.chat.completions.create(**params)

    logger.debug(result.model_dump())
    return self._to_message(result)
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    params = self._base_params(prompt_stack)
    logger.debug({"stream": True, **params})
    result = self.client.chat.completions.create(**params, stream=True)

    return self._to_delta_message_stream(result)
```

openai_chat_prompt_driver

- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *BasePromptDriver* “)

Attributes

Name	Type	Description
<a href="openai_chat_prompt_driver.griptape.drivers.prompt.openai_chat_prompt_driver.OpenAiChatPromptDriver.base_url" title="base_url = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.openai_chat_prompt_driver.OpenAiChatPromptDriver.base_url)"><code class="language-python">	*Optional[str]*	An optional OpenAi API URL.

continues on next page

Table 209 – continued from previous page

Name	Type	Description
<code class="language-python">	*Optional[str]*	An optional OpenAi API key. If not provided, the *OPENAI_API_KEY* environment variable will be used.
<code class="language-python">	*Optional[str]*	An optional OpenAI organization. If not provided, the *OPENAI_ORG_ID* environment variable will be used.
<code class="language-python">	*OpenAI*	An *openai.OpenAI* client.

continues on next page

Table 209 – continued from previous page

Name	Type	Description
<code class="language-python">	*str*	An OpenAI model name.
<code class="language-python">	<code class="language-python">BaseTokenizer</code>	An *OpenAiTokenizer*.

continues on next page

Table 209 – continued from previous page

Name	Type	Description
<a href="openai_chat_prompt_driver.r tape.drivers.prompt.openai_chat_prompt_driver.OpenAiChatPrompt-Driver.user" title="user = field(default=", kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.openai_chat_prompt_driver.OpenAiChatPrompt-Driver.user)"><code class="language-python">	*str*	A user id. Can be used to track requests by user.
<a href="openai_chat_prompt_driver.r tape.drivers.prompt.openai_chat_prompt_driver.OpenAiChatPrompt-Driver.response_format" title="response_format = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.openai_chat_prompt_driver.OpenAiChatPrompt-Driver.response_format)"><code class="language-python">	*Optional[dict]*	An optional OpenAi Chat Completion response format. Currently only supports *json_object* which will enable OpenAi's JSON mode.

continues on next page

Table 209 – continued from previous page

Name	Type	Description
<a href="openai_chat_prompt_driver.r tape.drivers.prompt.openai_chat_prompt_driver.C nAiChatPrompt-Driver.seed" title="seed = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.openai_chat_prompt_driver.C nAiChatPrompt-Driver.seed)"><code class="language-python">	*Optional[int]*	An optional OpenAi Chat Completion seed.
<a href="openai_chat_prompt_driver.r tape.drivers.prompt.openai_chat_prompt_driver.C nAiChatPrompt-Driver.ignored_exception_types" title="ignored_exception_types = field(default=Factory(lambda: (openai.BadRequestError, openai.AuthenticationError, openai.PermissionDeniedError, openai.NotFoundError, openai.ConflictError, openai.UnprocessableEntityError)), kw_only=True) class-attribute instance-attribute (griptape.drivers.prompt.openai_chat_prompt_driver.C nAiChatPrompt-Driver.ignored_exception_types)"><code class="language-python">	*tuple[type[Exception], ...]*	An optional tuple of exception types to ignore. Defaults to OpenAI's known exception types.

continues on next page

Table 209 – continued from previous page

Name	Type	Description
<a href="openai_chat_prompt_driver.r tape.drivers.prompt.openai_chat_prompt_driver.OpenAiChatPromptDriver.parallel_tool_calls" title="parallel_tool_calls = field(default=True, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.prompt.openai_chat_prompt_driver.OpenAiChatPromptDriver.parallel_tool_calls)"><code class="language-python">	*bool*	A flag to enable parallel tool calls. Defaults to *True*.

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
@define
class OpenAiChatPromptDriver(BasePromptDriver):
    """OpenAI Chat Prompt Driver.

    Attributes:
        base_url: An optional OpenAi API URL.
        api_key: An optional OpenAi API key. If not provided, the `OPENAI_API_KEY`␣
↪environment variable will be used.
        organization: An optional OpenAI organization. If not provided, the `OPENAI_
↪ORG_ID` environment variable will be used.
        client: An `openai.OpenAI` client.
        model: An OpenAI model name.
        tokenizer: An `OpenAiTokenizer`.
        user: A user id. Can be used to track requests by user.
        response_format: An optional OpenAi Chat Completion response format.␣
↪Currently only supports `json_object` which will enable OpenAis JSON mode.
        seed: An optional OpenAi Chat Completion seed.
        ignored_exception_types: An optional tuple of exception types to ignore.␣
↪Defaults to OpenAIs known exception types.
        parallel_tool_calls: A flag to enable parallel tool calls. Defaults to `True`.
    """

    base_url: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    organization: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
```

(continues on next page)

(continued from previous page)

```
    model: str = field(kw_only=True, metadata={"serializable": True})
    tokenizer: BaseTokenizer = field(
        default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_
↪self=True),
        kw_only=True,
    )
    user: str = field(default="", kw_only=True, metadata={"serializable": True})
    response_format: Optional[dict] = field(
        default=None,
        kw_only=True,
        metadata={"serializable": True},
    )
    seed: Optional[int] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    tool_choice: str = field(default="auto", kw_only=True, metadata={"serializable":␣
↪False})
    reasoning_effort: Literal["low", "medium", "high"] = field(
        default="medium", kw_only=True, metadata={"serializable": True}
    )
    use_native_tools: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})
    structured_output_strategy: StructuredOutputStrategy = field(
        default="native", kw_only=True, metadata={"serializable": True}
    )
    parallel_tool_calls: bool = field(default=True, kw_only=True, metadata={
↪"serializable": True})
    ignored_exception_types: tuple[type[Exception], ...] = field(
        default=Factory(
            lambda: (
                openai.BadRequestError,
                openai.AuthenticationError,
                openai.PermissionDeniedError,
                openai.NotFoundError,
                openai.ConflictError,
                openai.UnprocessableEntityError,
            ),
        ),
        kw_only=True,
    )
    modalities: list[str] = field(factory=list, kw_only=True, metadata={"serializable
↪": True})
    audio: dict = field(
        default=Factory(lambda: {"voice": "alloy", "format": "pcm16"}), kw_only=True,␣
↪metadata={"serializable": True}
    )
    _client: Optional[openai.OpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.OpenAI:
        return openai.OpenAI(
```

(continues on next page)

(continued from previous page)

```
            base_url=self.base_url,
            api_key=self.api_key,
            organization=self.organization,
        )

    @property
    def is_reasoning_model(self) -> bool:
        return self.model.startswith("o")

    @observable
    def try_run(self, prompt_stack: PromptStack) -> Message:
        params = self._base_params(prompt_stack)
        logger.debug(params)
        result = self.client.chat.completions.create(**params)

        logger.debug(result.model_dump())
        return self._to_message(result)

    @observable
    def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
        params = self._base_params(prompt_stack)
        logger.debug({"stream": True, **params})
        result = self.client.chat.completions.create(**params, stream=True)

        return self._to_delta_message_stream(result)

    def _to_message(self, result: ChatCompletion) -> Message:
        if len(result.choices) == 1:
            choice_message = result.choices[0].message

            message = Message(
                content=self.__to_prompt_stack_message_content(choice_message),
                role=Message.ASSISTANT_ROLE,
            )
            if result.usage is not None:
                message.usage = Message.Usage(
                    input_tokens=result.usage.prompt_tokens,
                    output_tokens=result.usage.completion_tokens,
                )

            return message
        raise Exception("Completion with more than one choice is not supported yet.")

    def _to_delta_message_stream(self, result: Stream[ChatCompletionChunk]) ->␣
↪Iterator[DeltaMessage]:
        for message in result:
            if message.usage is not None:
                yield DeltaMessage(
                    usage=DeltaMessage.Usage(
                        input_tokens=message.usage.prompt_tokens,
                        output_tokens=message.usage.completion_tokens,
                    ),
```

(continues on next page)

(continued from previous page)

```
                )
            if message.choices:
                choice = message.choices[0]
                delta = choice.delta

                content = self.__to_prompt_stack_delta_message_content(delta)

                if content is not None:
                    yield DeltaMessage(content=content)

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        params = {
            "model": self.model,
            **({"user": self.user} if self.user else {}),
            **({"seed": self.seed} if self.seed is not None else {}),
            **({"modalities": self.modalities} if self.modalities and not self.is_
↪reasoning_model else {}),
            **(
                {"reasoning_effort": self.reasoning_effort}
                if self.is_reasoning_model and self.model != "o1-mini"
                else {}
            ),
            **({"temperature": self.temperature} if not self.is_reasoning_model else
↪{}),
            **({"audio": self.audio} if "audio" in self.modalities else {}),
            **(
                {"stop": self.tokenizer.stop_sequences}
                if not self.is_reasoning_model and self.tokenizer.stop_sequences
                else {}
            ),
            **({"max_tokens": self.max_tokens} if self.max_tokens is not None else {}
↪),
            **({"stream_options": {"include_usage": True}} if self.stream else {}),
            **self.extra_params,
        }

        if prompt_stack.tools and self.use_native_tools:
            params["tool_choice"] = self.tool_choice
            params["parallel_tool_calls"] = self.parallel_tool_calls

        if prompt_stack.output_schema is not None:
            if self.structured_output_strategy == "native":
                params["response_format"] = {
                    "type": "json_schema",
                    "json_schema": {
                        "name": "Output",
                        "schema": prompt_stack.to_output_json_schema(),
                        "strict": True,
                    },
                }
            elif self.structured_output_strategy == "tool" and self.use_native_tools:
                params["tool_choice"] = "required"
```

(continues on next page)

(continued from previous page)

```
        if self.response_format is not None:
            if self.response_format == {"type": "json_object"}:
                params["response_format"] = self.response_format
                # JSON mode still requires a system message instructing the LLM to␣
↪output JSON.
                prompt_stack.add_system_message("Provide your response as a valid␣
↪JSON object.")
            else:
                params["response_format"] = self.response_format

        if prompt_stack.tools and self.use_native_tools:
            params["tools"] = self.__to_openai_tools(prompt_stack.tools)

        messages = self.__to_openai_messages(prompt_stack.messages)

        params["messages"] = messages

        return params

    def __to_openai_messages(self, messages: list[Message]) -> list[dict]:
        openai_messages = []

        for message in messages:
            # If the message only contains textual content we can send it as a single␣
↪content.
            if message.has_all_content_type(TextMessageContent):
                openai_messages.append({"role": self.__to_openai_role(message),
↪"content": message.to_text()})
            # Action results must be sent as separate messages.
            elif action_result_contents := message.get_content_
↪type(ActionResultMessageContent):
                openai_messages.extend(
                    {
                        "role": self.__to_openai_role(message, action_result_content),
                        "content": self.__to_openai_message_content(action_result_
↪content),
                        "tool_call_id": action_result_content.action.tag,
                    }
                    for action_result_content in action_result_contents
                )

                if message.has_any_content_type(TextMessageContent):
                    openai_messages.append({"role": self.__to_openai_role(message),
↪"content": message.to_text()})
            else:
                openai_message = {
                    "role": self.__to_openai_role(message),
                    "content": [],
                }

                for content in message.content:
```

(continues on next page)

(continued from previous page)

```
                    if isinstance(content, ActionCallMessageContent):
                        if "tool_calls" not in openai_message:
                            openai_message["tool_calls"] = []
                        openai_message["tool_calls"].append(self.__to_openai_message_
↪content(content))
                    elif (
                        isinstance(content, AudioMessageContent)
                        and message.is_assistant()
                        and time.time() < content.artifact.meta["expires_at"]
                    ):
                        # For assistant audio messages, we reference the audio id␣
↪instead of sending audio message content.
                        openai_message["audio"] = {
                            "id": content.artifact.meta["audio_id"],
                        }
                    else:
                        openai_message["content"].append(self.__to_openai_message_
↪content(content))

                # Some OpenAi-compatible services dont accept an empty array for␣
↪content
                if not openai_message["content"]:
                    del openai_message["content"]

                openai_messages.append(openai_message)

        return openai_messages

    def __to_openai_role(self, message: Message, message_content:␣
↪Optional[BaseMessageContent] = None) -> str:
        if message.is_system():
            if self.is_reasoning_model:
                return "developer"
            return "system"
        if message.is_assistant():
            return "assistant"
        if isinstance(message_content, ActionResultMessageContent):
            return "tool"
        return "user"

    def __to_openai_tools(self, tools: list[BaseTool]) -> list[dict]:
        return [
            {
                "function": {
                    "name": tool.to_native_tool_name(activity),
                    "description": tool.activity_description(activity),
                    "parameters": tool.to_activity_json_schema(activity, "Parameters␣
↪Schema"),
                },
                "type": "function",
            }
            for tool in tools
```

(continues on next page)

(continued from previous page)

```
            for activity in tool.activities()
        ]

    def __to_openai_message_content(self, content: BaseMessageContent) -> str | dict:
        if isinstance(content, TextMessageContent):
            return {"type": "text", "text": content.artifact.to_text()}
        if isinstance(content, ImageMessageContent):
            if isinstance(content.artifact, ImageArtifact):
                return {
                    "type": "image_url",
                    "image_url": {"url": f"data:{content.artifact.mime_type}; base64,
↪{content.artifact.base64}"},
                }
            if isinstance(content.artifact, ImageUrlArtifact):
                return {
                    "type": "image_url",
                    "image_url": {"url": content.artifact.value},
                }
            raise ValueError(f"Unsupported image artifact type: {type(content.
↪artifact)}")
        if isinstance(content, AudioMessageContent):
            artifact = content.artifact
            metadata = artifact.meta

            # If theres an expiration date, we can assume its an assistant message.
            if "expires_at" in metadata:
                # If its expired, we send the transcript instead.
                if time.time() >= metadata["expires_at"]:
                    return {
                        "type": "text",
                        "text": artifact.meta.get("transcript"),
                    }
                # This should never occur, since a non-expired audio content
                # should have already been referenced by the audio id.
                raise ValueError("Assistant audio messages should be sent as audio␣
↪ids.")
            # If theres no expiration date, we can assume its a user message where we␣
↪send the audio every time.
            return {
                "type": "input_audio",
                "input_audio": {
                    "data": base64.b64encode(artifact.value).decode("utf-8"),
                    "format": artifact.format,
                },
            }
        if isinstance(content, ActionCallMessageContent):
            action = content.artifact.value

            return {
                "type": "function",
                "id": action.tag,
                "function": {"name": action.to_native_tool_name(), "arguments": json.
```

(continues on next page)

(continued from previous page)

```
↪dumps(action.input)},
            }
        if isinstance(content, ActionResultMessageContent):
            return content.artifact.to_text()
        return {"type": "text", "text": content.artifact.to_text()}

    def __to_prompt_stack_message_content(self, response: ChatCompletionMessage) ->␣
↪list[BaseMessageContent]:
        content = []

        if response.content is not None:
            content.append(TextMessageContent(TextArtifact(response.content)))
        if hasattr(response, "audio") and response.audio is not None:
            content.append(
                AudioMessageContent(
                    AudioArtifact(
                        value=base64.b64decode(response.audio.data),
                        format="wav",
                        meta={
                            "audio_id": response.audio.id,
                            "transcript": response.audio.transcript,
                            "expires_at": response.audio.expires_at,
                        },
                    )
                )
            )
        if response.tool_calls is not None:
            content.extend(
                [
                    ActionCallMessageContent(
                        ActionArtifact(
                            ToolAction(
                                tag=tool_call.id,
                                name=ToolAction.from_native_tool_name(tool_call.
↪function.name)[0],
                                path=ToolAction.from_native_tool_name(tool_call.
↪function.name)[1],
                                input=json.loads(tool_call.function.arguments),
                            ),
                        ),
                    )
                    for tool_call in response.tool_calls
                ],
            )

        return content

    def __to_prompt_stack_delta_message_content(self, content_delta: ChoiceDelta) ->␣
↪Optional[BaseDeltaMessageContent]:
        if content_delta.content is not None:
            return TextDeltaMessageContent(content_delta.content)
        if content_delta.tool_calls is not None:
```

(continues on next page)

(continued from previous page)

```
            tool_calls = content_delta.tool_calls

            if len(tool_calls) == 1:
                tool_call = tool_calls[0]
                index = tool_call.index

                if tool_call.function is not None:
                    function_name = tool_call.function.name
                    return ActionCallDeltaMessageContent(
                        index=index,
                        tag=tool_call.id,
                        name=ToolAction.from_native_tool_name(function_name)[0] if␣
↪function_name else None,
                        path=ToolAction.from_native_tool_name(function_name)[1] if␣
↪function_name else None,
                        partial_input=tool_call.function.arguments,
                    )
                raise ValueError(f"Unsupported tool call delta: {tool_call}")
            raise ValueError(f"Unsupported tool call delta length: {len(tool_calls)}")
        # OpenAi doesnt have types for audio deltas so we need to use hasattr and␣
↪getattr.
        if hasattr(content_delta, "audio") and getattr(content_delta, "audio") is not␣
↪None:
            audio_chunk: dict = getattr(content_delta, "audio")
            return AudioDeltaMessageContent(
                id=audio_chunk.get("id"),
                data=audio_chunk.get("data"),
                expires_at=audio_chunk.get("expires_at"),
                transcript=audio_chunk.get("transcript"),
            )
        return None
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `audio = field(default=Factory(lambda: {'voice': 'alloy', 'format': 'pcm16'}), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `base_url = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `ignored_exception_types = field(default=Factory(lambda: (openai.BadRequestError, openai.AuthenticationError, openai.PermissionDeniedError, openai.NotFoundError, openai.ConflictError, openai.UnprocessableEntityError)), kw_only=True)` class-attribute instance-attribute
- `is_reasoning_model` property
- `modalities = field(factory=list, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

- `organization = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `parallel_tool_calls = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `reasoning_effort = field(default='medium', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `response_format = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `seed = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='native', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: OpenAiTokenizer(model=self.model), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `tool_choice = field(default='auto', kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `use_native_tools = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `user = field(default='', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__to_openai_message_content(content)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def __to_openai_message_content(self, content: BaseMessageContent) -> str | dict:
    if isinstance(content, TextMessageContent):
        return {"type": "text", "text": content.artifact.to_text()}
    if isinstance(content, ImageMessageContent):
        if isinstance(content.artifact, ImageArtifact):
            return {
                "type": "image_url",
                "image_url": {"url": f"data:{content.artifact.mime_type}; base64,
↪{content.artifact.base64}"},
            }
        if isinstance(content.artifact, ImageUrlArtifact):
            return {
                "type": "image_url",
                "image_url": {"url": content.artifact.value},
            }
        raise ValueError(f"Unsupported image artifact type: {type(content.artifact)}")
    if isinstance(content, AudioMessageContent):
        artifact = content.artifact
        metadata = artifact.meta

        # If theres an expiration date, we can assume its an assistant message.
        if "expires_at" in metadata:
            # If its expired, we send the transcript instead.
            if time.time() >= metadata["expires_at"]:
```

(continues on next page)

(continued from previous page)

```
                return {
                    "type": "text",
                    "text": artifact.meta.get("transcript"),
                }
            # This should never occur, since a non-expired audio content
            # should have already been referenced by the audio id.
            raise ValueError("Assistant audio messages should be sent as audio ids.")
        # If theres no expiration date, we can assume its a user message where we␣
↪send the audio every time.
        return {
            "type": "input_audio",
            "input_audio": {
                "data": base64.b64encode(artifact.value).decode("utf-8"),
                "format": artifact.format,
            },
        }
    if isinstance(content, ActionCallMessageContent):
        action = content.artifact.value

        return {
            "type": "function",
            "id": action.tag,
            "function": {"name": action.to_native_tool_name(), "arguments": json.
↪dumps(action.input)},
        }
    if isinstance(content, ActionResultMessageContent):
        return content.artifact.to_text()
    return {"type": "text", "text": content.artifact.to_text()}
```

__to_openai_messages(messages)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def __to_openai_messages(self, messages: list[Message]) -> list[dict]:
    openai_messages = []

    for message in messages:
        # If the message only contains textual content we can send it as a single␣
↪content.
        if message.has_all_content_type(TextMessageContent):
            openai_messages.append({"role": self.__to_openai_role(message), "content
↪": message.to_text()})
        # Action results must be sent as separate messages.
        elif action_result_contents := message.get_content_
↪type(ActionResultMessageContent):
            openai_messages.extend(
                {
                    "role": self.__to_openai_role(message, action_result_content),
                    "content": self.__to_openai_message_content(action_result_
↪content),
                    "tool_call_id": action_result_content.action.tag,
                }
```

(continues on next page)

(continued from previous page)

```
                for action_result_content in action_result_contents
            )

            if message.has_any_content_type(TextMessageContent):
                openai_messages.append({"role": self.__to_openai_role(message),
↪"content": message.to_text()})
        else:
            openai_message = {
                "role": self.__to_openai_role(message),
                "content": [],
            }

            for content in message.content:
                if isinstance(content, ActionCallMessageContent):
                    if "tool_calls" not in openai_message:
                        openai_message["tool_calls"] = []
                    openai_message["tool_calls"].append(self.__to_openai_message_
↪content(content))
                elif (
                    isinstance(content, AudioMessageContent)
                    and message.is_assistant()
                    and time.time() < content.artifact.meta["expires_at"]
                ):
                    # For assistant audio messages, we reference the audio id instead␣
↪of sending audio message content.
                    openai_message["audio"] = {
                        "id": content.artifact.meta["audio_id"],
                    }
                else:
                    openai_message["content"].append(self.__to_openai_message_
↪content(content))

            # Some OpenAi-compatible services dont accept an empty array for content
            if not openai_message["content"]:
                del openai_message["content"]

            openai_messages.append(openai_message)

    return openai_messages
```

__to_openai_role(message, message_content=None)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def __to_openai_role(self, message: Message, message_content:␣
↪Optional[BaseMessageContent] = None) -> str:
    if message.is_system():
        if self.is_reasoning_model:
            return "developer"
        return "system"
    if message.is_assistant():
        return "assistant"
```

(continues on next page)

(continued from previous page)

```
    if isinstance(message_content, ActionResultMessageContent):
        return "tool"
    return "user"
```

__to_openai_tools(tools)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def __to_openai_tools(self, tools: list[BaseTool]) -> list[dict]:
    return [
        {
            "function": {
                "name": tool.to_native_tool_name(activity),
                "description": tool.activity_description(activity),
                "parameters": tool.to_activity_json_schema(activity, "Parameters␣
↪Schema"),
            },
            "type": "function",
        }
        for tool in tools
        for activity in tool.activities()
    ]
```

__to_prompt_stack_delta_message_content(content_delta)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def __to_prompt_stack_delta_message_content(self, content_delta: ChoiceDelta) ->␣
↪Optional[BaseDeltaMessageContent]:
    if content_delta.content is not None:
        return TextDeltaMessageContent(content_delta.content)
    if content_delta.tool_calls is not None:
        tool_calls = content_delta.tool_calls

        if len(tool_calls) == 1:
            tool_call = tool_calls[0]
            index = tool_call.index

            if tool_call.function is not None:
                function_name = tool_call.function.name
                return ActionCallDeltaMessageContent(
                    index=index,
                    tag=tool_call.id,
                    name=ToolAction.from_native_tool_name(function_name)[0] if␣
↪function_name else None,
                    path=ToolAction.from_native_tool_name(function_name)[1] if␣
↪function_name else None,
                    partial_input=tool_call.function.arguments,
                )
            raise ValueError(f"Unsupported tool call delta: {tool_call}")
        raise ValueError(f"Unsupported tool call delta length: {len(tool_calls)}")
```

(continues on next page)

(continued from previous page)

```
    # OpenAi doesnt have types for audio deltas so we need to use hasattr and getattr.
    if hasattr(content_delta, "audio") and getattr(content_delta, "audio") is not␣
↪None:
        audio_chunk: dict = getattr(content_delta, "audio")
        return AudioDeltaMessageContent(
            id=audio_chunk.get("id"),
            data=audio_chunk.get("data"),
            expires_at=audio_chunk.get("expires_at"),
            transcript=audio_chunk.get("transcript"),
        )
    return None
```

__to_prompt_stack_message_content(response)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def __to_prompt_stack_message_content(self, response: ChatCompletionMessage) ->␣
↪list[BaseMessageContent]:
    content = []

    if response.content is not None:
        content.append(TextMessageContent(TextArtifact(response.content)))
    if hasattr(response, "audio") and response.audio is not None:
        content.append(
            AudioMessageContent(
                AudioArtifact(
                    value=base64.b64decode(response.audio.data),
                    format="wav",
                    meta={
                        "audio_id": response.audio.id,
                        "transcript": response.audio.transcript,
                        "expires_at": response.audio.expires_at,
                    },
                )
            )
        )
    if response.tool_calls is not None:
        content.extend(
            [
                ActionCallMessageContent(
                    ActionArtifact(
                        ToolAction(
                            tag=tool_call.id,
                            name=ToolAction.from_native_tool_name(tool_call.function.
↪name)[0],
                            path=ToolAction.from_native_tool_name(tool_call.function.
↪name)[1],
                            input=json.loads(tool_call.function.arguments),
                        ),
                    ),
                )
                for tool_call in response.tool_calls
```

(continues on next page)

(continued from previous page)

```
            ],
        )

    return content
```

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    params = {
        "model": self.model,
        **({"user": self.user} if self.user else {}),
        **({"seed": self.seed} if self.seed is not None else {}),
        **({"modalities": self.modalities} if self.modalities and not self.is_
↪reasoning_model else {}),
        **(
            {"reasoning_effort": self.reasoning_effort}
            if self.is_reasoning_model and self.model != "o1-mini"
            else {}
        ),
        **({"temperature": self.temperature} if not self.is_reasoning_model else {}),
        **({"audio": self.audio} if "audio" in self.modalities else {}),
        **(
            {"stop": self.tokenizer.stop_sequences}
            if not self.is_reasoning_model and self.tokenizer.stop_sequences
            else {}
        ),
        **({"max_tokens": self.max_tokens} if self.max_tokens is not None else {}),
        **({"stream_options": {"include_usage": True}} if self.stream else {}),
        **self.extra_params,
    }

    if prompt_stack.tools and self.use_native_tools:
        params["tool_choice"] = self.tool_choice
        params["parallel_tool_calls"] = self.parallel_tool_calls

    if prompt_stack.output_schema is not None:
        if self.structured_output_strategy == "native":
            params["response_format"] = {
                "type": "json_schema",
                "json_schema": {
                    "name": "Output",
                    "schema": prompt_stack.to_output_json_schema(),
                    "strict": True,
                },
            }
        elif self.structured_output_strategy == "tool" and self.use_native_tools:
            params["tool_choice"] = "required"

    if self.response_format is not None:
        if self.response_format == {"type": "json_object"}:
```

(continues on next page)

(continued from previous page)

```
            params["response_format"] = self.response_format
            # JSON mode still requires a system message instructing the LLM to output␣
↪JSON.
            prompt_stack.add_system_message("Provide your response as a valid JSON␣
↪object.")
        else:
            params["response_format"] = self.response_format

    if prompt_stack.tools and self.use_native_tools:
        params["tools"] = self.__to_openai_tools(prompt_stack.tools)

    messages = self.__to_openai_messages(prompt_stack.messages)

    params["messages"] = messages

    return params
```

_to_delta_message_stream(result)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def _to_delta_message_stream(self, result: Stream[ChatCompletionChunk]) ->␣
↪Iterator[DeltaMessage]:
    for message in result:
        if message.usage is not None:
            yield DeltaMessage(
                usage=DeltaMessage.Usage(
                    input_tokens=message.usage.prompt_tokens,
                    output_tokens=message.usage.completion_tokens,
                ),
            )
        if message.choices:
            choice = message.choices[0]
            delta = choice.delta

            content = self.__to_prompt_stack_delta_message_content(delta)

            if content is not None:
                yield DeltaMessage(content=content)
```

_to_message(result)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
def _to_message(self, result: ChatCompletion) -> Message:
    if len(result.choices) == 1:
        choice_message = result.choices[0].message

        message = Message(
            content=self.__to_prompt_stack_message_content(choice_message),
            role=Message.ASSISTANT_ROLE,
```

(continues on next page)

(continued from previous page)

```
        )
        if result.usage is not None:
            message.usage = Message.Usage(
                input_tokens=result.usage.prompt_tokens,
                output_tokens=result.usage.completion_tokens,
            )

        return message
    raise Exception("Completion with more than one choice is not supported yet.")
```

client()

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
@lazy_property()
def client(self) -> openai.OpenAI:
    return openai.OpenAI(
        base_url=self.base_url,
        api_key=self.api_key,
        organization=self.organization,
    )
```

try_run(prompt_stack)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
@observable
def try_run(self, prompt_stack: PromptStack) -> Message:
    params = self._base_params(prompt_stack)
    logger.debug(params)
    result = self.client.chat.completions.create(**params)

    logger.debug(result.model_dump())
    return self._to_message(result)
```

try_stream(prompt_stack)

Source Code in griptape/drivers/prompt/openai_chat_prompt_driver.py

```
@observable
def try_stream(self, prompt_stack: PromptStack) -> Iterator[DeltaMessage]:
    params = self._base_params(prompt_stack)
    logger.debug({"stream": True, **params})
    result = self.client.chat.completions.create(**params, stream=True)

    return self._to_delta_message_stream(result)
```

perplexity

- `__all__ = ['PerplexityPromptDriver']` module-attribute

Bases: *AzureOpenAiChatPromptDriver* “)

Source Code in griptape/drivers/prompt/perplexity_prompt_driver.py

```
@define
class PerplexityPromptDriver(OpenAiChatPromptDriver):
    base_url: str = field(default="https://api.perplexity.ai", kw_only=True, metadata=
↪{"serializable": True})
    structured_output_strategy: str = field(default="native", kw_only=True, metadata={
↪"serializable": True})

    @override
    def _to_message(self, result: ChatCompletion) -> Message:
        message = super()._to_message(result)

        message.content[0].artifact.meta["citations"] = getattr(result, "citations",␣
↪[])

        return message

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        params = super()._base_params(prompt_stack)

        if "stop" in params:
            del params["stop"]

        return params
```

- `base_url = field(default='https://api.perplexity.ai', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='native', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/perplexity_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    params = super()._base_params(prompt_stack)

    if "stop" in params:
        del params["stop"]

    return params
```

_to_message(result)

Source Code in griptape/drivers/prompt/perplexity_prompt_driver.py

```
@override
def _to_message(self, result: ChatCompletion) -> Message:
    message = super()._to_message(result)

    message.content[0].artifact.meta["citations"] = getattr(result, "citations", [])

    return message
```

perplexity_prompt_driver

Bases: *AzureOpenAiChatPromptDriver* “)

Source Code in griptape/drivers/prompt/perplexity_prompt_driver.py

```
@define
class PerplexityPromptDriver(OpenAiChatPromptDriver):
    base_url: str = field(default="https://api.perplexity.ai", kw_only=True, metadata=
↪{"serializable": True})
    structured_output_strategy: str = field(default="native", kw_only=True, metadata={
↪"serializable": True})

    @override
    def _to_message(self, result: ChatCompletion) -> Message:
        message = super()._to_message(result)

        message.content[0].artifact.meta["citations"] = getattr(result, "citations",␣
↪[])

        return message

    def _base_params(self, prompt_stack: PromptStack) -> dict:
        params = super()._base_params(prompt_stack)

        if "stop" in params:
            del params["stop"]

        return params
```

- `base_url = field(default='https://api.perplexity.ai', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structured_output_strategy = field(default='native', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_base_params(prompt_stack)

Source Code in griptape/drivers/prompt/perplexity_prompt_driver.py

```
def _base_params(self, prompt_stack: PromptStack) -> dict:
    params = super()._base_params(prompt_stack)

    if "stop" in params:
        del params["stop"]
```

(continues on next page)

(continued from previous page)

```
    return params
```

_to_message(result)

Source Code in griptape/drivers/prompt/perplexity_prompt_driver.py

```
@override
def _to_message(self, result: ChatCompletion) -> Message:
    message = super()._to_message(result)

    message.content[0].artifact.meta["citations"] = getattr(result, "citations", [])

    return message
```

rerank

- `__all__ = ['BaseRerankDriver']` module-attribute

Bases:

ABC

Source Code in griptape/drivers/rerank/base_rerank_driver.py

```
@define(kw_only=True)
class BaseRerankDriver(ABC):
    @abstractmethod
    def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]: ..
↪.
```

run(query, artifacts)abstractmethod

Source Code in griptape/drivers/rerank/base_rerank_driver.py

```
@abstractmethod
def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]: ...
```

base_rerank_driver

Bases:

ABC

Source Code in griptape/drivers/rerank/base_rerank_driver.py

```
@define(kw_only=True)
class BaseRerankDriver(ABC):
    @abstractmethod
    def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]: ..
↪.
```

run(query, artifacts)abstractmethod

Source Code in griptape/drivers/rerank/base_rerank_driver.py

```
@abstractmethod
def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]: ...
```

cohere

- `__all__ = ['CohereRerankDriver']` module-attribute

Bases: *BaseRerankDriver* “)

Source Code in griptape/drivers/rerank/cohere_rerank_driver.py

```
@define(kw_only=True)
class CohereRerankDriver(BaseRerankDriver):
    model: str = field(default="rerank-english-v3.0", metadata={"serializable": True})
    top_n: Optional[int] = field(default=None)

    api_key: str = field(metadata={"serializable": True})
    client: Client = field(
        default=Factory(lambda self: import_optional_dependency("cohere").Client(self.
↪api_key), takes_self=True),
    )

    def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]:
        # Cohere errors out if passed "empty" documents or no documents at all
        artifacts_dict = {str(hash(a.to_text())): a for a in artifacts if a}

        if artifacts_dict:
            response = self.client.rerank(
                model=self.model,
                query=query,
                documents=[a.to_text() for a in artifacts_dict.values()],
                return_documents=True,
                top_n=self.top_n,
            )
            return [artifacts_dict[str(hash(r.document.text))] for r in response.
↪results if r.document is not None]
        return []
```

- `api_key = field(metadata={'serializable': True})` class-attribute instance-attribute
- `client = field(default=Factory(lambda self: import_optional_dependency('cohere').Client(self.api_key), takes_self=True))` class-attribute instance-attribute
- `model = field(default='rerank-english-v3.0', metadata={'serializable': True})` class-attribute instance-attribute
- `top_n = field(default=None)` class-attribute instance-attribute

run(query, artifacts)

Source Code in griptape/drivers/rerank/cohere_rerank_driver.py

```
def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]:
    # Cohere errors out if passed "empty" documents or no documents at all
    artifacts_dict = {str(hash(a.to_text())): a for a in artifacts if a}

    if artifacts_dict:
        response = self.client.rerank(
            model=self.model,
            query=query,
            documents=[a.to_text() for a in artifacts_dict.values()],
            return_documents=True,
            top_n=self.top_n,
        )
        return [artifacts_dict[str(hash(r.document.text))] for r in response.results␣
↪if r.document is not None]
    return []
```

cohere_rerank_driver

Bases: *BaseRerankDriver* “)

Source Code in griptape/drivers/rerank/cohere_rerank_driver.py

```
@define(kw_only=True)
class CohereRerankDriver(BaseRerankDriver):
    model: str = field(default="rerank-english-v3.0", metadata={"serializable": True})
    top_n: Optional[int] = field(default=None)

    api_key: str = field(metadata={"serializable": True})
    client: Client = field(
        default=Factory(lambda self: import_optional_dependency("cohere").Client(self.
↪api_key), takes_self=True),
    )

    def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]:
        # Cohere errors out if passed "empty" documents or no documents at all
        artifacts_dict = {str(hash(a.to_text())): a for a in artifacts if a}

        if artifacts_dict:
            response = self.client.rerank(
                model=self.model,
                query=query,
                documents=[a.to_text() for a in artifacts_dict.values()],
                return_documents=True,
                top_n=self.top_n,
            )
            return [artifacts_dict[str(hash(r.document.text))] for r in response.
↪results if r.document is not None]
        return []
```

- `api_key = field(metadata={'serializable': True})` class-attribute instance-attribute

- `client = field(default=Factory(lambda self: import_optional_dependency('cohere').Client(self.api_key), takes_self=True))` class-attribute instance-attribute
- `model = field(default='rerank-english-v3.0', metadata={'serializable': True})` class-attribute instance-attribute
- `top_n = field(default=None)` class-attribute instance-attribute

run(query, artifacts)

Source Code in griptape/drivers/rerank/cohere_rerank_driver.py

```
def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]:
    # Cohere errors out if passed "empty" documents or no documents at all
    artifacts_dict = {str(hash(a.to_text())): a for a in artifacts if a}

    if artifacts_dict:
        response = self.client.rerank(
            model=self.model,
            query=query,
            documents=[a.to_text() for a in artifacts_dict.values()],
            return_documents=True,
            top_n=self.top_n,
        )
        return [artifacts_dict[str(hash(r.document.text))] for r in response.results␣
↪if r.document is not None]
    return []
```

local

- `__all__ = ['LocalRerankDriver']` module-attribute

Bases: *BaseRerankDriver* “) , *FuturesExecutorMixin* ”)

Source Code in griptape/drivers/rerank/local_rerank_driver.py

```
@define(kw_only=True)
class LocalRerankDriver(BaseRerankDriver, FuturesExecutorMixin):
    calculate_relatedness: Callable = field(default=lambda x, y: dot(x, y) / (norm(x)␣
↪* norm(y)))
    embedding_driver: BaseEmbeddingDriver = field(
        kw_only=True, default=Factory(lambda: Defaults.drivers_config.embedding_
↪driver), metadata={"serializable": True}
    )

    def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]:
        query_embedding = self.embedding_driver.embed(query, vector_operation="query")

        with self.create_futures_executor() as futures_executor:
            artifact_embeddings = execute_futures_list(
                [
                    futures_executor.submit(
                        with_contextvars(self.embedding_driver.embed_text_artifact),␣
↪a, vector_operation="upsert"
                    )
```

(continues on next page)

(continued from previous page)

```
                    for a in artifacts
                ],
            )

        artifacts_and_relatednesses = [
            (artifact, self.calculate_relatedness(query_embedding, artifact_
↪embedding))
            for artifact, artifact_embedding in zip(artifacts, artifact_embeddings)
        ]

        artifacts_and_relatednesses.sort(key=operator.itemgetter(1), reverse=True)

        return [artifact for artifact, _ in artifacts_and_relatednesses]
```

- `calculate_relatedness = field(default=lambda x, y: dot(x, y) / norm(x) * norm(y))` class-attribute instance-attribute
- `embedding_driver = field(kw_only=True, default=Factory(lambda: Defaults.drivers_config.embedding_driver), metadata={'serializable': True})` class-attribute instance-attribute

run(query, artifacts)

Source Code in griptape/drivers/rerank/local_rerank_driver.py

```
def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]:
    query_embedding = self.embedding_driver.embed(query, vector_operation="query")

    with self.create_futures_executor() as futures_executor:
        artifact_embeddings = execute_futures_list(
            [
                futures_executor.submit(
                    with_contextvars(self.embedding_driver.embed_text_artifact), a,␣
↪vector_operation="upsert"
                )
                for a in artifacts
            ],
        )

    artifacts_and_relatednesses = [
        (artifact, self.calculate_relatedness(query_embedding, artifact_embedding))
        for artifact, artifact_embedding in zip(artifacts, artifact_embeddings)
    ]

    artifacts_and_relatednesses.sort(key=operator.itemgetter(1), reverse=True)

    return [artifact for artifact, _ in artifacts_and_relatednesses]
```

local_rerank_driver

Bases: *BaseRerankDriver* “) , *FuturesExecutorMixin* ”)

Source Code in griptape/drivers/rerank/local_rerank_driver.py

```
@define(kw_only=True)
class LocalRerankDriver(BaseRerankDriver, FuturesExecutorMixin):
    calculate_relatedness: Callable = field(default=lambda x, y: dot(x, y) / (norm(x)␣
↪* norm(y)))
    embedding_driver: BaseEmbeddingDriver = field(
        kw_only=True, default=Factory(lambda: Defaults.drivers_config.embedding_
↪driver), metadata={"serializable": True}
    )

    def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]:
        query_embedding = self.embedding_driver.embed(query, vector_operation="query")

        with self.create_futures_executor() as futures_executor:
            artifact_embeddings = execute_futures_list(
                [
                    futures_executor.submit(
                        with_contextvars(self.embedding_driver.embed_text_artifact),␣
↪a, vector_operation="upsert"
                    )
                    for a in artifacts
                ],
            )

        artifacts_and_relatednesses = [
            (artifact, self.calculate_relatedness(query_embedding, artifact_
↪embedding))
            for artifact, artifact_embedding in zip(artifacts, artifact_embeddings)
        ]

        artifacts_and_relatednesses.sort(key=operator.itemgetter(1), reverse=True)

        return [artifact for artifact, _ in artifacts_and_relatednesses]
```

- `calculate_relatedness = field(default=lambda x, y: dot(x, y) / norm(x) * norm(y))` class-attribute instance-attribute
- `embedding_driver = field(kw_only=True, default=Factory(lambda: Defaults.drivers_config.embedding_driver), metadata={'serializable': True})` class-attribute instance-attribute

run(query, artifacts)

Source Code in griptape/drivers/rerank/local_rerank_driver.py

```
def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]:
    query_embedding = self.embedding_driver.embed(query, vector_operation="query")

    with self.create_futures_executor() as futures_executor:
        artifact_embeddings = execute_futures_list(
```

(continues on next page)

(continued from previous page)

```
            [
                futures_executor.submit(
                    with_contextvars(self.embedding_driver.embed_text_artifact), a,␣
↪vector_operation="upsert"
                )
                for a in artifacts
            ],
        )

    artifacts_and_relatednesses = [
        (artifact, self.calculate_relatedness(query_embedding, artifact_embedding))
        for artifact, artifact_embedding in zip(artifacts, artifact_embeddings)
    ]

    artifacts_and_relatednesses.sort(key=operator.itemgetter(1), reverse=True)

    return [artifact for artifact, _ in artifacts_and_relatednesses]
```

nvidia_nim

- `__all__ = ['NvidiaNimRerankDriver']` module-attribute

Bases: *BaseRerankDriver* “)

Source Code in griptape/drivers/rerank/nvidia_nim_rerank_driver.py

```
@define(kw_only=True)
class NvidiaNimRerankDriver(BaseRerankDriver):
    """Nvidia Rerank Driver."""

    model: str = field()
    base_url: str = field()
    truncate: Literal["NONE", "END"] = field(default="NONE")
    headers: dict = field(factory=dict)

    def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]:
        if not artifacts:
            return []

        response = requests.post(
            url=f"{self.base_url.rstrip(/)}/v1/ranking",
            json=self._get_body(query, artifacts),
            headers=self.headers,
        )

        response.raise_for_status()

        ranked_artifacts = []
        for ranking in response.json()["rankings"]:
            artifact = artifacts[ranking["index"]]
            artifact.meta.update({"logit": ranking["logit"], "usage": ranking.get(
↪"usage")})
            ranked_artifacts.append(artifact)
```

(continues on next page)

(continued from previous page)

```
        return ranked_artifacts

    def _get_body(self, query: str, artifacts: list[TextArtifact]) -> dict:
        return {
            "model": self.model,
            "query": {"text": query},
            "passages": [{"text": artifact.value} for artifact in artifacts],
            "truncate": self.truncate,
        }
```

- `base_url = field()` class-attribute instance-attribute
- `headers = field(factory=dict)` class-attribute instance-attribute
- `model = field()` class-attribute instance-attribute
- `truncate = field(default='NONE')` class-attribute instance-attribute

_get_body(query, artifacts)

Source Code in griptape/drivers/rerank/nvidia_nim_rerank_driver.py

```
def _get_body(self, query: str, artifacts: list[TextArtifact]) -> dict:
    return {
        "model": self.model,
        "query": {"text": query},
        "passages": [{"text": artifact.value} for artifact in artifacts],
        "truncate": self.truncate,
    }
```

run(query, artifacts)

Source Code in griptape/drivers/rerank/nvidia_nim_rerank_driver.py

```
def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]:
    if not artifacts:
        return []

    response = requests.post(
        url=f"{self.base_url.rstrip(/)}/v1/ranking",
        json=self._get_body(query, artifacts),
        headers=self.headers,
    )

    response.raise_for_status()

    ranked_artifacts = []
    for ranking in response.json()["rankings"]:
        artifact = artifacts[ranking["index"]]
        artifact.meta.update({"logit": ranking["logit"], "usage": ranking.get("usage
↪")})
        ranked_artifacts.append(artifact)
```

(continues on next page)

(continued from previous page)

```
    return ranked_artifacts
```

nvidia_nim_rerank_driver

Bases: *BaseRerankDriver* “)

Source Code in griptape/drivers/rerank/nvidia_nim_rerank_driver.py

```
@define(kw_only=True)
class NvidiaNimRerankDriver(BaseRerankDriver):
    """Nvidia Rerank Driver."""

    model: str = field()
    base_url: str = field()
    truncate: Literal["NONE", "END"] = field(default="NONE")
    headers: dict = field(factory=dict)

    def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]:
        if not artifacts:
            return []

        response = requests.post(
            url=f"{self.base_url.rstrip(/)}/v1/ranking",
            json=self._get_body(query, artifacts),
            headers=self.headers,
        )

        response.raise_for_status()

        ranked_artifacts = []
        for ranking in response.json()["rankings"]:
            artifact = artifacts[ranking["index"]]
            artifact.meta.update({"logit": ranking["logit"], "usage": ranking.get(
↪"usage")})
            ranked_artifacts.append(artifact)

        return ranked_artifacts

    def _get_body(self, query: str, artifacts: list[TextArtifact]) -> dict:
        return {
            "model": self.model,
            "query": {"text": query},
            "passages": [{"text": artifact.value} for artifact in artifacts],
            "truncate": self.truncate,
        }
```

- `base_url = field()` class-attribute instance-attribute
- `headers = field(factory=dict)` class-attribute instance-attribute
- `model = field()` class-attribute instance-attribute
- `truncate = field(default='NONE')` class-attribute instance-attribute

_get_body(query, artifacts)

Source Code in griptape/drivers/rerank/nvidia_nim_rerank_driver.py

```
def _get_body(self, query: str, artifacts: list[TextArtifact]) -> dict:
    return {
        "model": self.model,
        "query": {"text": query},
        "passages": [{"text": artifact.value} for artifact in artifacts],
        "truncate": self.truncate,
    }
```

run(query, artifacts)

Source Code in griptape/drivers/rerank/nvidia_nim_rerank_driver.py

```
def run(self, query: str, artifacts: list[TextArtifact]) -> list[TextArtifact]:
    if not artifacts:
        return []

    response = requests.post(
        url=f"{self.base_url.rstrip(/)}/v1/ranking",
        json=self._get_body(query, artifacts),
        headers=self.headers,
    )

    response.raise_for_status()

    ranked_artifacts = []
    for ranking in response.json()["rankings"]:
        artifact = artifacts[ranking["index"]]
        artifact.meta.update({"logit": ranking["logit"], "usage": ranking.get("usage
↪")})
        ranked_artifacts.append(artifact)

    return ranked_artifacts
```

ruleset

- `__all__ = ['BaseRulesetDriver']` module-attribute

Bases: *SerializableMixin* “) , ABC

Attributes

Name	Type	Description
<a href="./#griptape.drivers.ruleset.BaseRulesetDriver.raise_not_found" title="raise_not_found = field(default=True, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.ruleset.BaseRulesetDriver.raise_not_found)" class="language-python">	*bool*	Whether to raise an error if the ruleset is not found. Defaults to True.

Source Code in griptape/drivers/ruleset/base_ruleset_driver.py

```
@define
class BaseRulesetDriver(SerializableMixin, ABC):
    """Base class for ruleset drivers.

    Attributes:
        raise_not_found: Whether to raise an error if the ruleset is not found.␣
↪Defaults to True.
    """

    raise_not_found: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})

    @abstractmethod
    def load(self, ruleset_name: str) -> tuple[list[BaseRule], dict[str, Any]]: ...

    def _from_ruleset_dict(self, params_dict: dict[str, Any]) -> tuple[list[BaseRule],
↪ dict[str, Any]]:
        return [
            self._get_rule(rule["value"], rule.get("meta", {})) for rule in params_
↪dict.get("rules", [])
        ], params_dict.get("meta", {})

    def _get_rule(self, value: Any, meta: dict[str, Any]) -> BaseRule:
        from griptape.rules import JsonSchemaRule, Rule

        return JsonSchemaRule(value=value, meta=meta) if isinstance(value, dict) else␣
↪Rule(value=str(value), meta=meta)
```

- `raise_not_found = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_from_ruleset_dict(params_dict)

Source Code in griptape/drivers/ruleset/base_ruleset_driver.py

```
def _from_ruleset_dict(self, params_dict: dict[str, Any]) -> tuple[list[BaseRule],␣
↪dict[str, Any]]:
    return [
        self._get_rule(rule["value"], rule.get("meta", {})) for rule in params_dict.
↪get("rules", [])
    ], params_dict.get("meta", {})
```

_get_rule(value, meta)

Source Code in griptape/drivers/ruleset/base_ruleset_driver.py

```
def _get_rule(self, value: Any, meta: dict[str, Any]) -> BaseRule:
    from griptape.rules import JsonSchemaRule, Rule

    return JsonSchemaRule(value=value, meta=meta) if isinstance(value, dict) else␣
↪Rule(value=str(value), meta=meta)
```

load(ruleset_name)abstractmethod

Source Code in griptape/drivers/ruleset/base_ruleset_driver.py

```
@abstractmethod
def load(self, ruleset_name: str) -> tuple[list[BaseRule], dict[str, Any]]: ...
```

base_ruleset_driver

Bases: *SerializableMixin* “) , ABC

Attributes

Name	Type	Description
<a href="base_ruleset_driver.mdx#griptape.drivers.ruleset.base_ruleset_driver.BaseRulesetDriver.raise_not_found" title="raise_not_found = field(default=True, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.ruleset.base_ruleset_driver.BaseRulesetDriver.raise_not_found)">	*bool*	Whether to raise an error if the ruleset is not found. Defaults to True.

Source Code in griptape/drivers/ruleset/base_ruleset_driver.py

```
@define
class BaseRulesetDriver(SerializableMixin, ABC):
    """Base class for ruleset drivers.

    Attributes:
        raise_not_found: Whether to raise an error if the ruleset is not found.␣
↪Defaults to True.
    """

    raise_not_found: bool = field(default=True, kw_only=True, metadata={"serializable
↪": True})

    @abstractmethod
    def load(self, ruleset_name: str) -> tuple[list[BaseRule], dict[str, Any]]: ...

    def _from_ruleset_dict(self, params_dict: dict[str, Any]) -> tuple[list[BaseRule],
↪ dict[str, Any]]:
        return [
            self._get_rule(rule["value"], rule.get("meta", {})) for rule in params_
↪dict.get("rules", [])
        ], params_dict.get("meta", {})

    def _get_rule(self, value: Any, meta: dict[str, Any]) -> BaseRule:
        from griptape.rules import JsonSchemaRule, Rule

        return JsonSchemaRule(value=value, meta=meta) if isinstance(value, dict) else␣
↪Rule(value=str(value), meta=meta)
```

- `raise_not_found = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

_from_ruleset_dict(params_dict)

Source Code in griptape/drivers/ruleset/base_ruleset_driver.py

```
def _from_ruleset_dict(self, params_dict: dict[str, Any]) -> tuple[list[BaseRule],␣
↪dict[str, Any]]:
    return [
        self._get_rule(rule["value"], rule.get("meta", {})) for rule in params_dict.
↪get("rules", [])
    ], params_dict.get("meta", {})
```

_get_rule(value, meta)

Source Code in griptape/drivers/ruleset/base_ruleset_driver.py

```
def _get_rule(self, value: Any, meta: dict[str, Any]) -> BaseRule:
    from griptape.rules import JsonSchemaRule, Rule

    return JsonSchemaRule(value=value, meta=meta) if isinstance(value, dict) else␣
↪Rule(value=str(value), meta=meta)
```

load(ruleset_name)abstractmethod

Source Code in griptape/drivers/ruleset/base_ruleset_driver.py

```
@abstractmethod
def load(self, ruleset_name: str) -> tuple[list[BaseRule], dict[str, Any]]: ...
```

griptape_cloud

- `__all__ = ['GriptapeCloudRulesetDriver']` module-attribute

Bases: *BaseRulesetDriver* “)

Attributes

Name	Type	Description
<a href="griptape_cloud.mdx#griptape.drivers.ruleset.griptape_cloud.GriptapeCloudRulesetDriver.ruleset_id" title="ruleset_id = field(default=None, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.ruleset.griptape_cloud.GriptapeCloudRulesetDriver.ruleset_id)"><code class="language-python">	*Optional[str]*	The ID of the Thread to store the conversation memory in. If not provided, the driver will attempt to retrieve the ID from the environment variable *GT_CLOUD_THREAD_ID*. If that is not set, a new Thread will be created.
<a href="griptape_cloud.mdx#griptape.drivers.ruleset.griptape_cloud.GriptapeCloudRulesetDriver.base_url" title="base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_ 'https://cloud.griptape.ai'))) class-attribute instance-attribute (griptape.drivers.ruleset.griptape_cloud.GriptapeCloudRulesetDriver.base_url)"><code class="language-python">	*str*	The base URL of the Gen AI Builder API. Defaults to the value of the environment variable *GT_CLOUD_BASE_URL* or *https://cloud.griptape.ai*.

continues on next page

Table 212 – continued from previous page

Name	Type	Description
<a href="griptape_cloud.mdx#griptape.drivers.ruleset.griptape_cloud.GriptapeCloudRulesetDriver.api_key" title="api_key = field(default=Factory(lambda: os.getenv('GT_CLOUD_ class-attribute instance-attribute (griptape.drivers.ruleset.griptape_cloud.GriptapeCloudRulesetDriver.api_key)"><code class="language-python">	*Optional[str]*	The API key to use for authenticating with the Gen AI Builder API. If not provided, the driver will attempt to retrieve the API key from the environment variable *GT_CLOUD_API_KEY*.

Raises

Source Code in griptape/drivers/ruleset/griptape_cloud_ruleset_driver.py

```
@define(kw_only=True)
class GriptapeCloudRulesetDriver(BaseRulesetDriver):
    """A driver for storing conversation memory in the Gen AI Builder.

    Attributes:
        ruleset_id: The ID of the Thread to store the conversation memory in. If not␣
↪provided, the driver will attempt to
            retrieve the ID from the environment variable `GT_CLOUD_THREAD_ID`. If␣
↪that is not set, a new Thread will be
            created.
        base_url: The base URL of the Gen AI Builder API. Defaults to the value of␣
↪the environment variable
            `GT_CLOUD_BASE_URL` or `https://cloud.griptape.ai`.
        api_key: The API key to use for authenticating with the Gen AI Builder API.␣
↪If not provided, the driver will
            attempt to retrieve the API key from the environment variable `GT_CLOUD_
↪API_KEY`.

    Raises:
        ValueError: If `api_key` is not provided.
    """

    ruleset_id: Optional[str] = field(
        default=None,
        metadata={"serializable": True},
    )
    base_url: str = field(
```

(continues on next page)

(continued from previous page)

```
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: Optional[str] = field(default=Factory(lambda: os.getenv("GT_CLOUD_API_KEY
↪")))
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        init=False,
    )

    @api_key.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_api_key(self, _: Attribute, value: Optional[str]) -> str:
        if value is None:
            raise ValueError(f"{self.__class__.__name__} requires an API key")
        return value

    def load(self, ruleset_name: str) -> tuple[list[BaseRule], dict[str, Any]]:
        """Load the ruleset from Gen AI Builder, using the ruleset name as an alias␣
↪if ruleset_id is not provided."""
        ruleset = None

        if self.ruleset_id is not None:
            res = self._call_api("get", f"/rulesets/{self.ruleset_id}", raise_for_
↪status=False)
            if res.status_code == 200:
                ruleset = res.json()

        # use name as alias to get ruleset
        if ruleset is None:
            res = self._call_api("get", f"/rulesets?alias={ruleset_name}").json()
            if res.get("rulesets"):
                ruleset = res["rulesets"][0]

        # no ruleset by name or ruleset_id
        if ruleset is None:
            if self.raise_not_found:
                raise ValueError(f"No ruleset found with alias: {ruleset_name} or␣
↪ruleset_id: {self.ruleset_id}")
            return [], {}

        rules = self._call_api("get", f"/rules?ruleset_id={ruleset[ruleset_id]}").
↪json().get("rules", [])

        for rule in rules:
            rule["metadata"] = dict_merge(rule.get("metadata", {}), {"griptape_cloud_
↪rule_id": rule["rule_id"]})

        return [self._get_rule(rule["rule"], rule["metadata"]) for rule in rules],␣
↪ruleset.get("metadata", {})
```

(continues on next page)

(continued from previous page)

```
    def _get_url(self, path: str) -> str:
        path = path.lstrip("/")
        return griptape_cloud_url(self.base_url, f"api/{path}")

    def _call_api(self, method: str, path: str, *, raise_for_status: bool = True) ->␣
↪requests.Response:
        res = requests.request(method, self._get_url(path), headers=self.headers)
        if raise_for_status:
            res.raise_for_status()
        return res
```

- `api_key = field(default=Factory(lambda: os.getenv('GT_CLOUD_API_KEY')))` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')))` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), init=False)` class-attribute instance-attribute
- `ruleset_id = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

_call_api(method, path, *, raise_for_status=True)

Source Code in griptape/drivers/ruleset/griptape_cloud_ruleset_driver.py

```
def _call_api(self, method: str, path: str, *, raise_for_status: bool = True) ->␣
↪requests.Response:
    res = requests.request(method, self._get_url(path), headers=self.headers)
    if raise_for_status:
        res.raise_for_status()
    return res
```

_get_url(path)

Source Code in griptape/drivers/ruleset/griptape_cloud_ruleset_driver.py

```
def _get_url(self, path: str) -> str:
    path = path.lstrip("/")
    return griptape_cloud_url(self.base_url, f"api/{path}")
```

load(ruleset_name)

Source Code in griptape/drivers/ruleset/griptape_cloud_ruleset_driver.py

```
def load(self, ruleset_name: str) -> tuple[list[BaseRule], dict[str, Any]]:
    """Load the ruleset from Gen AI Builder, using the ruleset name as an alias if␣
↪ruleset_id is not provided."""
    ruleset = None

    if self.ruleset_id is not None:
        res = self._call_api("get", f"/rulesets/{self.ruleset_id}", raise_for_
```

(continues on next page)

(continued from previous page)

```
↪status=False)
        if res.status_code == 200:
            ruleset = res.json()

    # use name as alias to get ruleset
    if ruleset is None:
        res = self._call_api("get", f"/rulesets?alias={ruleset_name}").json()
        if res.get("rulesets"):
            ruleset = res["rulesets"][0]

    # no ruleset by name or ruleset_id
    if ruleset is None:
        if self.raise_not_found:
            raise ValueError(f"No ruleset found with alias: {ruleset_name} or ruleset_
↪id: {self.ruleset_id}")
        return [], {}

    rules = self._call_api("get", f"/rules?ruleset_id={ruleset[ruleset_id]}").json().
↪get("rules", [])

    for rule in rules:
        rule["metadata"] = dict_merge(rule.get("metadata", {}), {"griptape_cloud_rule_
↪id": rule["rule_id"]})

    return [self._get_rule(rule["rule"], rule["metadata"]) for rule in rules],␣
↪ruleset.get("metadata", {})
```

validate*api_key(*, value)

Source Code in griptape/drivers/ruleset/griptape_cloud_ruleset_driver.py

```
@api_key.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_api_key(self, _: Attribute, value: Optional[str]) -> str:
    if value is None:
        raise ValueError(f"{self.__class__.__name__} requires an API key")
    return value
```

griptape_cloud_ruleset_driver

Bases: *BaseRulesetDriver* “)

Attributes

Name	Type	Description
<a href="griptape_cloud_ruleset_driver.mdx#griptape.drivers.ruleset.griptape_cloud_ruleset_driver.GriptapeCloudRulesetDriver.ruleset_id" title="ruleset_id = field(default=None, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.ruleset.griptape_cloud_ruleset_driver.GriptapeCloudRulesetDriver.ruleset_id)"><code class="language-python">	*Optional[str]*	The ID of the Thread to store the conversation memory in. If not provided, the driver will attempt to retrieve the ID from the environment variable *GT_CLOUD_THREAD_ID*. If that is not set, a new Thread will be created.
<a href="griptape_cloud_ruleset_driver.mdx#griptape.drivers.ruleset.griptape_cloud_ruleset_driver.GriptapeCloudRulesetDriver.base_url" title="base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_ 'https://cloud.griptape.ai'))) class-attribute instance-attribute (griptape.drivers.ruleset.griptape_cloud_ruleset_driver.GriptapeCloudRulesetDriver.base_url)"><code class="language-python">	*str*	The base URL of the Gen AI Builder API. Defaults to the value of the environment variable *GT_CLOUD_BASE_URL* or *https://cloud.griptape.ai*.

continues on next page

Table 213 – continued from previous page

Name	Type	Description
<a href="griptape_cloud_ruleset_driver.mdx#griptape.drivers.ruleset.griptape_cloud_ruleset_driver.GriptapeCloudRulesetDriver.api_key" title="api_key = field(default=Factory(lambda: os.getenv('GT_CLOUD_ class-attribute instance-attribute (griptape.drivers.ruleset.griptape_cloud_ruleset_driver.GriptapeCloudRulesetDriver.api_key)"><code class="language-python">	*Optional[str]*	The API key to use for authenticating with the Gen AI Builder API. If not provided, the driver will attempt to retrieve the API key from the environment variable *GT_CLOUD_API_KEY*.

Raises

Source Code in griptape/drivers/ruleset/griptape_cloud_ruleset_driver.py

```
@define(kw_only=True)
class GriptapeCloudRulesetDriver(BaseRulesetDriver):
    """A driver for storing conversation memory in the Gen AI Builder.

    Attributes:
        ruleset_id: The ID of the Thread to store the conversation memory in. If not␣
↪provided, the driver will attempt to
            retrieve the ID from the environment variable `GT_CLOUD_THREAD_ID`. If␣
↪that is not set, a new Thread will be
            created.
        base_url: The base URL of the Gen AI Builder API. Defaults to the value of␣
↪the environment variable
            `GT_CLOUD_BASE_URL` or `https://cloud.griptape.ai`.
        api_key: The API key to use for authenticating with the Gen AI Builder API.␣
↪If not provided, the driver will
            attempt to retrieve the API key from the environment variable `GT_CLOUD_
↪API_KEY`.

    Raises:
        ValueError: If `api_key` is not provided.
    """

    ruleset_id: Optional[str] = field(
        default=None,
```

(continues on next page)

(continued from previous page)

```
        metadata={"serializable": True},
    )
    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: Optional[str] = field(default=Factory(lambda: os.getenv("GT_CLOUD_API_KEY
↪")))
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        init=False,
    )

    @api_key.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_api_key(self, _: Attribute, value: Optional[str]) -> str:
        if value is None:
            raise ValueError(f"{self.__class__.__name__} requires an API key")
        return value

    def load(self, ruleset_name: str) -> tuple[list[BaseRule], dict[str, Any]]:
        """Load the ruleset from Gen AI Builder, using the ruleset name as an alias␣
↪if ruleset_id is not provided."""
        ruleset = None

        if self.ruleset_id is not None:
            res = self._call_api("get", f"/rulesets/{self.ruleset_id}", raise_for_
↪status=False)
            if res.status_code == 200:
                ruleset = res.json()

        # use name as alias to get ruleset
        if ruleset is None:
            res = self._call_api("get", f"/rulesets?alias={ruleset_name}").json()
            if res.get("rulesets"):
                ruleset = res["rulesets"][0]

        # no ruleset by name or ruleset_id
        if ruleset is None:
            if self.raise_not_found:
                raise ValueError(f"No ruleset found with alias: {ruleset_name} or␣
↪ruleset_id: {self.ruleset_id}")
            return [], {}

        rules = self._call_api("get", f"/rules?ruleset_id={ruleset[ruleset_id]}").
↪json().get("rules", [])

        for rule in rules:
            rule["metadata"] = dict_merge(rule.get("metadata", {}), {"griptape_cloud_
↪rule_id": rule["rule_id"]})
```

(continues on next page)

(continued from previous page)

```
        return [self._get_rule(rule["rule"], rule["metadata"]) for rule in rules],␣
↪ruleset.get("metadata", {})

    def _get_url(self, path: str) -> str:
        path = path.lstrip("/")
        return griptape_cloud_url(self.base_url, f"api/{path}")

    def _call_api(self, method: str, path: str, *, raise_for_status: bool = True) ->␣
↪requests.Response:
        res = requests.request(method, self._get_url(path), headers=self.headers)
        if raise_for_status:
            res.raise_for_status()
        return res
```

- `api_key = field(default=Factory(lambda: os.getenv('GT_CLOUD_API_KEY')))` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')))` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), init=False)` class-attribute instance-attribute
- `ruleset_id = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

_call_api(method, path, *, raise_for_status=True)

Source Code in griptape/drivers/ruleset/griptape_cloud_ruleset_driver.py

```
def _call_api(self, method: str, path: str, *, raise_for_status: bool = True) ->␣
↪requests.Response:
    res = requests.request(method, self._get_url(path), headers=self.headers)
    if raise_for_status:
        res.raise_for_status()
    return res
```

_get_url(path)

Source Code in griptape/drivers/ruleset/griptape_cloud_ruleset_driver.py

```
def _get_url(self, path: str) -> str:
    path = path.lstrip("/")
    return griptape_cloud_url(self.base_url, f"api/{path}")
```

load(ruleset_name)

Source Code in griptape/drivers/ruleset/griptape_cloud_ruleset_driver.py

```
def load(self, ruleset_name: str) -> tuple[list[BaseRule], dict[str, Any]]:
    """Load the ruleset from Gen AI Builder, using the ruleset name as an alias if␣
↪ruleset_id is not provided."""
    ruleset = None
```

(continues on next page)

(continued from previous page)

```
    if self.ruleset_id is not None:
        res = self._call_api("get", f"/rulesets/{self.ruleset_id}", raise_for_
↪status=False)
        if res.status_code == 200:
            ruleset = res.json()

    # use name as alias to get ruleset
    if ruleset is None:
        res = self._call_api("get", f"/rulesets?alias={ruleset_name}").json()
        if res.get("rulesets"):
            ruleset = res["rulesets"][0]

    # no ruleset by name or ruleset_id
    if ruleset is None:
        if self.raise_not_found:
            raise ValueError(f"No ruleset found with alias: {ruleset_name} or ruleset_
↪id: {self.ruleset_id}")
        return [], {}

    rules = self._call_api("get", f"/rules?ruleset_id={ruleset[ruleset_id]}").json().
↪get("rules", [])

    for rule in rules:
        rule["metadata"] = dict_merge(rule.get("metadata", {}), {"griptape_cloud_rule_
↪id": rule["rule_id"]})

    return [self._get_rule(rule["rule"], rule["metadata"]) for rule in rules],␣
↪ruleset.get("metadata", {})
```

validate*api_key(*, value)

Source Code in griptape/drivers/ruleset/griptape_cloud_ruleset_driver.py

```
@api_key.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_api_key(self, _: Attribute, value: Optional[str]) -> str:
    if value is None:
        raise ValueError(f"{self.__class__.__name__} requires an API key")
    return value
```

local

- `__all__ = ['LocalRulesetDriver']` module-attribute

Bases: *BaseRulesetDriver* “)

Source Code in griptape/drivers/ruleset/local_ruleset_driver.py

```
@define(kw_only=True)
class LocalRulesetDriver(BaseRulesetDriver):
    persist_dir: Optional[str] = field(default=None, metadata={"serializable": True})
```

(continues on next page)

(continued from previous page)

```
    def load(self, ruleset_name: str) -> tuple[list[BaseRule], dict[str, Any]]:
        if self.persist_dir is None:
            return [], {}

        file_name = os.path.join(self.persist_dir, ruleset_name)

        if (
            file_name is not None
            and os.path.exists(file_name)
            and (loaded_str := Path(file_name).read_text()) is not None
        ):
            try:
                return self._from_ruleset_dict(json.loads(loaded_str))
            except Exception as e:
                raise ValueError(f"Unable to load data from {file_name}") from e

        if self.raise_not_found:
            raise ValueError(f"Ruleset not found with name {file_name}")
        return [], {}
```

- `persist_dir = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

load(ruleset_name)

Source Code in griptape/drivers/ruleset/local_ruleset_driver.py

```
def load(self, ruleset_name: str) -> tuple[list[BaseRule], dict[str, Any]]:
    if self.persist_dir is None:
        return [], {}

    file_name = os.path.join(self.persist_dir, ruleset_name)

    if (
        file_name is not None
        and os.path.exists(file_name)
        and (loaded_str := Path(file_name).read_text()) is not None
    ):
        try:
            return self._from_ruleset_dict(json.loads(loaded_str))
        except Exception as e:
            raise ValueError(f"Unable to load data from {file_name}") from e

    if self.raise_not_found:
        raise ValueError(f"Ruleset not found with name {file_name}")
    return [], {}
```

local_ruleset_driver

Bases: *BaseRulesetDriver* “)

Source Code in griptape/drivers/ruleset/local_ruleset_driver.py

```
@define(kw_only=True)
class LocalRulesetDriver(BaseRulesetDriver):
    persist_dir: Optional[str] = field(default=None, metadata={"serializable": True})

    def load(self, ruleset_name: str) -> tuple[list[BaseRule], dict[str, Any]]:
        if self.persist_dir is None:
            return [], {}

        file_name = os.path.join(self.persist_dir, ruleset_name)

        if (
            file_name is not None
            and os.path.exists(file_name)
            and (loaded_str := Path(file_name).read_text()) is not None
        ):
            try:
                return self._from_ruleset_dict(json.loads(loaded_str))
            except Exception as e:
                raise ValueError(f"Unable to load data from {file_name}") from e

        if self.raise_not_found:
            raise ValueError(f"Ruleset not found with name {file_name}")
        return [], {}
```

- `persist_dir = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

load(ruleset_name)

Source Code in griptape/drivers/ruleset/local_ruleset_driver.py

```
def load(self, ruleset_name: str) -> tuple[list[BaseRule], dict[str, Any]]:
    if self.persist_dir is None:
        return [], {}

    file_name = os.path.join(self.persist_dir, ruleset_name)

    if (
        file_name is not None
        and os.path.exists(file_name)
        and (loaded_str := Path(file_name).read_text()) is not None
    ):
        try:
            return self._from_ruleset_dict(json.loads(loaded_str))
        except Exception as e:
            raise ValueError(f"Unable to load data from {file_name}") from e

    if self.raise_not_found:
```

(continues on next page)

(continued from previous page)

```
        raise ValueError(f"Ruleset not found with name {file_name}")
    return [], {}
```

sql

- `__all__ = ['BaseSqlDriver', 'SqlDriver']` module-attribute

Bases:

ABC

Source Code in griptape/drivers/sql/base_sql_driver.py

```
@define
class BaseSqlDriver(ABC):
    @dataclass
    class RowResult:
        cells: dict[str, Any]

    @abstractmethod
    def execute_query(self, query: str) -> Optional[list[RowResult]]: ...

    @abstractmethod
    def execute_query_raw(self, query: str) -> Optional[list[dict[str, Any]]]: ...

    @abstractmethod
    def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]: ...
```

RowResultdataclass

Source Code in griptape/drivers/sql/base_sql_driver.py

```
@dataclass
class RowResult:
    cells: dict[str, Any]
```

- `cells` instance-attribute

execute_query(query)abstractmethod

Source Code in griptape/drivers/sql/base_sql_driver.py

```
@abstractmethod
def execute_query(self, query: str) -> Optional[list[RowResult]]: ...
```

execute_query_raw(query)abstractmethod

Source Code in griptape/drivers/sql/base_sql_driver.py

```
@abstractmethod
def execute_query_raw(self, query: str) -> Optional[list[dict[str, Any]]]: ...
```

get_table_schema(table_name, schema=None)abstractmethod

Source Code in griptape/drivers/sql/base_sql_driver.py

```
@abstractmethod
def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]: ...
```

SqlDriver

Bases: *BaseSqlDriver* “)

Source Code in griptape/drivers/sql/sql_driver.py

```
@define
class SqlDriver(BaseSqlDriver):
    engine_url: str = field(kw_only=True)
    create_engine_params: dict = field(factory=dict, kw_only=True)
    _engine: Optional[Engine] = field(default=None, kw_only=True, alias="engine",␣
↪metadata={"serializable": False})

    @lazy_property()
    def engine(self) -> Engine:
        return import_optional_dependency("sqlalchemy").create_engine(self.engine_url,
↪ **self.create_engine_params)

    def execute_query(self, query: str) -> Optional[list[BaseSqlDriver.RowResult]]:
        rows = self.execute_query_raw(query)

        if rows:
            return [BaseSqlDriver.RowResult(row) for row in rows]
        return None

    def execute_query_raw(self, query: str) -> Optional[list[dict[str,␣
↪Optional[Any]]]]:
        sqlalchemy = import_optional_dependency("sqlalchemy")

        with self.engine.connect() as con:
            results = con.execute(sqlalchemy.text(query))

            if results is not None:
                if results.returns_rows:
                    return [dict(result._mapping) for result in results]
                con.commit()
                return None
            raise ValueError("No result found")

    def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]:
        sqlalchemy_exc = import_optional_dependency("sqlalchemy.exc")

        try:
            return str(SqlDriver._get_table_schema(self.engine, table_name, schema))
        except sqlalchemy_exc.NoSuchTableError:
```

(continues on next page)

(continued from previous page)

```
            return None

    @staticmethod
    @lru_cache
    def _get_table_schema(
        engine: Engine, table_name: str, schema: Optional[str] = None
    ) -> Optional[list[tuple[str, str]]]:
        sqlalchemy = import_optional_dependency("sqlalchemy")

        return [(col["name"], col["type"]) for col in sqlalchemy.inspect(engine).get_
↪columns(table_name, schema=schema)]
```

- `_engine = field(default=None, kw_only=True, alias='engine', metadata={'serializable': False})` class-attribute instance-attribute
- `create_engine_params = field(factory=dict, kw_only=True)` class-attribute instance-attribute
- `engine_url = field(kw_only=True)` class-attribute instance-attribute

_get_table_schema(engine, table_name, schema=None)cachedstaticmethod

Source Code in griptape/drivers/sql/sql_driver.py

```
@staticmethod
@lru_cache
def _get_table_schema(
    engine: Engine, table_name: str, schema: Optional[str] = None
) -> Optional[list[tuple[str, str]]]:
    sqlalchemy = import_optional_dependency("sqlalchemy")

    return [(col["name"], col["type"]) for col in sqlalchemy.inspect(engine).get_
↪columns(table_name, schema=schema)]
```

engine()

Source Code in griptape/drivers/sql/sql_driver.py

```
@lazy_property()
def engine(self) -> Engine:
    return import_optional_dependency("sqlalchemy").create_engine(self.engine_url,␣
↪**self.create_engine_params)
```

execute_query(query)

Source Code in griptape/drivers/sql/sql_driver.py

```
def execute_query(self, query: str) -> Optional[list[BaseSqlDriver.RowResult]]:
    rows = self.execute_query_raw(query)

    if rows:
        return [BaseSqlDriver.RowResult(row) for row in rows]
    return None
```

execute_query_raw(query)

Source Code in griptape/drivers/sql/sql_driver.py

```
def execute_query_raw(self, query: str) -> Optional[list[dict[str, Optional[Any]]]]:
    sqlalchemy = import_optional_dependency("sqlalchemy")

    with self.engine.connect() as con:
        results = con.execute(sqlalchemy.text(query))

        if results is not None:
            if results.returns_rows:
                return [dict(result._mapping) for result in results]
            con.commit()
            return None
        raise ValueError("No result found")
```

get_table_schema(table_name, schema=None)

Source Code in griptape/drivers/sql/sql_driver.py

```
def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]:
    sqlalchemy_exc = import_optional_dependency("sqlalchemy.exc")

    try:
        return str(SqlDriver._get_table_schema(self.engine, table_name, schema))
    except sqlalchemy_exc.NoSuchTableError:
        return None
```

amazon_redshift

- `__all__ = ['AmazonRedshiftSqlDriver']` module-attribute

Bases: *BaseSqlDriver* “)

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@define
class AmazonRedshiftSqlDriver(BaseSqlDriver):
    database: str = field(kw_only=True)
    session: boto3.Session = field(kw_only=True)
    cluster_identifier: Optional[str] = field(default=None, kw_only=True)
    workgroup_name: Optional[str] = field(default=None, kw_only=True)
    db_user: Optional[str] = field(default=None, kw_only=True)
    database_credentials_secret_arn: Optional[str] = field(default=None, kw_only=True)
    wait_for_query_completion_sec: float = field(default=0.3, kw_only=True)
    _client: Optional[RedshiftDataAPIServiceClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> RedshiftDataAPIServiceClient:
        return self.session.client("redshift-data")
```

(continues on next page)

(continued from previous page)

```
    @workgroup_name.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_params(self, _: Attribute, workgroup_name: Optional[str]) -> None:
        if not self.cluster_identifier and not self.workgroup_name:
            raise ValueError("Provide a value for one of `cluster_identifier` or␣
↪`workgroup_name`")
        if self.cluster_identifier and self.workgroup_name:
            raise ValueError("Provide a value for either `cluster_identifier` or␣
↪`workgroup_name`, but not both")

    @classmethod
    def _process_rows_from_records(cls, records: list) -> list[list]:
        return [[c[list(c.keys())[0]] for c in r] for r in records]

    @classmethod
    def _process_cells_from_rows_and_columns(cls, columns: list, rows: list[list]) ->␣
↪list[dict[str, Any]]:
        return [{column: r[idx] for idx, column in enumerate(columns)} for r in rows]

    @classmethod
    def _process_columns_from_column_metadata(cls, meta: dict) -> list:
        return [k["name"] for k in meta]

    @classmethod
    def _post_process(cls, meta: dict, records: list) -> list[dict[str, Any]]:
        columns = cls._process_columns_from_column_metadata(meta)
        rows = cls._process_rows_from_records(records)
        return cls._process_cells_from_rows_and_columns(columns, rows)

    def execute_query(self, query: str) -> Optional[list[BaseSqlDriver.RowResult]]:
        rows = self.execute_query_raw(query)
        if rows:
            return [BaseSqlDriver.RowResult(row) for row in rows]
        return None

    def execute_query_raw(self, query: str) -> Optional[list[dict[str,␣
↪Optional[Any]]]]:
        function_kwargs = {"Sql": query, "Database": self.database}
        if self.workgroup_name:
            function_kwargs["WorkgroupName"] = self.workgroup_name
        if self.cluster_identifier:
            function_kwargs["ClusterIdentifier"] = self.cluster_identifier
        if self.db_user:
            function_kwargs["DbUser"] = self.db_user
        if self.database_credentials_secret_arn:
            function_kwargs["SecretArn"] = self.database_credentials_secret_arn

        response = self.client.execute_statement(**function_kwargs)  # pyright:␣
↪ignore[reportArgumentType]
        response_id = response["Id"]
```

(continues on next page)

(continued from previous page)

```
        statement = self.client.describe_statement(Id=response_id)

        while statement["Status"] in ["SUBMITTED", "PICKED", "STARTED"]:
            time.sleep(self.wait_for_query_completion_sec)
            statement = self.client.describe_statement(Id=response_id)

        if statement["Status"] == "FINISHED":
            statement_result = self.client.get_statement_result(Id=response_id)
            results = statement_result.get("Records", [])

            while "NextToken" in statement_result:
                statement_result = self.client.get_statement_result(
                    Id=response_id,
                    NextToken=statement_result["NextToken"],
                )
                results = results + response.get("Records", [])

            return self._post_process(statement_result["ColumnMetadata"], results)  #␣
↪pyright: ignore[reportArgumentType]

        if statement["Status"] in ["FAILED", "ABORTED"]:
            return None
        return None

    def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]:
        function_kwargs = {"Database": self.database, "Table": table_name}
        if schema:
            function_kwargs["Schema"] = schema
        if self.workgroup_name:
            function_kwargs["WorkgroupName"] = self.workgroup_name
        if self.cluster_identifier:
            function_kwargs["ClusterIdentifier"] = self.cluster_identifier
        if self.db_user:
            function_kwargs["DbUser"] = self.db_user
        if self.database_credentials_secret_arn:
            function_kwargs["SecretArn"] = self.database_credentials_secret_arn
        response = self.client.describe_table(**function_kwargs)  # pyright:␣
↪ignore[reportArgumentType]
        return str([col["name"] for col in response["ColumnList"] if "name" in col])
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `cluster_identifier = field(default=None, kw_only=True)` class-attribute instance-attribute
- `database = field(kw_only=True)` class-attribute instance-attribute
- `database_credentials_secret_arn = field(default=None, kw_only=True)` class-attribute instance-attribute
- `db_user = field(default=None, kw_only=True)` class-attribute instance-attribute
- `session = field(kw_only=True)` class-attribute instance-attribute
- `wait_for_query_completion_sec = field(default=0.3, kw_only=True)` class-attribute

instance-attribute

- `workgroup_name = field(default=None, kw_only=True)` class-attribute instance-attribute

_post_process(meta, records)classmethod

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@classmethod
def _post_process(cls, meta: dict, records: list) -> list[dict[str, Any]]:
    columns = cls._process_columns_from_column_metadata(meta)
    rows = cls._process_rows_from_records(records)
    return cls._process_cells_from_rows_and_columns(columns, rows)
```

_process_cells_from_rows_and_columns(columns, rows)classmethod

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@classmethod
def _process_cells_from_rows_and_columns(cls, columns: list, rows: list[list]) ->␣
↪list[dict[str, Any]]:
    return [{column: r[idx] for idx, column in enumerate(columns)} for r in rows]
```

_process_columns_from_column_metadata(meta)classmethod

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@classmethod
def _process_columns_from_column_metadata(cls, meta: dict) -> list:
    return [k["name"] for k in meta]
```

_process_rows_from_records(records)classmethod

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@classmethod
def _process_rows_from_records(cls, records: list) -> list[list]:
    return [[c[list(c.keys())[0]] for c in r] for r in records]
```

client()

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@lazy_property()
def client(self) -> RedshiftDataAPIServiceClient:
    return self.session.client("redshift-data")
```

execute_query(query)

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
def execute_query(self, query: str) -> Optional[list[BaseSqlDriver.RowResult]]:
    rows = self.execute_query_raw(query)
    if rows:
        return [BaseSqlDriver.RowResult(row) for row in rows]
    return None
```

execute_query_raw(query)

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
def execute_query_raw(self, query: str) -> Optional[list[dict[str, Optional[Any]]]]:
    function_kwargs = {"Sql": query, "Database": self.database}
    if self.workgroup_name:
        function_kwargs["WorkgroupName"] = self.workgroup_name
    if self.cluster_identifier:
        function_kwargs["ClusterIdentifier"] = self.cluster_identifier
    if self.db_user:
        function_kwargs["DbUser"] = self.db_user
    if self.database_credentials_secret_arn:
        function_kwargs["SecretArn"] = self.database_credentials_secret_arn

    response = self.client.execute_statement(**function_kwargs)  # pyright:␣
↪ignore[reportArgumentType]
    response_id = response["Id"]

    statement = self.client.describe_statement(Id=response_id)

    while statement["Status"] in ["SUBMITTED", "PICKED", "STARTED"]:
        time.sleep(self.wait_for_query_completion_sec)
        statement = self.client.describe_statement(Id=response_id)

    if statement["Status"] == "FINISHED":
        statement_result = self.client.get_statement_result(Id=response_id)
        results = statement_result.get("Records", [])

        while "NextToken" in statement_result:
            statement_result = self.client.get_statement_result(
                Id=response_id,
                NextToken=statement_result["NextToken"],
            )
            results = results + response.get("Records", [])

        return self._post_process(statement_result["ColumnMetadata"], results)  #␣
↪pyright: ignore[reportArgumentType]

    if statement["Status"] in ["FAILED", "ABORTED"]:
        return None
    return None
```

get_table_schema(table_name, schema=None)

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]:
    function_kwargs = {"Database": self.database, "Table": table_name}
    if schema:
        function_kwargs["Schema"] = schema
    if self.workgroup_name:
        function_kwargs["WorkgroupName"] = self.workgroup_name
    if self.cluster_identifier:
        function_kwargs["ClusterIdentifier"] = self.cluster_identifier
    if self.db_user:
        function_kwargs["DbUser"] = self.db_user
    if self.database_credentials_secret_arn:
        function_kwargs["SecretArn"] = self.database_credentials_secret_arn
    response = self.client.describe_table(**function_kwargs)  # pyright:␣
↪ignore[reportArgumentType]
    return str([col["name"] for col in response["ColumnList"] if "name" in col])
```

validate*params(*, workgroup_name)

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@workgroup_name.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_params(self, _: Attribute, workgroup_name: Optional[str]) -> None:
    if not self.cluster_identifier and not self.workgroup_name:
        raise ValueError("Provide a value for one of `cluster_identifier` or␣
↪`workgroup_name`")
    if self.cluster_identifier and self.workgroup_name:
        raise ValueError("Provide a value for either `cluster_identifier` or␣
↪`workgroup_name`, but not both")
```

amazon_redshift_sql_driver

Bases: *BaseSqlDriver* “)

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@define
class AmazonRedshiftSqlDriver(BaseSqlDriver):
    database: str = field(kw_only=True)
    session: boto3.Session = field(kw_only=True)
    cluster_identifier: Optional[str] = field(default=None, kw_only=True)
    workgroup_name: Optional[str] = field(default=None, kw_only=True)
    db_user: Optional[str] = field(default=None, kw_only=True)
    database_credentials_secret_arn: Optional[str] = field(default=None, kw_only=True)
    wait_for_query_completion_sec: float = field(default=0.3, kw_only=True)
    _client: Optional[RedshiftDataAPIServiceClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )
```

(continues on next page)

(continued from previous page)

```
    @lazy_property()
    def client(self) -> RedshiftDataAPIServiceClient:
        return self.session.client("redshift-data")

    @workgroup_name.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_params(self, _: Attribute, workgroup_name: Optional[str]) -> None:
        if not self.cluster_identifier and not self.workgroup_name:
            raise ValueError("Provide a value for one of `cluster_identifier` or␣
↪`workgroup_name`")
        if self.cluster_identifier and self.workgroup_name:
            raise ValueError("Provide a value for either `cluster_identifier` or␣
↪`workgroup_name`, but not both")

    @classmethod
    def _process_rows_from_records(cls, records: list) -> list[list]:
        return [[c[list(c.keys())[0]] for c in r] for r in records]

    @classmethod
    def _process_cells_from_rows_and_columns(cls, columns: list, rows: list[list]) ->␣
↪list[dict[str, Any]]:
        return [{column: r[idx] for idx, column in enumerate(columns)} for r in rows]

    @classmethod
    def _process_columns_from_column_metadata(cls, meta: dict) -> list:
        return [k["name"] for k in meta]

    @classmethod
    def _post_process(cls, meta: dict, records: list) -> list[dict[str, Any]]:
        columns = cls._process_columns_from_column_metadata(meta)
        rows = cls._process_rows_from_records(records)
        return cls._process_cells_from_rows_and_columns(columns, rows)

    def execute_query(self, query: str) -> Optional[list[BaseSqlDriver.RowResult]]:
        rows = self.execute_query_raw(query)
        if rows:
            return [BaseSqlDriver.RowResult(row) for row in rows]
        return None

    def execute_query_raw(self, query: str) -> Optional[list[dict[str,␣
↪Optional[Any]]]]:
        function_kwargs = {"Sql": query, "Database": self.database}
        if self.workgroup_name:
            function_kwargs["WorkgroupName"] = self.workgroup_name
        if self.cluster_identifier:
            function_kwargs["ClusterIdentifier"] = self.cluster_identifier
        if self.db_user:
            function_kwargs["DbUser"] = self.db_user
        if self.database_credentials_secret_arn:
            function_kwargs["SecretArn"] = self.database_credentials_secret_arn
```

(continues on next page)

(continued from previous page)

```
        response = self.client.execute_statement(**function_kwargs)  # pyright:␣
↪ignore[reportArgumentType]
        response_id = response["Id"]

        statement = self.client.describe_statement(Id=response_id)

        while statement["Status"] in ["SUBMITTED", "PICKED", "STARTED"]:
            time.sleep(self.wait_for_query_completion_sec)
            statement = self.client.describe_statement(Id=response_id)

        if statement["Status"] == "FINISHED":
            statement_result = self.client.get_statement_result(Id=response_id)
            results = statement_result.get("Records", [])

            while "NextToken" in statement_result:
                statement_result = self.client.get_statement_result(
                    Id=response_id,
                    NextToken=statement_result["NextToken"],
                )
                results = results + response.get("Records", [])

            return self._post_process(statement_result["ColumnMetadata"], results)  #␣
↪pyright: ignore[reportArgumentType]

        if statement["Status"] in ["FAILED", "ABORTED"]:
            return None
        return None

    def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]:
        function_kwargs = {"Database": self.database, "Table": table_name}
        if schema:
            function_kwargs["Schema"] = schema
        if self.workgroup_name:
            function_kwargs["WorkgroupName"] = self.workgroup_name
        if self.cluster_identifier:
            function_kwargs["ClusterIdentifier"] = self.cluster_identifier
        if self.db_user:
            function_kwargs["DbUser"] = self.db_user
        if self.database_credentials_secret_arn:
            function_kwargs["SecretArn"] = self.database_credentials_secret_arn
        response = self.client.describe_table(**function_kwargs)  # pyright:␣
↪ignore[reportArgumentType]
        return str([col["name"] for col in response["ColumnList"] if "name" in col])
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `cluster_identifier = field(default=None, kw_only=True)` class-attribute instance-attribute
- `database = field(kw_only=True)` class-attribute instance-attribute
- `database_credentials_secret_arn = field(default=None, kw_only=True)` class-attribute instance-attribute

- `db_user = field(default=None, kw_only=True)` class-attribute instance-attribute
- `session = field(kw_only=True)` class-attribute instance-attribute
- `wait_for_query_completion_sec = field(default=0.3, kw_only=True)` class-attribute instance-attribute
- `workgroup_name = field(default=None, kw_only=True)` class-attribute instance-attribute

_post_process(meta, records)classmethod

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@classmethod
def _post_process(cls, meta: dict, records: list) -> list[dict[str, Any]]:
    columns = cls._process_columns_from_column_metadata(meta)
    rows = cls._process_rows_from_records(records)
    return cls._process_cells_from_rows_and_columns(columns, rows)
```

_process_cells_from_rows_and_columns(columns, rows)classmethod

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@classmethod
def _process_cells_from_rows_and_columns(cls, columns: list, rows: list[list]) ->␣
↪list[dict[str, Any]]:
    return [{column: r[idx] for idx, column in enumerate(columns)} for r in rows]
```

_process_columns_from_column_metadata(meta)classmethod

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@classmethod
def _process_columns_from_column_metadata(cls, meta: dict) -> list:
    return [k["name"] for k in meta]
```

_process_rows_from_records(records)classmethod

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@classmethod
def _process_rows_from_records(cls, records: list) -> list[list]:
    return [[c[list(c.keys())[0]] for c in r] for r in records]
```

client()

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@lazy_property()
def client(self) -> RedshiftDataAPIServiceClient:
    return self.session.client("redshift-data")
```

execute_query(query)

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
def execute_query(self, query: str) -> Optional[list[BaseSqlDriver.RowResult]]:
    rows = self.execute_query_raw(query)
    if rows:
        return [BaseSqlDriver.RowResult(row) for row in rows]
    return None
```

execute_query_raw(query)

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
def execute_query_raw(self, query: str) -> Optional[list[dict[str, Optional[Any]]]]:
    function_kwargs = {"Sql": query, "Database": self.database}
    if self.workgroup_name:
        function_kwargs["WorkgroupName"] = self.workgroup_name
    if self.cluster_identifier:
        function_kwargs["ClusterIdentifier"] = self.cluster_identifier
    if self.db_user:
        function_kwargs["DbUser"] = self.db_user
    if self.database_credentials_secret_arn:
        function_kwargs["SecretArn"] = self.database_credentials_secret_arn

    response = self.client.execute_statement(**function_kwargs)  # pyright:␣
↪ignore[reportArgumentType]
    response_id = response["Id"]

    statement = self.client.describe_statement(Id=response_id)

    while statement["Status"] in ["SUBMITTED", "PICKED", "STARTED"]:
        time.sleep(self.wait_for_query_completion_sec)
        statement = self.client.describe_statement(Id=response_id)

    if statement["Status"] == "FINISHED":
        statement_result = self.client.get_statement_result(Id=response_id)
        results = statement_result.get("Records", [])

        while "NextToken" in statement_result:
            statement_result = self.client.get_statement_result(
                Id=response_id,
                NextToken=statement_result["NextToken"],
            )
            results = results + response.get("Records", [])

        return self._post_process(statement_result["ColumnMetadata"], results)  #␣
↪pyright: ignore[reportArgumentType]

    if statement["Status"] in ["FAILED", "ABORTED"]:
        return None
    return None
```

get_table_schema(table_name, schema=None)

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]:
    function_kwargs = {"Database": self.database, "Table": table_name}
    if schema:
        function_kwargs["Schema"] = schema
    if self.workgroup_name:
        function_kwargs["WorkgroupName"] = self.workgroup_name
    if self.cluster_identifier:
        function_kwargs["ClusterIdentifier"] = self.cluster_identifier
    if self.db_user:
        function_kwargs["DbUser"] = self.db_user
    if self.database_credentials_secret_arn:
        function_kwargs["SecretArn"] = self.database_credentials_secret_arn
    response = self.client.describe_table(**function_kwargs)  # pyright:␣
↪ignore[reportArgumentType]
    return str([col["name"] for col in response["ColumnList"] if "name" in col])
```

validate*params(*, workgroup_name)

Source Code in griptape/drivers/sql/amazon_redshift_sql_driver.py

```
@workgroup_name.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_params(self, _: Attribute, workgroup_name: Optional[str]) -> None:
    if not self.cluster_identifier and not self.workgroup_name:
        raise ValueError("Provide a value for one of `cluster_identifier` or␣
↪`workgroup_name`")
    if self.cluster_identifier and self.workgroup_name:
        raise ValueError("Provide a value for either `cluster_identifier` or␣
↪`workgroup_name`, but not both")
```

base_sql_driver

Bases:

ABC

Source Code in griptape/drivers/sql/base_sql_driver.py

```
@define
class BaseSqlDriver(ABC):
    @dataclass
    class RowResult:
        cells: dict[str, Any]

    @abstractmethod
    def execute_query(self, query: str) -> Optional[list[RowResult]]: ...

    @abstractmethod
    def execute_query_raw(self, query: str) -> Optional[list[dict[str, Any]]]: ...
```

(continues on next page)

(continued from previous page)

```
    @abstractmethod
    def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]: ...
```

RowResultdataclass

Source Code in griptape/drivers/sql/base_sql_driver.py

```
@dataclass
class RowResult:
    cells: dict[str, Any]
```

- `cells` instance-attribute

execute_query(query)abstractmethod

Source Code in griptape/drivers/sql/base_sql_driver.py

```
@abstractmethod
def execute_query(self, query: str) -> Optional[list[RowResult]]: ...
```

execute_query_raw(query)abstractmethod

Source Code in griptape/drivers/sql/base_sql_driver.py

```
@abstractmethod
def execute_query_raw(self, query: str) -> Optional[list[dict[str, Any]]]: ...
```

get_table_schema(table_name, schema=None)abstractmethod

Source Code in griptape/drivers/sql/base_sql_driver.py

```
@abstractmethod
def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]: ...
```

snowflake

- `__all__ = ['SnowflakeSqlDriver']` module-attribute

Bases: *BaseSqlDriver* “)

Source Code in griptape/drivers/sql/snowflake_sql_driver.py

```
@define
class SnowflakeSqlDriver(BaseSqlDriver):
    get_connection: Callable[[], SnowflakeConnection] = field(kw_only=True)
    _engine: Optional[Engine] = field(default=None, kw_only=True, alias="engine",␣
↪metadata={"serializable": False})

    @get_connection.validator  # pyright: ignore[reportFunctionMemberAccess]
```

(continues on next page)

(continued from previous page)

```
    def validate_get_connection(self, _: Attribute, get_connection: Callable[[],␣
↪SnowflakeConnection]) -> None:
        snowflake_connection = get_connection()
        snowflake = import_optional_dependency("snowflake")

        if not isinstance(snowflake_connection, snowflake.connector.
↪SnowflakeConnection):
            raise ValueError("The get_connection function must return a␣
↪SnowflakeConnection")
        if not snowflake_connection.schema or not snowflake_connection.database:
            raise ValueError("Provide a schema and database for the Snowflake␣
↪connection")

    @lazy_property()
    def engine(self) -> Engine:
        return import_optional_dependency("sqlalchemy").create_engine(
            "snowflake://not@used/db",
            creator=self.get_connection,
        )

    def execute_query(self, query: str) -> Optional[list[BaseSqlDriver.RowResult]]:
        rows = self.execute_query_raw(query)

        if rows:
            return [BaseSqlDriver.RowResult(row) for row in rows]
        return None

    def execute_query_raw(self, query: str) -> Optional[list[dict[str, Any]]]:
        sqlalchemy = import_optional_dependency("sqlalchemy")

        with self.engine.connect() as con:
            results = con.execute(sqlalchemy.text(query))

            if results is not None:
                if results.returns_rows:
                    return [dict(result._mapping) for result in results]
                return None
            raise ValueError("No results found")

    def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]:
        sqlalchemy = import_optional_dependency("sqlalchemy")

        try:
            metadata_obj = sqlalchemy.MetaData()
            metadata_obj.reflect(bind=self.engine)
            table = sqlalchemy.Table(table_name, metadata_obj, schema=schema,␣
↪autoload=True, autoload_with=self.engine)
            return str([(c.name, c.type) for c in table.columns])
        except sqlalchemy.exc.NoSuchTableError:
            return None
```

- `_engine = field(default=None, kw_only=True, alias='engine',`

`metadata={'serializable': False})` class-attribute instance-attribute

- `get_connection = field(kw_only=True)` class-attribute instance-attribute

engine()

Source Code in griptape/drivers/sql/snowflake_sql_driver.py

```
@lazy_property()
def engine(self) -> Engine:
    return import_optional_dependency("sqlalchemy").create_engine(
        "snowflake://not@used/db",
        creator=self.get_connection,
    )
```

execute_query(query)

Source Code in griptape/drivers/sql/snowflake_sql_driver.py

```
def execute_query(self, query: str) -> Optional[list[BaseSqlDriver.RowResult]]:
    rows = self.execute_query_raw(query)

    if rows:
        return [BaseSqlDriver.RowResult(row) for row in rows]
    return None
```

execute_query_raw(query)

Source Code in griptape/drivers/sql/snowflake_sql_driver.py

```
def execute_query_raw(self, query: str) -> Optional[list[dict[str, Any]]]:
    sqlalchemy = import_optional_dependency("sqlalchemy")

    with self.engine.connect() as con:
        results = con.execute(sqlalchemy.text(query))

        if results is not None:
            if results.returns_rows:
                return [dict(result._mapping) for result in results]
            return None
        raise ValueError("No results found")
```

get_table_schema(table_name, schema=None)

Source Code in griptape/drivers/sql/snowflake_sql_driver.py

```
def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]:
    sqlalchemy = import_optional_dependency("sqlalchemy")

    try:
        metadata_obj = sqlalchemy.MetaData()
        metadata_obj.reflect(bind=self.engine)
```

(continues on next page)

(continued from previous page)

```
        table = sqlalchemy.Table(table_name, metadata_obj, schema=schema,␣
↪autoload=True, autoload_with=self.engine)
        return str([(c.name, c.type) for c in table.columns])
    except sqlalchemy.exc.NoSuchTableError:
        return None
```

validate*get_connection(*, get_connection)

Source Code in griptape/drivers/sql/snowflake_sql_driver.py

```
@get_connection.validator  # pyright: ignore[reportFunctionMemberAccess]
def validate_get_connection(self, _: Attribute, get_connection: Callable[[],␣
↪SnowflakeConnection]) -> None:
    snowflake_connection = get_connection()
    snowflake = import_optional_dependency("snowflake")

    if not isinstance(snowflake_connection, snowflake.connector.SnowflakeConnection):
        raise ValueError("The get_connection function must return a␣
↪SnowflakeConnection")
    if not snowflake_connection.schema or not snowflake_connection.database:
        raise ValueError("Provide a schema and database for the Snowflake connection")
```

snowflake_sql_driver

Bases: *BaseSqlDriver* “)

Source Code in griptape/drivers/sql/snowflake_sql_driver.py

```
@define
class SnowflakeSqlDriver(BaseSqlDriver):
    get_connection: Callable[[], SnowflakeConnection] = field(kw_only=True)
    _engine: Optional[Engine] = field(default=None, kw_only=True, alias="engine",␣
↪metadata={"serializable": False})

    @get_connection.validator  # pyright: ignore[reportFunctionMemberAccess]
    def validate_get_connection(self, _: Attribute, get_connection: Callable[[],␣
↪SnowflakeConnection]) -> None:
        snowflake_connection = get_connection()
        snowflake = import_optional_dependency("snowflake")

        if not isinstance(snowflake_connection, snowflake.connector.
↪SnowflakeConnection):
            raise ValueError("The get_connection function must return a␣
↪SnowflakeConnection")
        if not snowflake_connection.schema or not snowflake_connection.database:
            raise ValueError("Provide a schema and database for the Snowflake␣
↪connection")

    @lazy_property()
    def engine(self) -> Engine:
        return import_optional_dependency("sqlalchemy").create_engine(
            "snowflake://not@used/db",
```

(continues on next page)

(continued from previous page)

```
            creator=self.get_connection,
        )

    def execute_query(self, query: str) -> Optional[list[BaseSqlDriver.RowResult]]:
        rows = self.execute_query_raw(query)

        if rows:
            return [BaseSqlDriver.RowResult(row) for row in rows]
        return None

    def execute_query_raw(self, query: str) -> Optional[list[dict[str, Any]]]:
        sqlalchemy = import_optional_dependency("sqlalchemy")

        with self.engine.connect() as con:
            results = con.execute(sqlalchemy.text(query))

            if results is not None:
                if results.returns_rows:
                    return [dict(result._mapping) for result in results]
                return None
            raise ValueError("No results found")

    def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]:
        sqlalchemy = import_optional_dependency("sqlalchemy")

        try:
            metadata_obj = sqlalchemy.MetaData()
            metadata_obj.reflect(bind=self.engine)
            table = sqlalchemy.Table(table_name, metadata_obj, schema=schema,␣
↪autoload=True, autoload_with=self.engine)
            return str([(c.name, c.type) for c in table.columns])
        except sqlalchemy.exc.NoSuchTableError:
            return None
```

- `_engine = field(default=None, kw_only=True, alias='engine', metadata={'serializable': False})` class-attribute instance-attribute
- `get_connection = field(kw_only=True)` class-attribute instance-attribute

engine()

Source Code in griptape/drivers/sql/snowflake_sql_driver.py

```
@lazy_property()
def engine(self) -> Engine:
    return import_optional_dependency("sqlalchemy").create_engine(
        "snowflake://not@used/db",
        creator=self.get_connection,
    )
```

execute_query(query)

Source Code in griptape/drivers/sql/snowflake_sql_driver.py

```
def execute_query(self, query: str) -> Optional[list[BaseSqlDriver.RowResult]]:
    rows = self.execute_query_raw(query)

    if rows:
        return [BaseSqlDriver.RowResult(row) for row in rows]
    return None
```

execute_query_raw(query)

Source Code in griptape/drivers/sql/snowflake_sql_driver.py

```
def execute_query_raw(self, query: str) -> Optional[list[dict[str, Any]]]:
    sqlalchemy = import_optional_dependency("sqlalchemy")

    with self.engine.connect() as con:
        results = con.execute(sqlalchemy.text(query))

        if results is not None:
            if results.returns_rows:
                return [dict(result._mapping) for result in results]
            return None
        raise ValueError("No results found")
```

get_table_schema(table_name, schema=None)

Source Code in griptape/drivers/sql/snowflake_sql_driver.py

```
def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]:
    sqlalchemy = import_optional_dependency("sqlalchemy")

    try:
        metadata_obj = sqlalchemy.MetaData()
        metadata_obj.reflect(bind=self.engine)
        table = sqlalchemy.Table(table_name, metadata_obj, schema=schema,␣
↪autoload=True, autoload_with=self.engine)
        return str([(c.name, c.type) for c in table.columns])
    except sqlalchemy.exc.NoSuchTableError:
        return None
```

validate*get_connection(*, get_connection)

Source Code in griptape/drivers/sql/snowflake_sql_driver.py

```
@get_connection.validator  # pyright: ignore[reportFunctionMemberAccess]
def validate_get_connection(self, _: Attribute, get_connection: Callable[[],␣
↪SnowflakeConnection]) -> None:
    snowflake_connection = get_connection()
    snowflake = import_optional_dependency("snowflake")
```

(continues on next page)

(continued from previous page)

```
    if not isinstance(snowflake_connection, snowflake.connector.SnowflakeConnection):
        raise ValueError("The get_connection function must return a␣
↪SnowflakeConnection")
    if not snowflake_connection.schema or not snowflake_connection.database:
        raise ValueError("Provide a schema and database for the Snowflake connection")
```

sql_driver

Bases: *BaseSqlDriver* “)

Source Code in griptape/drivers/sql/sql_driver.py

```
@define
class SqlDriver(BaseSqlDriver):
    engine_url: str = field(kw_only=True)
    create_engine_params: dict = field(factory=dict, kw_only=True)
    _engine: Optional[Engine] = field(default=None, kw_only=True, alias="engine",␣
↪metadata={"serializable": False})

    @lazy_property()
    def engine(self) -> Engine:
        return import_optional_dependency("sqlalchemy").create_engine(self.engine_url,
↪ **self.create_engine_params)

    def execute_query(self, query: str) -> Optional[list[BaseSqlDriver.RowResult]]:
        rows = self.execute_query_raw(query)

        if rows:
            return [BaseSqlDriver.RowResult(row) for row in rows]
        return None

    def execute_query_raw(self, query: str) -> Optional[list[dict[str,␣
↪Optional[Any]]]]:
        sqlalchemy = import_optional_dependency("sqlalchemy")

        with self.engine.connect() as con:
            results = con.execute(sqlalchemy.text(query))

            if results is not None:
                if results.returns_rows:
                    return [dict(result._mapping) for result in results]
                con.commit()
                return None
            raise ValueError("No result found")

    def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]:
        sqlalchemy_exc = import_optional_dependency("sqlalchemy.exc")

        try:
            return str(SqlDriver._get_table_schema(self.engine, table_name, schema))
```

(continues on next page)

(continued from previous page)

```
        except sqlalchemy_exc.NoSuchTableError:
            return None

    @staticmethod
    @lru_cache
    def _get_table_schema(
        engine: Engine, table_name: str, schema: Optional[str] = None
    ) -> Optional[list[tuple[str, str]]]:
        sqlalchemy = import_optional_dependency("sqlalchemy")

        return [(col["name"], col["type"]) for col in sqlalchemy.inspect(engine).get_
↪columns(table_name, schema=schema)]
```

- `_engine = field(default=None, kw_only=True, alias='engine', metadata={'serializable': False})` class-attribute instance-attribute
- `create_engine_params = field(factory=dict, kw_only=True)` class-attribute instance-attribute
- `engine_url = field(kw_only=True)` class-attribute instance-attribute

_get_table_schema(engine, table_name, schema=None)cachedstaticmethod

Source Code in griptape/drivers/sql/sql_driver.py

```
@staticmethod
@lru_cache
def _get_table_schema(
    engine: Engine, table_name: str, schema: Optional[str] = None
) -> Optional[list[tuple[str, str]]]:
    sqlalchemy = import_optional_dependency("sqlalchemy")

    return [(col["name"], col["type"]) for col in sqlalchemy.inspect(engine).get_
↪columns(table_name, schema=schema)]
```

engine()

Source Code in griptape/drivers/sql/sql_driver.py

```
@lazy_property()
def engine(self) -> Engine:
    return import_optional_dependency("sqlalchemy").create_engine(self.engine_url,␣
↪**self.create_engine_params)
```

execute_query(query)

Source Code in griptape/drivers/sql/sql_driver.py

```
def execute_query(self, query: str) -> Optional[list[BaseSqlDriver.RowResult]]:
    rows = self.execute_query_raw(query)

    if rows:
        return [BaseSqlDriver.RowResult(row) for row in rows]
    return None
```

execute_query_raw(query)

Source Code in griptape/drivers/sql/sql_driver.py

```
def execute_query_raw(self, query: str) -> Optional[list[dict[str, Optional[Any]]]]:
    sqlalchemy = import_optional_dependency("sqlalchemy")

    with self.engine.connect() as con:
        results = con.execute(sqlalchemy.text(query))

        if results is not None:
            if results.returns_rows:
                return [dict(result._mapping) for result in results]
            con.commit()
            return None
        raise ValueError("No result found")
```

get_table_schema(table_name, schema=None)

Source Code in griptape/drivers/sql/sql_driver.py

```
def get_table_schema(self, table_name: str, schema: Optional[str] = None) ->␣
↪Optional[str]:
    sqlalchemy_exc = import_optional_dependency("sqlalchemy.exc")

    try:
        return str(SqlDriver._get_table_schema(self.engine, table_name, schema))
    except sqlalchemy_exc.NoSuchTableError:
        return None
```

structure_run

- `__all__ = ['BaseStructureRunDriver']` module-attribute

Bases:

ABC

Attributes

Name	Type	Description
<a href="./#griptape.drivers.structure_run.BaseStructureRunDriver.env" title="env = field(default=Factory(dict), kw_only=True) class-attribute instance-attribute (griptape.drivers.structure_run.BaseStructureRunDriver.env)"><code class="language-python">	*dict[str, str]*	Environment variables to set before running the Structure.

Source Code in griptape/drivers/structure_run/base_structure_run_driver.py

```
@define
class BaseStructureRunDriver(ABC):
    """Base class for Structure Run Drivers.

    Attributes:
        env: Environment variables to set before running the Structure.
    """

    env: dict[str, str] = field(default=Factory(dict), kw_only=True)

    def run(self, *args: BaseArtifact) -> BaseArtifact:
        return self.try_run(*args)

    @abstractmethod
    def try_run(self, *args: BaseArtifact) -> BaseArtifact: ...
```

- `env = field(default=Factory(dict), kw_only=True)` class-attribute instance-attribute

run(*args)

Source Code in griptape/drivers/structure_run/base_structure_run_driver.py

```
def run(self, *args: BaseArtifact) -> BaseArtifact:
    return self.try_run(*args)
```

try_run(*args)abstractmethod

Source Code in griptape/drivers/structure_run/base_structure_run_driver.py

```
@abstractmethod
def try_run(self, *args: BaseArtifact) -> BaseArtifact: ...
```

base_structure_run_driver

Bases:

ABC

Attributes

Name	Type	Description
<a href="base_structure_run_driver.mdx#griptape.drivers.structure_run.base_structure_run_driver.BaseStructureRunDriver.env" title="env = field(default=Factory(dict), kw_only=True) class-attribute instance-attribute (griptape.drivers.structure_run.base_structure_run_driver.BaseStructureRunDriver.env)"><code class="language-python">	*dict[str, str]*	Environment variables to set before running the Structure.

Source Code in griptape/drivers/structure_run/base_structure_run_driver.py

```
@define
class BaseStructureRunDriver(ABC):
    """Base class for Structure Run Drivers.

    Attributes:
        env: Environment variables to set before running the Structure.
    """

    env: dict[str, str] = field(default=Factory(dict), kw_only=True)

    def run(self, *args: BaseArtifact) -> BaseArtifact:
        return self.try_run(*args)

    @abstractmethod
    def try_run(self, *args: BaseArtifact) -> BaseArtifact: ...
```

- `env = field(default=Factory(dict), kw_only=True)` class-attribute instance-attribute

run(*args)

Source Code in griptape/drivers/structure_run/base_structure_run_driver.py

```
def run(self, *args: BaseArtifact) -> BaseArtifact:
    return self.try_run(*args)
```

try_run(*args)abstractmethod

Source Code in griptape/drivers/structure_run/base_structure_run_driver.py

```
@abstractmethod
def try_run(self, *args: BaseArtifact) -> BaseArtifact: ...
```

griptape_cloud

- `__all__ = ['GriptapeCloudStructureRunDriver']` module-attribute

Bases: *BaseStructureRunDriver* “)

Source Code in griptape/drivers/structure_run/griptape_cloud_structure_run_driver.py

```
@define
class GriptapeCloudStructureRunDriver(BaseStructureRunDriver):
    base_url: str = field(default="https://cloud.griptape.ai", kw_only=True)
    api_key: str = field(kw_only=True)
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        kw_only=True,
    )
    structure_id: str = field(kw_only=True)
    structure_run_wait_time_interval: int = field(default=2, kw_only=True)
    structure_run_max_wait_time_attempts: int = field(default=20, kw_only=True)
    async_run: bool = field(default=False, kw_only=True)

    def try_run(self, *args: BaseArtifact) -> BaseArtifact | InfoArtifact:
        structure_run_id = self._create_run(*args)

        if self.async_run:
            return InfoArtifact("Run started successfully")
        return self._get_run_result(structure_run_id)

    def _create_run(self, *args: BaseArtifact) -> str:
        url = griptape_cloud_url(self.base_url, f"api/structures/{self.structure_id}/
↪runs")

        env_vars = [{"name": key, "value": value, "source": "manual"} for key, value␣
↪in self.env.items()]

        response = requests.post(
            url,
            json={"args": [arg.value for arg in args], "env_vars": env_vars},
            headers=self.headers,
        )
        response.raise_for_status()
        response_json = response.json()

        return response_json["structure_run_id"]

    def _get_run_result(self, structure_run_id: str) -> BaseArtifact | InfoArtifact:
```

(continues on next page)

(continued from previous page)

```
        events = self._get_run_events(structure_run_id)
        output = None

        for event in events:
            event_type = event["type"]
            event_payload = event.get("payload", {})
            if event["origin"] == "USER":
                try:
                    if "span_id" in event_payload:
                        span_id = event_payload.pop("span_id")
                        if "meta" in event_payload:
                            event_payload["meta"]["span_id"] = span_id
                        else:
                            event_payload["meta"] = {"span_id": span_id}
                    EventBus.publish_event(BaseEvent.from_dict(event_payload))
                except ValueError as e:
                    logger.warning("Failed to deserialize event: %s", e)
                if event["type"] == "FinishStructureRunEvent":
                    output = BaseArtifact.from_dict(event_payload["output_task_output
↪"])
            elif event["origin"] == "SYSTEM":
                if event_type == "StructureRunError":
                    output = ErrorArtifact(event_payload["status_detail"]["error"])

        if output is None:
            raise ValueError("Output not found.")

        return output

    def _get_run_events(self, structure_run_id: str) -> Iterator[dict]:
        url = griptape_cloud_url(self.base_url, f"api/structure-runs/{structure_run_
↪id}/events/stream")
        with requests.get(url, headers=self.headers, stream=True) as response:
            response.raise_for_status()
            for line in response.iter_lines():
                if line:
                    decoded_line = line.decode("utf-8")
                    if decoded_line.startswith("data:"):
                        yield json.loads(decoded_line.removeprefix("data:").strip())
```

- `api_key = field(kw_only=True)` class-attribute instance-attribute
- `async_run = field(default=False, kw_only=True)` class-attribute instance-attribute
- `base_url = field(default='https://cloud.griptape.ai', kw_only=True)` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `structure_id = field(kw_only=True)` class-attribute instance-attribute
- `structure_run_max_wait_time_attempts = field(default=20, kw_only=True)` class-attribute instance-attribute
- `structure_run_wait_time_interval = field(default=2, kw_only=True)` class-attribute

instance-attribute

_create_run(*args)

Source Code in griptape/drivers/structure_run/griptape_cloud_structure_run_driver.py

```
def _create_run(self, *args: BaseArtifact) -> str:
    url = griptape_cloud_url(self.base_url, f"api/structures/{self.structure_id}/runs
↪")

    env_vars = [{"name": key, "value": value, "source": "manual"} for key, value in␣
↪self.env.items()]

    response = requests.post(
        url,
        json={"args": [arg.value for arg in args], "env_vars": env_vars},
        headers=self.headers,
    )
    response.raise_for_status()
    response_json = response.json()

    return response_json["structure_run_id"]
```

_get_run_events(structure_run_id)

Source Code in griptape/drivers/structure_run/griptape_cloud_structure_run_driver.py

```
def _get_run_events(self, structure_run_id: str) -> Iterator[dict]:
    url = griptape_cloud_url(self.base_url, f"api/structure-runs/{structure_run_id}/
↪events/stream")
    with requests.get(url, headers=self.headers, stream=True) as response:
        response.raise_for_status()
        for line in response.iter_lines():
            if line:
                decoded_line = line.decode("utf-8")
                if decoded_line.startswith("data:"):
                    yield json.loads(decoded_line.removeprefix("data:").strip())
```

_get_run_result(structure_run_id)

Source Code in griptape/drivers/structure_run/griptape_cloud_structure_run_driver.py

```
def _get_run_result(self, structure_run_id: str) -> BaseArtifact | InfoArtifact:
    events = self._get_run_events(structure_run_id)
    output = None

    for event in events:
        event_type = event["type"]
        event_payload = event.get("payload", {})
        if event["origin"] == "USER":
            try:
                if "span_id" in event_payload:
                    span_id = event_payload.pop("span_id")
```

(continues on next page)

(continued from previous page)

```
                    if "meta" in event_payload:
                        event_payload["meta"]["span_id"] = span_id
                    else:
                        event_payload["meta"] = {"span_id": span_id}
                EventBus.publish_event(BaseEvent.from_dict(event_payload))
            except ValueError as e:
                logger.warning("Failed to deserialize event: %s", e)
            if event["type"] == "FinishStructureRunEvent":
                output = BaseArtifact.from_dict(event_payload["output_task_output"])
        elif event["origin"] == "SYSTEM":
            if event_type == "StructureRunError":
                output = ErrorArtifact(event_payload["status_detail"]["error"])

    if output is None:
        raise ValueError("Output not found.")

    return output
```

try_run(*args)

Source Code in griptape/drivers/structure_run/griptape_cloud_structure_run_driver.py

```
def try_run(self, *args: BaseArtifact) -> BaseArtifact | InfoArtifact:
    structure_run_id = self._create_run(*args)

    if self.async_run:
        return InfoArtifact("Run started successfully")
    return self._get_run_result(structure_run_id)
```

griptape_cloud_structure_run_driver

- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *BaseStructureRunDriver* “)

Source Code in griptape/drivers/structure_run/griptape_cloud_structure_run_driver.py

```
@define
class GriptapeCloudStructureRunDriver(BaseStructureRunDriver):
    base_url: str = field(default="https://cloud.griptape.ai", kw_only=True)
    api_key: str = field(kw_only=True)
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        kw_only=True,
    )
    structure_id: str = field(kw_only=True)
    structure_run_wait_time_interval: int = field(default=2, kw_only=True)
    structure_run_max_wait_time_attempts: int = field(default=20, kw_only=True)
    async_run: bool = field(default=False, kw_only=True)

    def try_run(self, *args: BaseArtifact) -> BaseArtifact | InfoArtifact:
```

(continues on next page)

(continued from previous page)

```
        structure_run_id = self._create_run(*args)

        if self.async_run:
            return InfoArtifact("Run started successfully")
        return self._get_run_result(structure_run_id)

    def _create_run(self, *args: BaseArtifact) -> str:
        url = griptape_cloud_url(self.base_url, f"api/structures/{self.structure_id}/
↪runs")

        env_vars = [{"name": key, "value": value, "source": "manual"} for key, value␣
↪in self.env.items()]

        response = requests.post(
            url,
            json={"args": [arg.value for arg in args], "env_vars": env_vars},
            headers=self.headers,
        )
        response.raise_for_status()
        response_json = response.json()

        return response_json["structure_run_id"]

    def _get_run_result(self, structure_run_id: str) -> BaseArtifact | InfoArtifact:
        events = self._get_run_events(structure_run_id)
        output = None

        for event in events:
            event_type = event["type"]
            event_payload = event.get("payload", {})
            if event["origin"] == "USER":
                try:
                    if "span_id" in event_payload:
                        span_id = event_payload.pop("span_id")
                        if "meta" in event_payload:
                            event_payload["meta"]["span_id"] = span_id
                        else:
                            event_payload["meta"] = {"span_id": span_id}
                    EventBus.publish_event(BaseEvent.from_dict(event_payload))
                except ValueError as e:
                    logger.warning("Failed to deserialize event: %s", e)
                if event["type"] == "FinishStructureRunEvent":
                    output = BaseArtifact.from_dict(event_payload["output_task_output
↪"])
            elif event["origin"] == "SYSTEM":
                if event_type == "StructureRunError":
                    output = ErrorArtifact(event_payload["status_detail"]["error"])

        if output is None:
            raise ValueError("Output not found.")

        return output
```

(continues on next page)

(continued from previous page)

```
    def _get_run_events(self, structure_run_id: str) -> Iterator[dict]:
        url = griptape_cloud_url(self.base_url, f"api/structure-runs/{structure_run_
↪id}/events/stream")
        with requests.get(url, headers=self.headers, stream=True) as response:
            response.raise_for_status()
            for line in response.iter_lines():
                if line:
                    decoded_line = line.decode("utf-8")
                    if decoded_line.startswith("data:"):
                        yield json.loads(decoded_line.removeprefix("data:").strip())
```

- `api_key = field(kw_only=True)` class-attribute instance-attribute
- `async_run = field(default=False, kw_only=True)` class-attribute instance-attribute
- `base_url = field(default='https://cloud.griptape.ai', kw_only=True)` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `structure_id = field(kw_only=True)` class-attribute instance-attribute
- `structure_run_max_wait_time_attempts = field(default=20, kw_only=True)` class-attribute instance-attribute
- `structure_run_wait_time_interval = field(default=2, kw_only=True)` class-attribute instance-attribute

_create_run(*args)

Source Code in griptape/drivers/structure_run/griptape_cloud_structure_run_driver.py

```
def _create_run(self, *args: BaseArtifact) -> str:
    url = griptape_cloud_url(self.base_url, f"api/structures/{self.structure_id}/runs
↪")

    env_vars = [{"name": key, "value": value, "source": "manual"} for key, value in␣
↪self.env.items()]

    response = requests.post(
        url,
        json={"args": [arg.value for arg in args], "env_vars": env_vars},
        headers=self.headers,
    )
    response.raise_for_status()
    response_json = response.json()

    return response_json["structure_run_id"]
```

_get_run_events(structure_run_id)

Source Code in griptape/drivers/structure_run/griptape_cloud_structure_run_driver.py

```
def _get_run_events(self, structure_run_id: str) -> Iterator[dict]:
    url = griptape_cloud_url(self.base_url, f"api/structure-runs/{structure_run_id}/
↪events/stream")
    with requests.get(url, headers=self.headers, stream=True) as response:
        response.raise_for_status()
        for line in response.iter_lines():
            if line:
                decoded_line = line.decode("utf-8")
                if decoded_line.startswith("data:"):
                    yield json.loads(decoded_line.removeprefix("data:").strip())
```

_get_run_result(structure_run_id)

Source Code in griptape/drivers/structure_run/griptape_cloud_structure_run_driver.py

```
def _get_run_result(self, structure_run_id: str) -> BaseArtifact | InfoArtifact:
    events = self._get_run_events(structure_run_id)
    output = None

    for event in events:
        event_type = event["type"]
        event_payload = event.get("payload", {})
        if event["origin"] == "USER":
            try:
                if "span_id" in event_payload:
                    span_id = event_payload.pop("span_id")
                    if "meta" in event_payload:
                        event_payload["meta"]["span_id"] = span_id
                    else:
                        event_payload["meta"] = {"span_id": span_id}
                EventBus.publish_event(BaseEvent.from_dict(event_payload))
            except ValueError as e:
                logger.warning("Failed to deserialize event: %s", e)
            if event["type"] == "FinishStructureRunEvent":
                output = BaseArtifact.from_dict(event_payload["output_task_output"])
        elif event["origin"] == "SYSTEM":
            if event_type == "StructureRunError":
                output = ErrorArtifact(event_payload["status_detail"]["error"])

    if output is None:
        raise ValueError("Output not found.")

    return output
```

try_run(*args)

Source Code in griptape/drivers/structure_run/griptape_cloud_structure_run_driver.py

```
def try_run(self, *args: BaseArtifact) -> BaseArtifact | InfoArtifact:
    structure_run_id = self._create_run(*args)

    if self.async_run:
        return InfoArtifact("Run started successfully")
    return self._get_run_result(structure_run_id)
```

local

- `__all__ = ['LocalStructureRunDriver']` module-attribute

Bases: *BaseStructureRunDriver* “)

Source Code in griptape/drivers/structure_run/local_structure_run_driver.py

```
@define
class LocalStructureRunDriver(BaseStructureRunDriver):
    create_structure: Callable[[], Structure] = field(kw_only=True)

    def try_run(self, *args: BaseArtifact) -> BaseArtifact:
        old_env = os.environ.copy()
        try:
            os.environ.update(self.env)
            structure = self.create_structure().run(*[arg.value for arg in args])
        finally:
            os.environ.clear()
            os.environ.update(old_env)

        return structure.output
```

- `create_structure = field(kw_only=True)` class-attribute instance-attribute

try_run(*args)

Source Code in griptape/drivers/structure_run/local_structure_run_driver.py

```
def try_run(self, *args: BaseArtifact) -> BaseArtifact:
    old_env = os.environ.copy()
    try:
        os.environ.update(self.env)
        structure = self.create_structure().run(*[arg.value for arg in args])
    finally:
        os.environ.clear()
        os.environ.update(old_env)

    return structure.output
```

local_structure_run_driver

Bases: *BaseStructureRunDriver* “)

Source Code in griptape/drivers/structure_run/local_structure_run_driver.py

```
@define
class LocalStructureRunDriver(BaseStructureRunDriver):
    create_structure: Callable[[], Structure] = field(kw_only=True)

    def try_run(self, *args: BaseArtifact) -> BaseArtifact:
        old_env = os.environ.copy()
        try:
            os.environ.update(self.env)
            structure = self.create_structure().run(*[arg.value for arg in args])
        finally:
            os.environ.clear()
            os.environ.update(old_env)

        return structure.output
```

- `create_structure = field(kw_only=True)` class-attribute instance-attribute

try_run(*args)

Source Code in griptape/drivers/structure_run/local_structure_run_driver.py

```
def try_run(self, *args: BaseArtifact) -> BaseArtifact:
    old_env = os.environ.copy()
    try:
        os.environ.update(self.env)
        structure = self.create_structure().run(*[arg.value for arg in args])
    finally:
        os.environ.clear()
        os.environ.update(old_env)

    return structure.output
```

text_to_speech

- `__all__ = ['BaseTextToSpeechDriver']` module-attribute

Bases: *SerializableMixin* “) , *ExponentialBackoffMixin* ”) , ABC

Source Code in griptape/drivers/text_to_speech/base_text_to_speech_driver.py

```
@define
class BaseTextToSpeechDriver(SerializableMixin, ExponentialBackoffMixin, ABC):
    model: str = field(kw_only=True, metadata={"serializable": True})

    def before_run(self, prompts: list[str]) -> None:
        EventBus.publish_event(StartTextToSpeechEvent(prompts=prompts))

    def after_run(self) -> None:
        EventBus.publish_event(FinishTextToSpeechEvent())

    def run_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
        for attempt in self.retrying():
            with attempt:
```

(continues on next page)

(continued from previous page)

```
                self.before_run(prompts)
                result = self.try_text_to_audio(prompts)
                self.after_run()

                return result

        raise Exception("Failed to run text to audio generation")

    @abstractmethod
    def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact: ...
```

- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

after_run()

Source Code in griptape/drivers/text_to_speech/base_text_to_speech_driver.py

```
def after_run(self) -> None:
    EventBus.publish_event(FinishTextToSpeechEvent())
```

before_run(prompts)

Source Code in griptape/drivers/text_to_speech/base_text_to_speech_driver.py

```
def before_run(self, prompts: list[str]) -> None:
    EventBus.publish_event(StartTextToSpeechEvent(prompts=prompts))
```

run_text_to_audio(prompts)

Source Code in griptape/drivers/text_to_speech/base_text_to_speech_driver.py

```
def run_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
    for attempt in self.retrying():
        with attempt:
            self.before_run(prompts)
            result = self.try_text_to_audio(prompts)
            self.after_run()

            return result

    raise Exception("Failed to run text to audio generation")
```

try_text_to_audio(prompts)abstractmethod

Source Code in griptape/drivers/text_to_speech/base_text_to_speech_driver.py

```
@abstractmethod
def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact: ...
```

azure_openai_text_to_speech_driver

Bases: *AzureOpenAiTextToSpeechDriver* “)

Attributes

Name	Type	Description
<a href="azure_openai_text_to_speech_drivetape.drivers.text_to_speecnai_text_to_speech_driveOpenAiTextToSpeechDriver.azure_deployment" title="azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.text_to_speecnai_text_to_speech_driveOpenAiTextToSpeechDriver.azure_deployment)"><code class="language-python">	*str*	An optional Azure OpenAi deployment id. Defaults to the model name.
<a href="azure_openai_text_to_speech_drivetape.drivers.text_to_speecnai_text_to_speech_driveOpenAiTextToSpeechDriver.azure_endpoint" title="azure_endpoint = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.text_to_speecnai_text_to_speech_driveOpenAiTextToSpeechDriver.azure_endpoint)"><code class="language-python">	*str*	An Azure OpenAi endpoint.

continues on next page

Table 216 – continued from previous page

Name	Type	Description
<a href="azure_openai_text_to_speech_driver tape.drivers.text_to_speec nai_text_to_speech_driver OpenAiTextToSpeech-Driver.azure_ad_token" title="azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.text_to_speec nai_text_to_speech_driver OpenAiTextToSpeech-Driver.azure_ad_token)"><code class="language-python">	*Optional[str]*	An optional Azure Active Directory token.
<a href="azure_openai_text_to_speech_driver tape.drivers.text_to_speec nai_text_to_speech_driver OpenAiTextToSpeech-Driver.azure_ad_token_provider" title="azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.text_to_speec nai_text_to_speech_driver OpenAiTextToSpeech-Driver.azure_ad_token_provider)"><code class="language-python">	*Optional[Callable[[], str]]*	An optional Azure Active Directory token provider.

continues on next page

Table 216 – continued from previous page

Name	Type	Description
<a href="azure_openai_text_to_speech_driver.html#griptape.drivers.text_to_speech.azure_openai_text_to_speech_driver.OpenAiTextToSpeechDriver.api_version" title="api_version = field(default='2024-07-01-preview', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.text_to_speech.azure_openai_text_to_speech_driver.OpenAiTextToSpeechDriver.api_version)"><code class="language-python">	*str*	An Azure OpenAi API version.
<a href="azure_openai_text_to_speech_driver.html#griptape.drivers.text_to_speech.azure_openai_text_to_speech_driver.OpenAiTextToSpeechDriver.client" title="client() (griptape.drivers.text_to_speech.azure_openai_text_to_speech_driver.OpenAiTextToSpeechDriver.client)"><code class="language-python">	*AzureOpenAI*	An *openai.AzureOpenAI* client.

Source Code in griptape/drivers/text_to_speech/azure_openai_text_to_speech_driver.py

```
@define
class AzureOpenAiTextToSpeechDriver(OpenAiTextToSpeechDriver):
    """Azure OpenAi Text to Speech Driver.

    Attributes:
        azure_deployment: An optional Azure OpenAi deployment id. Defaults to the␣
↪model name.
        azure_endpoint: An Azure OpenAi endpoint.
        azure_ad_token: An optional Azure Active Directory token.
        azure_ad_token_provider: An optional Azure Active Directory token provider.
        api_version: An Azure OpenAi API version.
        client: An `openai.AzureOpenAI` client.
    """

    model: str = field(default="tts", kw_only=True, metadata={"serializable": True})
    azure_deployment: str = field(
```

(continues on next page)

(continued from previous page)

```
        kw_only=True,
        default=Factory(lambda self: self.model, takes_self=True),
        metadata={"serializable": True},
    )
    azure_endpoint: str = field(kw_only=True, metadata={"serializable": True})
    azure_ad_token: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": False})
    azure_ad_token_provider: Optional[Callable[[], str]] = field(
        kw_only=True,
        default=None,
        metadata={"serializable": False},
    )
    api_version: str = field(default="2024-07-01-preview", kw_only=True, metadata={
↪"serializable": True})
    _client: Optional[openai.AzureOpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.AzureOpenAI:
        return openai.AzureOpenAI(
            organization=self.organization,
            api_key=self.api_key,
            api_version=self.api_version,
            azure_endpoint=self.azure_endpoint,
            azure_deployment=self.azure_deployment,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_version = field(default='2024-07-01-preview', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True})` class-attribute instance-attribute
- `azure_endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(default='tts', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/text_to_speech/azure_openai_text_to_speech_driver.py

```
@lazy_property()
def client(self) -> openai.AzureOpenAI:
    return openai.AzureOpenAI(
        organization=self.organization,
        api_key=self.api_key,
        api_version=self.api_version,
        azure_endpoint=self.azure_endpoint,
        azure_deployment=self.azure_deployment,
        azure_ad_token=self.azure_ad_token,
        azure_ad_token_provider=self.azure_ad_token_provider,
    )
```

base_text_to_speech_driver

Bases: *SerializableMixin* “) , *ExponentialBackoffMixin* ”) , ABC

Source Code in griptape/drivers/text_to_speech/base_text_to_speech_driver.py

```
@define
class BaseTextToSpeechDriver(SerializableMixin, ExponentialBackoffMixin, ABC):
    model: str = field(kw_only=True, metadata={"serializable": True})

    def before_run(self, prompts: list[str]) -> None:
        EventBus.publish_event(StartTextToSpeechEvent(prompts=prompts))

    def after_run(self) -> None:
        EventBus.publish_event(FinishTextToSpeechEvent())

    def run_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
        for attempt in self.retrying():
            with attempt:
                self.before_run(prompts)
                result = self.try_text_to_audio(prompts)
                self.after_run()

                return result

        raise Exception("Failed to run text to audio generation")

    @abstractmethod
    def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact: ...
```

- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

after_run()

Source Code in griptape/drivers/text_to_speech/base_text_to_speech_driver.py

```
def after_run(self) -> None:
    EventBus.publish_event(FinishTextToSpeechEvent())
```

before_run(prompts)

Source Code in griptape/drivers/text_to_speech/base_text_to_speech_driver.py

```
def before_run(self, prompts: list[str]) -> None:
    EventBus.publish_event(StartTextToSpeechEvent(prompts=prompts))
```

run_text_to_audio(prompts)

Source Code in griptape/drivers/text_to_speech/base_text_to_speech_driver.py

```
def run_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
    for attempt in self.retrying():
        with attempt:
            self.before_run(prompts)
            result = self.try_text_to_audio(prompts)
            self.after_run()

            return result

    raise Exception("Failed to run text to audio generation")
```

try_text_to_audio(prompts)abstractmethod

Source Code in griptape/drivers/text_to_speech/base_text_to_speech_driver.py

```
@abstractmethod
def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact: ...
```

dummy

- `__all__ = ['DummyTextToSpeechDriver']` module-attribute

Bases: *BaseTextToSpeechDriver* “)

Source Code in griptape/drivers/text_to_speech/dummy_text_to_speech_driver.py

```
@define
class DummyTextToSpeechDriver(BaseTextToSpeechDriver):
    model: None = field(init=False, default=None, kw_only=True)

    def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
        raise DummyError(__class__.__name__, "try_text_to_audio")
```

- `model = field(init=False, default=None, kw_only=True)` class-attribute instance-attribute

try_text_to_audio(prompts)

Source Code in griptape/drivers/text_to_speech/dummy_text_to_speech_driver.py

```
def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
    raise DummyError(__class__.__name__, "try_text_to_audio")
```

dummy_text_to_speech_driver

Bases: *BaseTextToSpeechDriver* “)

Source Code in griptape/drivers/text_to_speech/dummy_text_to_speech_driver.py

```
@define
class DummyTextToSpeechDriver(BaseTextToSpeechDriver):
    model: None = field(init=False, default=None, kw_only=True)

    def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
        raise DummyError(__class__.__name__, "try_text_to_audio")
```

- `model = field(init=False, default=None, kw_only=True)` class-attribute instance-attribute

try_text_to_audio(prompts)

Source Code in griptape/drivers/text_to_speech/dummy_text_to_speech_driver.py

```
def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
    raise DummyError(__class__.__name__, "try_text_to_audio")
```

elevenlabs

- `__all__ = ['ElevenLabsTextToSpeechDriver']` module-attribute

Bases: *BaseTextToSpeechDriver* “)

Source Code in griptape/drivers/text_to_speech/elevenlabs_text_to_speech_driver.py

```
@define
class ElevenLabsTextToSpeechDriver(BaseTextToSpeechDriver):
    api_key: str = field(kw_only=True, metadata={"serializable": True})
    voice: str = field(kw_only=True, metadata={"serializable": True})
    output_format: str = field(default="mp3_44100_128", kw_only=True, metadata={
↪"serializable": True})
    _client: Optional[ElevenLabs] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> ElevenLabs:
        return import_optional_dependency("elevenlabs.client").ElevenLabs(api_
↪key=self.api_key)

    def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
        audio = self.client.generate(
            text=". ".join(prompts),
            voice=self.voice,
```

(continues on next page)

(continued from previous page)

```
            model=self.model,
            output_format=self.output_format,
        )

        content = b""
        for chunk in audio:
            content += chunk

        # All ElevenLabs audio format strings have the following structure:
        # {format}_{sample_rate}_{bitrate}
        artifact_format = self.output_format.split("_")[0]

        return AudioArtifact(value=content, format=artifact_format)
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output_format = field(default='mp3_44100_128', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `voice = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/text_to_speech/elevenlabs_text_to_speech_driver.py

```
@lazy_property()
def client(self) -> ElevenLabs:
    return import_optional_dependency("elevenlabs.client").ElevenLabs(api_key=self.
↪api_key)
```

try_text_to_audio(prompts)

Source Code in griptape/drivers/text_to_speech/elevenlabs_text_to_speech_driver.py

```
def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
    audio = self.client.generate(
        text=". ".join(prompts),
        voice=self.voice,
        model=self.model,
        output_format=self.output_format,
    )

    content = b""
    for chunk in audio:
        content += chunk

    # All ElevenLabs audio format strings have the following structure:
    # {format}_{sample_rate}_{bitrate}
```

(continues on next page)

(continued from previous page)

```
    artifact_format = self.output_format.split("_")[0]

    return AudioArtifact(value=content, format=artifact_format)
```

elevenlabs_text_to_speech_driver

Bases: *BaseTextToSpeechDriver* “)

Source Code in griptape/drivers/text_to_speech/elevenlabs_text_to_speech_driver.py

```
@define
class ElevenLabsTextToSpeechDriver(BaseTextToSpeechDriver):
    api_key: str = field(kw_only=True, metadata={"serializable": True})
    voice: str = field(kw_only=True, metadata={"serializable": True})
    output_format: str = field(default="mp3_44100_128", kw_only=True, metadata={
↪"serializable": True})
    _client: Optional[ElevenLabs] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> ElevenLabs:
        return import_optional_dependency("elevenlabs.client").ElevenLabs(api_
↪key=self.api_key)

    def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
        audio = self.client.generate(
            text=". ".join(prompts),
            voice=self.voice,
            model=self.model,
            output_format=self.output_format,
        )

        content = b""
        for chunk in audio:
            content += chunk

        # All ElevenLabs audio format strings have the following structure:
        # {format}_{sample_rate}_{bitrate}
        artifact_format = self.output_format.split("_")[0]

        return AudioArtifact(value=content, format=artifact_format)
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output_format = field(default='mp3_44100_128', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `voice = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/text_to_speech/elevenlabs_text_to_speech_driver.py

```
@lazy_property()
def client(self) -> ElevenLabs:
    return import_optional_dependency("elevenlabs.client").ElevenLabs(api_key=self.
↪api_key)
```

try_text_to_audio(prompts)

Source Code in griptape/drivers/text_to_speech/elevenlabs_text_to_speech_driver.py

```
def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
    audio = self.client.generate(
        text=". ".join(prompts),
        voice=self.voice,
        model=self.model,
        output_format=self.output_format,
    )

    content = b""
    for chunk in audio:
        content += chunk

    # All ElevenLabs audio format strings have the following structure:
    # {format}_{sample_rate}_{bitrate}
    artifact_format = self.output_format.split("_")[0]

    return AudioArtifact(value=content, format=artifact_format)
```

openai

- `__all__ = ['AzureOpenAiTextToSpeechDriver', 'OpenAiTextToSpeechDriver']` module-attribute

Bases: *AzureOpenAiTextToSpeechDriver* “)

Attributes

Name	Type	Description
<a href="openai.mdx#griptape.drivers.text_to_speech.openai.AzureOpenAiTextToSpeechDriver.azure_deployment" title="azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.text_to_speech.openai.AzureOpenAiTextToSpeechDriver.azure_deployment)"><code class="language-python">	*str*	An optional Azure OpenAi deployment id. Defaults to the model name.
<a href="openai.mdx#griptape.drivers.text_to_speech.openai.AzureOpenAiTextToSpeechDriver.azure_endpoint" title="azure_endpoint = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.text_to_speech.openai.AzureOpenAiTextToSpeechDriver.azure_endpoint)"><code class="language-python">	*str*	An Azure OpenAi endpoint.

continues on next page

Table 217 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.text_to_speech.openai.AzureOpenAiTextToSpeechDriver.azure_ad_token" title="azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.text_to_speech.openai.AzureOpenAiTextToSpeechDriver.azure_ad_token)"><code class="language-python">	*Optional[str]*	An optional Azure Active Directory token.
<a href="openai.mdx#griptape.drivers.text_to_speech.openai.AzureOpenAiTextToSpeechDriver.azure_ad_token_provider" title="azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.text_to_speech.openai.AzureOpenAiTextToSpeechDriver.azure_ad_token_provider)"><code class="language-python">	*Optional[Callable[[], str]]*	An optional Azure Active Directory token provider.

continues on next page

Table 217 – continued from previous page

Name	Type	Description
<a href="openai.mdx#griptape.drivers.text_to_speech.openai.AzureOpenAiTextToSpeechDriver.api_version" title="api_version = field(default='2024-07-01-preview', kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.text_to_speech.openai.AzureOpenAiTextToSpeechDriver.api_version)"><code class="language-python">	*str*	An Azure OpenAi API version.
<a href="openai.mdx#griptape.drivers.text_to_speech.openai.AzureOpenAiTextToSpeechDriver.client" title="client() (griptape.drivers.text_to_speech.openai.AzureOpenAiTextToSpeechDriver.client)"><code class="language-python">	*AzureOpenAI*	An *openai.AzureOpenAI* client.

Source Code in griptape/drivers/text_to_speech/azure_openai_text_to_speech_driver.py

```
@define
class AzureOpenAiTextToSpeechDriver(OpenAiTextToSpeechDriver):
    """Azure OpenAi Text to Speech Driver.

    Attributes:
        azure_deployment: An optional Azure OpenAi deployment id. Defaults to the␣
↪model name.
        azure_endpoint: An Azure OpenAi endpoint.
        azure_ad_token: An optional Azure Active Directory token.
        azure_ad_token_provider: An optional Azure Active Directory token provider.
        api_version: An Azure OpenAi API version.
        client: An `openai.AzureOpenAI` client.
    """

    model: str = field(default="tts", kw_only=True, metadata={"serializable": True})
    azure_deployment: str = field(
        kw_only=True,
```

(continues on next page)

(continued from previous page)

```
        default=Factory(lambda self: self.model, takes_self=True),
        metadata={"serializable": True},
    )
    azure_endpoint: str = field(kw_only=True, metadata={"serializable": True})
    azure_ad_token: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": False})
    azure_ad_token_provider: Optional[Callable[[], str]] = field(
        kw_only=True,
        default=None,
        metadata={"serializable": False},
    )
    api_version: str = field(default="2024-07-01-preview", kw_only=True, metadata={
↪"serializable": True})
    _client: Optional[openai.AzureOpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.AzureOpenAI:
        return openai.AzureOpenAI(
            organization=self.organization,
            api_key=self.api_key,
            api_version=self.api_version,
            azure_endpoint=self.azure_endpoint,
            azure_deployment=self.azure_deployment,
            azure_ad_token=self.azure_ad_token,
            azure_ad_token_provider=self.azure_ad_token_provider,
        )
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_version = field(default='2024-07-01-preview', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `azure_ad_token = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_ad_token_provider = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute
- `azure_deployment = field(kw_only=True, default=Factory(lambda self: self.model, takes_self=True), metadata={'serializable': True})` class-attribute instance-attribute
- `azure_endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(default='tts', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/text_to_speech/azure_openai_text_to_speech_driver.py

```
@lazy_property()
def client(self) -> openai.AzureOpenAI:
    return openai.AzureOpenAI(
        organization=self.organization,
        api_key=self.api_key,
        api_version=self.api_version,
        azure_endpoint=self.azure_endpoint,
        azure_deployment=self.azure_deployment,
        azure_ad_token=self.azure_ad_token,
        azure_ad_token_provider=self.azure_ad_token_provider,
    )
```

OpenAiTextToSpeechDriver

Bases: *BaseTextToSpeechDriver* “)

Source Code in griptape/drivers/text_to_speech/openai_text_to_speech_driver.py

```
@define
class OpenAiTextToSpeechDriver(BaseTextToSpeechDriver):
    model: str = field(default="tts-1", kw_only=True, metadata={"serializable": True})
    voice: Literal["alloy", "echo", "fable", "onyx", "nova", "shimmer"] = field(
        default="alloy",
        kw_only=True,
        metadata={"serializable": True},
    )
    format: Literal["mp3", "opus", "aac", "flac"] = field(default="mp3", kw_only=True,
↪ metadata={"serializable": True})
    api_type: Optional[str] = field(default=openai.api_type, kw_only=True)
    api_version: Optional[str] = field(default=openai.api_version, kw_only=True,␣
↪metadata={"serializable": True})
    base_url: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    api_key: Optional[str] = field(default=None, kw_only=True)
    organization: Optional[str] = field(default=openai.organization, kw_only=True,␣
↪metadata={"serializable": True})
    _client: Optional[openai.OpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.OpenAI:
        return openai.OpenAI(
            api_key=self.api_key,
            base_url=self.base_url,
            organization=self.organization,
        )

    def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
        response = self.client.audio.speech.create(
            input=". ".join(prompts),
            voice=self.voice,
            model=self.model,
```

(continues on next page)

(continued from previous page)

```
            response_format=self.format,
        )

        return AudioArtifact(value=response.content, format=self.format)
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True)` class-attribute instance-attribute
- `api_type = field(default=openai.api_type, kw_only=True)` class-attribute instance-attribute
- `api_version = field(default=openai.api_version, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `base_url = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `format = field(default='mp3', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(default='tts-1', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `organization = field(default=openai.organization, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `voice = field(default='alloy', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/text_to_speech/openai_text_to_speech_driver.py

```
@lazy_property()
def client(self) -> openai.OpenAI:
    return openai.OpenAI(
        api_key=self.api_key,
        base_url=self.base_url,
        organization=self.organization,
    )
```

try_text_to_audio(prompts)

Source Code in griptape/drivers/text_to_speech/openai_text_to_speech_driver.py

```
def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
    response = self.client.audio.speech.create(
        input=". ".join(prompts),
        voice=self.voice,
        model=self.model,
        response_format=self.format,
    )

    return AudioArtifact(value=response.content, format=self.format)
```

openai_text_to_speech_driver

Bases: *BaseTextToSpeechDriver* “)

Source Code in griptape/drivers/text_to_speech/openai_text_to_speech_driver.py

```
@define
class OpenAiTextToSpeechDriver(BaseTextToSpeechDriver):
    model: str = field(default="tts-1", kw_only=True, metadata={"serializable": True})
    voice: Literal["alloy", "echo", "fable", "onyx", "nova", "shimmer"] = field(
        default="alloy",
        kw_only=True,
        metadata={"serializable": True},
    )
    format: Literal["mp3", "opus", "aac", "flac"] = field(default="mp3", kw_only=True,
↪ metadata={"serializable": True})
    api_type: Optional[str] = field(default=openai.api_type, kw_only=True)
    api_version: Optional[str] = field(default=openai.api_version, kw_only=True,␣
↪metadata={"serializable": True})
    base_url: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    api_key: Optional[str] = field(default=None, kw_only=True)
    organization: Optional[str] = field(default=openai.organization, kw_only=True,␣
↪metadata={"serializable": True})
    _client: Optional[openai.OpenAI] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> openai.OpenAI:
        return openai.OpenAI(
            api_key=self.api_key,
            base_url=self.base_url,
            organization=self.organization,
        )

    def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
        response = self.client.audio.speech.create(
            input=". ".join(prompts),
            voice=self.voice,
            model=self.model,
            response_format=self.format,
        )

        return AudioArtifact(value=response.content, format=self.format)
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True)` class-attribute instance-attribute
- `api_type = field(default=openai.api_type, kw_only=True)` class-attribute instance-attribute
- `api_version = field(default=openai.api_version, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `base_url = field(default=None, kw_only=True, metadata={'serializable': True})`

class-attribute instance-attribute

- `format = field(default='mp3', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(default='tts-1', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `organization = field(default=openai.organization, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `voice = field(default='alloy', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/text_to_speech/openai_text_to_speech_driver.py

```
@lazy_property()
def client(self) -> openai.OpenAI:
    return openai.OpenAI(
        api_key=self.api_key,
        base_url=self.base_url,
        organization=self.organization,
    )
```

try_text_to_audio(prompts)

Source Code in griptape/drivers/text_to_speech/openai_text_to_speech_driver.py

```
def try_text_to_audio(self, prompts: list[str]) -> AudioArtifact:
    response = self.client.audio.speech.create(
        input=". ".join(prompts),
        voice=self.voice,
        model=self.model,
        response_format=self.format,
    )

    return AudioArtifact(value=response.content, format=self.format)
```

vector

- `__all__ = ['BaseVectorStoreDriver']` module-attribute

Bases: *SerializableMixin* “) , *FuturesExecutorMixin* ”) , ABC

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
@define
class BaseVectorStoreDriver(SerializableMixin, FuturesExecutorMixin, ABC):
    DEFAULT_QUERY_COUNT = 5

    @define
    class Entry(SerializableMixin):
        id: str = field(metadata={"serializable": True})
        vector: Optional[list[float]] = field(default=None, metadata={"serializable":␣
```

(continues on next page)

(continued from previous page)

```
↪True})
        score: Optional[float] = field(default=None, metadata={"serializable": True})
        meta: Optional[dict] = field(default=None, metadata={"serializable": True})
        namespace: Optional[str] = field(default=None, metadata={"serializable": True}
↪)

        def to_artifact(self) -> BaseArtifact:
            return BaseArtifact.from_json(self.meta["artifact"])  # pyright:␣
↪ignore[reportOptionalSubscript]

    embedding_driver: BaseEmbeddingDriver = field(kw_only=True, metadata={
↪"serializable": True})

    def upsert_text_artifacts(
        self,
        artifacts: list[TextArtifact] | dict[str, list[TextArtifact]],
        *,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> list[str] | dict[str, list[str]]:
        warnings.warn(
            "`BaseVectorStoreDriver.upsert_text_artifacts` is deprecated and will be␣
↪removed in a future release. `BaseVectorStoreDriver.upsert_collection` is a drop-in␣
↪replacement.",
            DeprecationWarning,
            stacklevel=2,
        )
        return self.upsert_collection(artifacts, meta=meta, **kwargs)

    def upsert_text_artifact(
        self,
        artifact: TextArtifact,
        *,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        vector_id: Optional[str] = None,
        **kwargs,
    ) -> str:
        warnings.warn(
            "`BaseVectorStoreDriver.upsert_text_artifacts` is deprecated and will be␣
↪removed in a future release. `BaseVectorStoreDriver.upsert` is a drop-in␣
↪replacement.",
            DeprecationWarning,
            stacklevel=2,
        )
        return self.upsert(artifact, namespace=namespace, meta=meta, vector_id=vector_
↪id, **kwargs)

    def upsert_text(
        self,
        string: str,
        *,
```

(continues on next page)

(continued from previous page)

```
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        vector_id: Optional[str] = None,
        **kwargs,
    ) -> str:
        warnings.warn(
            "`BaseVectorStoreDriver.upsert_text` is deprecated and will be removed in␣
↪a future release. `BaseVectorStoreDriver.upsert` is a drop-in replacement.",
            DeprecationWarning,
            stacklevel=2,
        )
        return self.upsert(string, namespace=namespace, meta=meta, vector_id=vector_
↪id, **kwargs)

    @overload
    def upsert_collection(
        self,
        artifacts: list[TextArtifact] | list[ImageArtifact],
        *,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> list[str]: ...

    @overload
    def upsert_collection(
        self,
        artifacts: dict[str, list[TextArtifact]] | dict[str, list[ImageArtifact]],
        *,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> dict[str, list[str]]: ...

    def upsert_collection(
        self,
        artifacts: list[TextArtifact]
        | list[ImageArtifact]
        | dict[str, list[TextArtifact]]
        | dict[str, list[ImageArtifact]],
        *,
        meta: Optional[dict] = None,
        **kwargs,
    ):
        with self.create_futures_executor() as futures_executor:
            if isinstance(artifacts, list):
                return utils.execute_futures_list(
                    [
                        futures_executor.submit(with_contextvars(self.upsert), a,␣
↪namespace=None, meta=meta, **kwargs)
                        for a in artifacts
                    ],
                )
            futures_dict = {}
```

(continues on next page)

(continued from previous page)

```
            for namespace, artifact_list in artifacts.items():
                for a in artifact_list:
                    if not futures_dict.get(namespace):
                        futures_dict[namespace] = []

                    futures_dict[namespace].append(
                        futures_executor.submit(
                            with_contextvars(self.upsert), a, namespace=namespace,␣
↪meta=meta, **kwargs
                        )
                    )

            return utils.execute_futures_list_dict(futures_dict)

    def upsert(
        self,
        value: str | TextArtifact | ImageArtifact,
        *,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        vector_id: Optional[str] = None,
        **kwargs,
    ) -> str:
        artifact = TextArtifact(value) if isinstance(value, str) else value

        meta = {} if meta is None else meta

        if vector_id is None:
            value = artifact.to_text() if artifact.reference is None else artifact.to_
↪text() + str(artifact.reference)
            vector_id = self._get_default_vector_id(value)

        if self.does_entry_exist(vector_id, namespace=namespace):
            return vector_id
        meta = {**meta, "artifact": artifact.to_json()}

        vector = self.embedding_driver.embed(artifact, vector_operation="upsert")

        return self.upsert_vector(vector, vector_id=vector_id, namespace=namespace,␣
↪meta=meta, **kwargs)

    def does_entry_exist(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪bool:
        try:
            return self.load_entry(vector_id, namespace=namespace) is not None
        except Exception:
            return False

    def load_artifacts(self, *, namespace: Optional[str] = None) -> ListArtifact:
        result = self.load_entries(namespace=namespace)
        artifacts = [r.to_artifact() for r in result]
```

(continues on next page)

(continued from previous page)

```
        return ListArtifact([a for a in artifacts if isinstance(a, TextArtifact)])

    @abstractmethod
    def delete_vector(self, vector_id: str) -> None: ...

    @abstractmethod
    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str: ...

    @abstractmethod
    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[Entry]: ...

    @abstractmethod
    def load_entries(self, *, namespace: Optional[str] = None) -> list[Entry]: ...

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[Entry]:
        # TODO: Mark as abstract method for griptape 2.0
        raise NotImplementedError(f"{self.__class__.__name__} does not support vector␣
↪query.")

    def query(
        self,
        query: str | TextArtifact | ImageArtifact,
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[Entry]:
        try:
            vector = self.embedding_driver.embed(query, vector_operation="query")
        except ValueError as e:
            raise ValueError(
                "The Embedding Driver, %s, used by the Vector Store does not support␣
↪embedding the %s type."
```

(continues on next page)

(continued from previous page)

```
                "To resolve, provide an Embedding Driver that supports this type.",
                self.embedding_driver.__class__.__name__,
                type(query),
            ) from e
        return self.query_vector(vector, count=count, namespace=namespace, include_
↪vectors=include_vectors, **kwargs)

    def _get_default_vector_id(self, value: str) -> str:
        return str(uuid.uuid5(uuid.NAMESPACE_OID, value))
```

- `DEFAULT_QUERY_COUNT = 5` class-attribute instance-attribute
- `embedding_driver = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

Entry

Bases: *SerializableMixin* “)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
@define
class Entry(SerializableMixin):
    id: str = field(metadata={"serializable": True})
    vector: Optional[list[float]] = field(default=None, metadata={"serializable":␣
↪True})
    score: Optional[float] = field(default=None, metadata={"serializable": True})
    meta: Optional[dict] = field(default=None, metadata={"serializable": True})
    namespace: Optional[str] = field(default=None, metadata={"serializable": True})

    def to_artifact(self) -> BaseArtifact:
        return BaseArtifact.from_json(self.meta["artifact"])  # pyright:␣
↪ignore[reportOptionalSubscript]
```

- `id = field(metadata={'serializable': True})` class-attribute instance-attribute
- `meta = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `namespace = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `score = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `vector = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

_get_default_vector_id(value)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def _get_default_vector_id(self, value: str) -> str:
    return str(uuid.uuid5(uuid.NAMESPACE_OID, value))
```

delete_vector(vector_id)abstractmethod

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
@abstractmethod
def delete_vector(self, vector_id: str) -> None: ...
```

does_entry_exist(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def does_entry_exist(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪bool:
    try:
        return self.load_entry(vector_id, namespace=namespace) is not None
    except Exception:
        return False
```

load_artifacts(*, namespace=None)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def load_artifacts(self, *, namespace: Optional[str] = None) -> ListArtifact:
    result = self.load_entries(namespace=namespace)
    artifacts = [r.to_artifact() for r in result]

    return ListArtifact([a for a in artifacts if isinstance(a, TextArtifact)])
```

load_entries(*, namespace=None)abstractmethod

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
@abstractmethod
def load_entries(self, *, namespace: Optional[str] = None) -> list[Entry]: ...
```

load_entry(vector_id, *, namespace=None)abstractmethod

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
@abstractmethod
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[Entry]: ...
```

query(query, *, count=None, namespace=None, include_vectors=False, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def query(
    self,
    query: str | TextArtifact | ImageArtifact,
    *,
    count: Optional[int] = None,
```

(continues on next page)

(continued from previous page)

```
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[Entry]:
    try:
        vector = self.embedding_driver.embed(query, vector_operation="query")
    except ValueError as e:
        raise ValueError(
            "The Embedding Driver, %s, used by the Vector Store does not support␣
↪embedding the %s type."
            "To resolve, provide an Embedding Driver that supports this type.",
            self.embedding_driver.__class__.__name__,
            type(query),
        ) from e
    return self.query_vector(vector, count=count, namespace=namespace, include_
↪vectors=include_vectors, **kwargs)
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[Entry]:
    # TODO: Mark as abstract method for griptape 2.0
    raise NotImplementedError(f"{self.__class__.__name__} does not support vector␣
↪query.")
```

upsert(value, *, namespace=None, meta=None, vector_id=None, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def upsert(
    self,
    value: str | TextArtifact | ImageArtifact,
    *,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    vector_id: Optional[str] = None,
    **kwargs,
) -> str:
    artifact = TextArtifact(value) if isinstance(value, str) else value

    meta = {} if meta is None else meta
```

(continues on next page)

(continued from previous page)

```
    if vector_id is None:
        value = artifact.to_text() if artifact.reference is None else artifact.to_
↪text() + str(artifact.reference)
        vector_id = self._get_default_vector_id(value)

    if self.does_entry_exist(vector_id, namespace=namespace):
        return vector_id
    meta = {**meta, "artifact": artifact.to_json()}

    vector = self.embedding_driver.embed(artifact, vector_operation="upsert")

    return self.upsert_vector(vector, vector_id=vector_id, namespace=namespace,␣
↪meta=meta, **kwargs)
```

upsert_collection(artifacts, *, meta=None, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def upsert_collection(
    self,
    artifacts: list[TextArtifact]
    | list[ImageArtifact]
    | dict[str, list[TextArtifact]]
    | dict[str, list[ImageArtifact]],
    *,
    meta: Optional[dict] = None,
    **kwargs,
):
    with self.create_futures_executor() as futures_executor:
        if isinstance(artifacts, list):
            return utils.execute_futures_list(
                [
                    futures_executor.submit(with_contextvars(self.upsert), a,␣
↪namespace=None, meta=meta, **kwargs)
                    for a in artifacts
                ],
            )
        futures_dict = {}

        for namespace, artifact_list in artifacts.items():
            for a in artifact_list:
                if not futures_dict.get(namespace):
                    futures_dict[namespace] = []

                futures_dict[namespace].append(
                    futures_executor.submit(
                        with_contextvars(self.upsert), a, namespace=namespace,␣
↪meta=meta, **kwargs
                    )
                )

        return utils.execute_futures_list_dict(futures_dict)
```

upsert_text(string, *, namespace=None, meta=None, vector_id=None, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def upsert_text(
    self,
    string: str,
    *,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    vector_id: Optional[str] = None,
    **kwargs,
) -> str:
    warnings.warn(
        "`BaseVectorStoreDriver.upsert_text` is deprecated and will be removed in a␣
↪future release. `BaseVectorStoreDriver.upsert` is a drop-in replacement.",
        DeprecationWarning,
        stacklevel=2,
    )
    return self.upsert(string, namespace=namespace, meta=meta, vector_id=vector_id,␣
↪**kwargs)
```

upsert_text_artifact(artifact, *, namespace=None, meta=None, vector_id=None, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def upsert_text_artifact(
    self,
    artifact: TextArtifact,
    *,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    vector_id: Optional[str] = None,
    **kwargs,
) -> str:
    warnings.warn(
        "`BaseVectorStoreDriver.upsert_text_artifacts` is deprecated and will be␣
↪removed in a future release. `BaseVectorStoreDriver.upsert` is a drop-in␣
↪replacement.",
        DeprecationWarning,
        stacklevel=2,
    )
    return self.upsert(artifact, namespace=namespace, meta=meta, vector_id=vector_id,␣
↪**kwargs)
```

upsert_text_artifacts(artifacts, *, meta=None, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def upsert_text_artifacts(
    self,
    artifacts: list[TextArtifact] | dict[str, list[TextArtifact]],
    *,
```

(continues on next page)

(continued from previous page)

```
    meta: Optional[dict] = None,
    **kwargs,
) -> list[str] | dict[str, list[str]]:
    warnings.warn(
        "`BaseVectorStoreDriver.upsert_text_artifacts` is deprecated and will be␣
↪removed in a future release. `BaseVectorStoreDriver.upsert_collection` is a drop-in␣
↪replacement.",
        DeprecationWarning,
        stacklevel=2,
    )
    return self.upsert_collection(artifacts, meta=meta, **kwargs)
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)abstractmethod

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
@abstractmethod
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str: ...
```

amazon_opensearch

- `__all__ = ['AmazonOpenSearchVectorStoreDriver']` module-attribute

Bases: *AmazonOpenSearchVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="amazon_opensearch.mdx#griptape.drivers.vector.amazon_opensearch.AmazonOpenSearchVectorStoreDriver.session" title="session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True) class-attribute instance-attribute (griptape.drivers.vector.amazon_opensearch.AmazonOpenSearchVectorStoreDriver.session)"><code class="language-python">	*Session*	The boto3 session to use.
<a href="amazon_opensearch.mdx#griptape.drivers.vector.amazon_opensearch.AmazonOpenSearchVectorStoreDriver.service" title="service = field(default='es', kw_only=True) class-attribute instance-attribute (griptape.drivers.vector.amazon_opensearch.AmazonOpenSearchVectorStoreDriver.service)"><code class="language-python">	*str*	Service name for AWS Signature v4. Values can be 'es' or 'aoss' for for OpenSearch Serverless. Defaults to 'es'.

continues on next page

Table 218 – continued from previous page

Name	Type	Description
<a href="amazon_opensearch.mdx#griptape.drivers.vector.amazon_opensearch.AmazonOpenSearchVectorStoreDriver.http_auth" title="http_auth = field(default=Factory(lambda self: import_optional_dependency('opensearchpy').AWSerAuth(self.session.get_credentials(), self.session.region_name, self.service), takes_self=True)) class-attribute instance-attribute (griptape.drivers.vector.amazon_opensearch.AmazonOpenSearchVectorStoreDriver.http_auth)"><class="language-python">	*str I tuple[str, str]*	The HTTP authentication credentials to use. Defaults to using credentials in the boto3 session.
<a href="opensearch.mdx#griptape.drivers.vector.opensearch.OpenSearchVectorStoreDriver.client" title="client() (griptape.drivers.vector.amazon_opensearch.AmazonOpenSearchVectorStoreDriver.client)"><coc class="language-python">	*OpenSearch*	An optional OpenSearch client to use. Defaults to a new client using the host, port, http_auth, use_ssl, and verify_certs attributes.

Source Code in griptape/drivers/vector/amazon_opensearch_vector_store_driver.py

```
@define
class AmazonOpenSearchVectorStoreDriver(OpenSearchVectorStoreDriver):
    """A Vector Store Driver for Amazon OpenSearch.

    Attributes:
        session: The boto3 session to use.
        service: Service name for AWS Signature v4. Values can be es or aoss for for␣
↪OpenSearch Serverless. Defaults to es.
        http_auth: The HTTP authentication credentials to use. Defaults to using␣
↪credentials in the boto3 session.
        client: An optional OpenSearch client to use. Defaults to a new client using␣
↪the host, port, http_auth, use_ssl, and verify_certs attributes.
    """
```

(continues on next page)

(continued from previous page)

```
    session: Session = field(default=Factory(lambda: import_optional_dependency("boto3
↪").Session()), kw_only=True)
    service: str = field(default="es", kw_only=True)
    http_auth: str | tuple[str, str] = field(
        default=Factory(
            lambda self: import_optional_dependency("opensearchpy").AWSV4SignerAuth(
                self.session.get_credentials(),
                self.session.region_name,
                self.service,
            ),
            takes_self=True,
        ),
    )

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        """Inserts or updates a vector in OpenSearch.

        If a vector with the given vector ID already exists, it is updated; otherwise,
↪ a new vector is inserted.
        Metadata associated with the vector can also be provided.
        """
        vector_id = vector_id or str_to_hash(str(vector))
        doc = {"vector": vector, "namespace": namespace, "metadata": meta}
        doc.update(kwargs)
        if self.service == "aoss":
            response = self.client.index(index=self.index_name, body=doc)
        else:
            response = self.client.index(index=self.index_name, id=vector_id,␣
↪body=doc)

        return response["_id"]
```

- `http_auth = field(default=Factory(lambda self: import_optional_dependency('opensearchpy').AWSV4SignerAuth(self.session.get_credentials(), self.session.region_name, self.service), takes_self=True))` class-attribute instance-attribute
- `service = field(default='es', kw_only=True)` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True)` class-attribute instance-attribute

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/amazon_opensearch_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    """Inserts or updates a vector in OpenSearch.

    If a vector with the given vector ID already exists, it is updated; otherwise, a new vector is inserted.
    Metadata associated with the vector can also be provided.
    """
    vector_id = vector_id or str_to_hash(str(vector))
    doc = {"vector": vector, "namespace": namespace, "metadata": meta}
    doc.update(kwargs)
    if self.service == "aoss":
        response = self.client.index(index=self.index_name, body=doc)
    else:
        response = self.client.index(index=self.index_name, id=vector_id, body=doc)

    return response["_id"]
```

amazon_opensearch_vector_store_driver

Bases: *AmazonOpenSearchVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="amazon_opensearch_vector_store_driver.mdx#griptape.drivers.vector.amazon_opensearch_vector_store_driver.AmazonOpenSearchVectorStoreDriver.session" title="session = field(default=Factory(lambda: import_optional_dependency('boto3').Session()), kw_only=True) class-attribute instance-attribute (griptape.drivers.vector.amazon_opensearch_vector_store_driver.AmazonOpenSearchVectorStoreDriver.session)"><code class="language-python">	*Session*	The boto3 session to use.
<a href="amazon_opensearch_vector_store_driver.mdx#griptape.drivers.vector.amazon_opensearch_vector_store_driver.AmazonOpenSearchVectorStoreDriver.service" title="service = field(default='es', kw_only=True) class-attribute instance-attribute (griptape.drivers.vector.amazon_opensearch_vector_store_driver.AmazonOpenSearchVectorStoreDriver.service)"><code class="language-python">	*str*	Service name for AWS Signature v4. Values can be 'es' or 'aoss' for for OpenSearch Serverless. Defaults to 'es'.

continues on next page

Table 219 – continued from previous page

Name	Type	Description
<a href="amazon_opensearch_vector_store_driver.mdx#griptape.drivers.vector.amazon_opensearch_vector_store_driver.AmazonOpenSearchVectorStoreDriver.http_auth" title="http_auth = field(default=Factory(lambda self: import_optional_dependency('opensearchpy').AWSerAuth(self.session.get_credentials(), self.session.region_name, self.service), takes_self=True)) class-attribute instance-attribute (griptape.drivers.vector.amazon_opensearch_vector_store_driver.AmazonOpenSearchVectorStoreDriver.http_auth)"><class="language-python">	*str I tuple[str, str]*	The HTTP authentication credentials to use. Defaults to using credentials in the boto3 session.
<a href="opensearch_vector_store_driver.mdx#griptape.drivers.vector.opensearch_vector_store_driver.OpenSearchVectorStoreDriver.client" title="client() (griptape.drivers.vector.amazon_opensearch_vector_store_driver.AmazonOpenSearchVectorStoreDriver.client)"><coc class="language-python">	*OpenSearch*	An optional OpenSearch client to use. Defaults to a new client using the host, port, http_auth, use_ssl, and verify_certs attributes.

Source Code in griptape/drivers/vector/amazon_opensearch_vector_store_driver.py

```
@define
class AmazonOpenSearchVectorStoreDriver(OpenSearchVectorStoreDriver):
    """A Vector Store Driver for Amazon OpenSearch.

    Attributes:
        session: The boto3 session to use.
        service: Service name for AWS Signature v4. Values can be es or aoss for for␣
```

(continues on next page)

(continued from previous page)

```
↪OpenSearch Serverless. Defaults to es.
        http_auth: The HTTP authentication credentials to use. Defaults to using␣
↪credentials in the boto3 session.
        client: An optional OpenSearch client to use. Defaults to a new client using␣
↪the host, port, http_auth, use_ssl, and verify_certs attributes.
    """

    session: Session = field(default=Factory(lambda: import_optional_dependency("boto3
↪").Session()), kw_only=True)
    service: str = field(default="es", kw_only=True)
    http_auth: str | tuple[str, str] = field(
        default=Factory(
            lambda self: import_optional_dependency("opensearchpy").AWSV4SignerAuth(
                self.session.get_credentials(),
                self.session.region_name,
                self.service,
            ),
            takes_self=True,
        ),
    )

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        """Inserts or updates a vector in OpenSearch.

        If a vector with the given vector ID already exists, it is updated; otherwise,
↪ a new vector is inserted.
        Metadata associated with the vector can also be provided.
        """
        vector_id = vector_id or str_to_hash(str(vector))
        doc = {"vector": vector, "namespace": namespace, "metadata": meta}
        doc.update(kwargs)
        if self.service == "aoss":
            response = self.client.index(index=self.index_name, body=doc)
        else:
            response = self.client.index(index=self.index_name, id=vector_id,␣
↪body=doc)

        return response["_id"]
```

- `http_auth = field(default=Factory(lambda self: import_optional_dependency('opensearchpy').AWSV4SignerAuth(self.session.get_credentials(), self.session.region_name, self.service), takes_self=True))` class-attribute instance-attribute
- `service = field(default='es', kw_only=True)` class-attribute instance-attribute
- `session = field(default=Factory(lambda: import_optional_dependency('boto3').`

`Session()), kw_only=True)` class-attribute instance-attribute

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/amazon_opensearch_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    """Inserts or updates a vector in OpenSearch.

    If a vector with the given vector ID already exists, it is updated; otherwise, a␣
↪new vector is inserted.
    Metadata associated with the vector can also be provided.
    """
    vector_id = vector_id or str_to_hash(str(vector))
    doc = {"vector": vector, "namespace": namespace, "metadata": meta}
    doc.update(kwargs)
    if self.service == "aoss":
        response = self.client.index(index=self.index_name, body=doc)
    else:
        response = self.client.index(index=self.index_name, id=vector_id, body=doc)

    return response["_id"]
```

astradb

- `__all__ = ['AstraDbVectorStoreDriver']` module-attribute

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="base_vector_store_driver.mdx#griptape.drivers.vector.base_vector_store_driver.BaseVectorStoreDriver.embedding_driver" title="embedding_driver = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.astradb.AstraDbVectorStoreDriver.embedding_driver)"><code class="language-python">	<a href="../embedding#griptape.drivers.embedding.BaseEmbeddingDriver" title="BaseEmbeddingDriver (griptape.drivers.embedding.BaseEmbeddingDriver)"><code class="language-python">BaseEmbeddingDriver</cod	a *griptape.drivers.BaseEmbeddingDriver* for embedding computations within the store
<a href="astradb.mdx#griptape.drivers.vector.astradb.AstraDbVectorStoreDriver.api_endpoint" title="api_endpoint = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.astradb.AstraDbVectorStoreDriver.api_endpoint)"><code class="language-python">	*str*	the "API Endpoint" for the Astra DB instance.

continues on next page

Table 220 – continued from previous page

Name	Type	Description
<a href="astradb.mdx#griptape.drivers.vector.astradb.AstraDbVectorStoreDriver.token" title="token = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.vector.astradb.AstraDbVectorStoreDriver.token)"><code class="language-python">	*Optional[str I TokenProvider]*	a Database Token ("AstraCS:...") secret to access Astra DB. An instance of *astrapy.authentication.TokenProvider* is also accepted.
<a href="astradb.mdx#griptape.drivers.vector.astradb.AstraDbVectorStoreDriver.collection_name" title="collection_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.astradb.AstraDbVectorStoreDriver.collection_name)"><code class="language-python">	*str*	the name of the collection on Astra DB. The collection must have been created beforehand, and support vectors with a vector dimension matching the embeddings being used by this driver.
<a href="astradb.mdx#griptape.drivers.vector.astradb.AstraDbVectorStoreDriver.environment" title="environment = field(kw_only=True, default=None, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.astradb.AstraDbVectorStoreDriver.environment)"><code class="language-python">	*Optional[str]*	the environment ("prod", "hcd", ...) hosting the target Data API. It can be omitted for production Astra DB targets. See *astrapy.constants.Environment* for allowed values.

continues on next page

Table 220 – continued from previous page

Name	Type	Description
<a href="astradb.mdx#griptape.drivers.vector.astradb.AstraDbVectorStoreDriver.astra_db_namespace" title="astra_db_namespace = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.astradb.AstraDbVectorStoreDriver.astra_db_namespace)"><code class="language-python">	*Optional[str]*	optional specification of the namespace (in the Astra database) for the data. *Note* : not to be confused with the "namespace" mentioned elsewhere, which is a grouping within this vector store.
<a href="astradb.mdx#griptape.drivers.vector.astradb.AstraDbVectorStoreDriver.caller_name" title="caller_name = field(default='griptape', kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.vector.astradb.AstraDbVectorStoreDriver.caller_name)" class="language-python">	*str*	the name of the caller for the Astra DB client. Defaults to "griptape".
<a href="astradb.mdx#griptape.drivers.vector.astradb.AstraDbVectorStoreDriver.client" title="client() (griptape.drivers.vector.astradb.AstraDbVectorStoreDriver.client)"><coc class="language-python">	*DataAPIClient*	an instance of *astrapy.DataAPIClient* for the Astra DB.

continues on next page

Table 220 – continued from previous page

Name	Type	Description
<a href="astradb.mdx#griptape.drivers.vector.astradb.AstraDbVectorStoreDriver.collection" title="collection() (griptape.drivers.vector.astradb.AstraDbVectorStoreDriver.collection)"><code class="language-python">	*Collection*	an instance of *astrapy.Collection* for the Astra DB.

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
@define
class AstraDbVectorStoreDriver(BaseVectorStoreDriver):
    """A Vector Store Driver for Astra DB.

    Attributes:
        embedding_driver: a `griptape.drivers.BaseEmbeddingDriver` for embedding␣
↪computations within the store
        api_endpoint: the "API Endpoint" for the Astra DB instance.
        token: a Database Token ("AstraCS:...") secret to access Astra DB. An␣
↪instance of `astrapy.authentication.TokenProvider` is also accepted.
        collection_name: the name of the collection on Astra DB. The collection must␣
↪have been created beforehand,
            and support vectors with a vector dimension matching the embeddings being␣
↪used by this driver.
        environment: the environment ("prod", "hcd", ...) hosting the target Data API.
            It can be omitted for production Astra DB targets. See `astrapy.constants.
↪Environment` for allowed values.
        astra_db_namespace: optional specification of the namespace (in the Astra␣
↪database) for the data.
            *Note*: not to be confused with the "namespace" mentioned elsewhere,␣
↪which is a grouping within this vector store.
        caller_name: the name of the caller for the Astra DB client. Defaults to
↪"griptape".
        client: an instance of `astrapy.DataAPIClient` for the Astra DB.
        collection: an instance of `astrapy.Collection` for the Astra DB.
    """

    api_endpoint: str = field(kw_only=True, metadata={"serializable": True})
    token: Optional[str | astrapy.authentication.TokenProvider] = field(
        kw_only=True, default=None, metadata={"serializable": False}
    )
    collection_name: str = field(kw_only=True, metadata={"serializable": True})
    environment: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": True})
    astra_db_namespace: Optional[str] = field(default=None, kw_only=True, metadata={
```

(continues on next page)

(continued from previous page)

```
↪"serializable": True})
    caller_name: str = field(default="griptape", kw_only=True, metadata={"serializable
↪": False})
    _client: Optional[astrapy.DataAPIClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )
    _collection: Optional[astrapy.Collection] = field(
        default=None, kw_only=True, alias="collection", metadata={"serializable":␣
↪False}
    )

    @lazy_property()
    def client(self) -> astrapy.DataAPIClient:
        astrapy = import_optional_dependency("astrapy")

        return astrapy.DataAPIClient(
            callers=[(self.caller_name, None)],
            environment=self.environment or astrapy.utils.unset.UnsetType(),
        )

    @lazy_property()
    def collection(self) -> astrapy.Collection:
        if self.token is None:
            raise ValueError("Astra DB token must be provided.")
        return self.client.get_database(
            self.api_endpoint, token=self.token, keyspace=self.astra_db_namespace
        ).get_collection(self.collection_name)

    def delete_vector(self, vector_id: str) -> None:
        """Delete a vector from Astra DB store.

        The method succeeds regardless of whether a vector with the provided ID
        was actually stored or not in the first place.

        Args:
            vector_id: ID of the vector to delete.
        """
        self.collection.delete_one({"_id": vector_id})

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs: Any,
    ) -> str:
        """Write a vector to the Astra DB store.

        In case the provided ID exists already, an overwrite will take place.
```

(continues on next page)

(continued from previous page)

```
        Args:
            vector: the vector to be upserted.
            vector_id: the ID for the vector to store. If omitted, a server-provided␣
↪new ID will be employed.
            namespace: a namespace (a grouping within the vector store) to assign the␣
↪vector to.
            meta: a metadata dictionary associated to the vector.
            kwargs: additional keyword arguments. Currently none is used: if they are␣
↪passed, they will be ignored with a warning.

        Returns:
            the ID of the written vector (str).
        """
        document = {
            k: v
            for k, v in {"$vector": vector, "_id": vector_id, "keyspace": namespace,
↪"meta": meta}.items()
            if v is not None
        }
        if vector_id is not None:
            self.collection.find_one_and_replace({"_id": vector_id}, document,␣
↪upsert=True)
            return vector_id
        insert_result = self.collection.insert_one(document)
        return insert_result.inserted_id

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        """Load a single vector entry from the Astra DB store given its ID.

        Args:
            vector_id: the ID of the required vector.
            namespace: a namespace, within the vector store, to constrain the search.

        Returns:
            The vector entry (a `BaseVectorStoreDriver.Entry`) if found, otherwise␣
↪None.
        """
        find_filter = {k: v for k, v in {"_id": vector_id, "keyspace": namespace}.
↪items() if v is not None}
        match = self.collection.find_one(filter=find_filter, projection={"*": 1})
        if match is not None:
            return BaseVectorStoreDriver.Entry(
                id=match["_id"], vector=match.get("$vector"), meta=match.get("meta"),␣
↪namespace=match.get("keyspace")
            )
        return None

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Load entries from the Astra DB store.
```

(continues on next page)

(continued from previous page)

```
        Args:
            namespace: a namespace, within the vector store, to constrain the search.

        Returns:
            A list of vector (`BaseVectorStoreDriver.Entry`) entries.
        """
        find_filter: dict[str, str] = {} if namespace is None else {"keyspace":␣
↪namespace}
        return [
            BaseVectorStoreDriver.Entry(
                id=match["_id"], vector=match.get("$vector"), meta=match.get("meta"),␣
↪namespace=match.get("keyspace")
            )
            for match in self.collection.find(filter=find_filter, projection={"*": 1})
        ]

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs: Any,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Run a similarity search on the Astra DB store, based on a vector list.

        Args:
            vector: the vector to be queried.
            count: the maximum number of results to return. If omitted, defaults will␣
↪apply.
            namespace: the namespace to filter results by.
            include_vectors: whether to include vector data in the results.
            kwargs: additional keyword arguments. Currently only the free-form dict␣
↪`filter`
                is recognized (and goes straight to the Data API query);
                others will generate a warning and be ignored.

        Returns:
            A list of vector (`BaseVectorStoreDriver.Entry`) entries,
            with their `score` attribute set to the vector similarity to the query.
        """
        query_filter: Optional[dict[str, Any]] = kwargs.get("filter")
        find_filter_ns: dict[str, Any] = {} if namespace is None else {"keyspace":␣
↪namespace}
        find_filter = {**(query_filter or {}), **find_filter_ns}
        find_projection: Optional[dict[str, int]] = {"*": 1} if include_vectors else␣
↪None
        ann_limit = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
        matches = self.collection.find(
            filter=find_filter,
            sort={"$vector": vector},
```

(continues on next page)

(continued from previous page)

```
            limit=ann_limit,
            projection=find_projection,
            include_similarity=True,
        )
        return [
            BaseVectorStoreDriver.Entry(
                id=match["_id"],
                vector=match.get("$vector"),
                score=match["$similarity"],
                meta=match.get("meta"),
                namespace=match.get("keyspace"),
            )
            for match in matches
        ]
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `_collection = field(default=None, kw_only=True, alias='collection', metadata={'serializable': False})` class-attribute instance-attribute
- `api_endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `astra_db_namespace = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `caller_name = field(default='griptape', kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `collection_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `environment = field(kw_only=True, default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `token = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
@lazy_property()
def client(self) -> astrapy.DataAPIClient:
    astrapy = import_optional_dependency("astrapy")

    return astrapy.DataAPIClient(
        callers=[(self.caller_name, None)],
        environment=self.environment or astrapy.utils.unset.UnsetType(),
    )
```

collection()

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
@lazy_property()
def collection(self) -> astrapy.Collection:
    if self.token is None:
        raise ValueError("Astra DB token must be provided.")
    return self.client.get_database(
        self.api_endpoint, token=self.token, keyspace=self.astra_db_namespace
    ).get_collection(self.collection_name)
```

delete_vector(vector_id)

Delete a vector from Astra DB store.

The method succeeds regardless of whether a vector with the provided ID was actually stored or not in the first place.

Parameters

Name	Type	Description	Default
vector_id	*str*	ID of the vector to delete. 	*required*

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> None:
    """Delete a vector from Astra DB store.

    The method succeeds regardless of whether a vector with the provided ID
    was actually stored or not in the first place.

    Args:
        vector_id: ID of the vector to delete.
    """
    self.collection.delete_one({"_id": vector_id})
```

load_entries(*, namespace=None)

Load entries from the Astra DB store.

Parameters

Name	Type	Description	Default
namespace	*Optional[str]*	a namespace, within the vector store, to constrain the search. 	*None*

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	A list of vector (*BaseVectorStoreDriver.Entry*) entries.

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Load entries from the Astra DB store.

    Args:
        namespace: a namespace, within the vector store, to constrain the search.

    Returns:
        A list of vector (`BaseVectorStoreDriver.Entry`) entries.
    """
    find_filter: dict[str, str] = {} if namespace is None else {"keyspace": namespace}
    return [
        BaseVectorStoreDriver.Entry(
            id=match["_id"], vector=match.get("$vector"), meta=match.get("meta"),␣
↪namespace=match.get("keyspace")
        )
        for match in self.collection.find(filter=find_filter, projection={"*": 1})
    ]
```

load_entry(vector_id, *, namespace=None)

Load a single vector entry from the Astra DB store given its ID.

Parameters

Name	Type	Description	Default
vector_id	*str*	the ID of the required vector. 	*required*
namespace	*Optional[str]*	a namespace, within the vector store, to constrain the search. 	*None*

Returns

Type	Description
<a href="./#griptape.drivers.vector.BaseVectorStoreDriver.Entry" title="Entry (griptape.drivers.vector.BaseVectorStoreDriver.Entry)"><code class="language-python">C	The vector entry (a *BaseVectorStoreDriver.Entry*) if found, otherwise None.

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    """Load a single vector entry from the Astra DB store given its ID.

    Args:
        vector_id: the ID of the required vector.
        namespace: a namespace, within the vector store, to constrain the search.

    Returns:
        The vector entry (a `BaseVectorStoreDriver.Entry`) if found, otherwise None.
    """
    find_filter = {k: v for k, v in {"_id": vector_id, "keyspace": namespace}.items()␣
↪if v is not None}
    match = self.collection.find_one(filter=find_filter, projection={"*": 1})
    if match is not None:
        return BaseVectorStoreDriver.Entry(
            id=match["_id"], vector=match.get("$vector"), meta=match.get("meta"),␣
↪namespace=match.get("keyspace")
        )
    return None
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs)

Run a similarity search on the Astra DB store, based on a vector list.

Parameters

Name	Type	Description	Default
vector	<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	the vector to be queried. 	*required*

continues on next page

Table 226 – continued from previous page

Name	Type	Description	Default
count	*Optional[int]*	the maximum number of results to return. If omitted, defaults will apply. 	*None*
namespace	*Optional[str]*	the namespace to filter results by. 	*None*
include_vectors	*bool*	whether to include vector data in the results. 	*False*
kwargs	*Any*	additional keyword arguments. Currently only the free-form dict *filter* is recognized (and goes straight to the Data API query); others will generate a warning and be ignored. 	*{}*

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	A list of vector (*BaseVectorStoreDriver.Entry*) entries,
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	with their *score* attribute set to the vector similarity to the query.

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs: Any,
) -> list[BaseVectorStoreDriver.Entry]:
    """Run a similarity search on the Astra DB store, based on a vector list.

    Args:
        vector: the vector to be queried.
```

(continues on next page)

(continued from previous page)

```
        count: the maximum number of results to return. If omitted, defaults will␣
↪apply.
        namespace: the namespace to filter results by.
        include_vectors: whether to include vector data in the results.
        kwargs: additional keyword arguments. Currently only the free-form dict␣
↪`filter`
            is recognized (and goes straight to the Data API query);
            others will generate a warning and be ignored.

    Returns:
        A list of vector (`BaseVectorStoreDriver.Entry`) entries,
        with their `score` attribute set to the vector similarity to the query.
    """
    query_filter: Optional[dict[str, Any]] = kwargs.get("filter")
    find_filter_ns: dict[str, Any] = {} if namespace is None else {"keyspace":␣
↪namespace}
    find_filter = {**(query_filter or {}), **find_filter_ns}
    find_projection: Optional[dict[str, int]] = {"*": 1} if include_vectors else None
    ann_limit = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
    matches = self.collection.find(
        filter=find_filter,
        sort={"$vector": vector},
        limit=ann_limit,
        projection=find_projection,
        include_similarity=True,
    )
    return [
        BaseVectorStoreDriver.Entry(
            id=match["_id"],
            vector=match.get("$vector"),
            score=match["$similarity"],
            meta=match.get("meta"),
            namespace=match.get("keyspace"),
        )
        for match in matches
    ]
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Write a vector to the Astra DB store.

In case the provided ID exists already, an overwrite will take place.

Parameters

Name	Type	Description	Default
vector	<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	the vector to be upserted. 	*required*
vector_id	*Optional[str]*	the ID for the vector to store. If omitted, a server-provided new ID will be employed. 	*None*
namespace	*Optional[str]*	a namespace (a grouping within the vector store) to assign the vector to. 	*None*
meta	*Optional[dict]*	a metadata dictionary associated to the vector. 	*None*
kwargs	*Any*	additional keyword arguments. Currently none is used: if they are passed, they will be ignored with a warning. 	*{}*

Returns

Type	Description
str	the ID of the written vector (str).

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs: Any,
) -> str:
    """Write a vector to the Astra DB store.

    In case the provided ID exists already, an overwrite will take place.

    Args:
        vector: the vector to be upserted.
        vector_id: the ID for the vector to store. If omitted, a server-provided new␣
```

(continues on next page)

(continued from previous page)

```
↪ID will be employed.
        namespace: a namespace (a grouping within the vector store) to assign the␣
↪vector to.
        meta: a metadata dictionary associated to the vector.
        kwargs: additional keyword arguments. Currently none is used: if they are␣
↪passed, they will be ignored with a warning.

    Returns:
        the ID of the written vector (str).
    """
    document = {
        k: v
        for k, v in {"$vector": vector, "_id": vector_id, "keyspace": namespace, "meta
↪": meta}.items()
        if v is not None
    }
    if vector_id is not None:
        self.collection.find_one_and_replace({"_id": vector_id}, document,␣
↪upsert=True)
        return vector_id
    insert_result = self.collection.insert_one(document)
    return insert_result.inserted_id
```

astradb_vector_store_driver

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="./#griptape.drivers.vector.BaseVectorStoreDriver.embedding_driver" title="embedding_driver = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.astradb_vector_store_driver.AstraDbVectorStoreDriver.embedding_driver)"><code class="language-python">	<a href="../embedding#griptape.drivers.embedding.BaseEmbeddingDriver" title="BaseEmbeddingDriver (griptape.drivers.embedding.BaseEmbeddingDriver)"><code class="language-python">BaseEmbeddingDriver</cod	a *griptape.drivers.BaseEmbeddingDriver* for embedding computations within the store

continues on next page

Table 230 – continued from previous page

Name	Type	Description
<a href="astradb_vector_store_driver.mdx#griptape.drivers.vector.astradb_vector_store_driver.AstraDbVectorStoreDriver.api_endpoint" title="api_endpoint = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.astradb_vector_store_driver.AstraDbVectorStoreDriver.api_endpoint)"><code class="language-python">	*str*	the "API Endpoint" for the Astra DB instance.
<a href="astradb_vector_store_driver.mdx#griptape.drivers.vector.astradb_vector_store_driver.AstraDbVectorStoreDriver.token" title="token = field(kw_only=True, default=None, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.vector.astradb_vector_store_driver.AstraDbVectorStoreDriver.token)"><code class="language-python">	*Optional[str I TokenProvider]*	a Database Token ("AstraCS:...") secret to access Astra DB. An instance of *astrapy.authentication.TokenProvider* is also accepted.

continues on next page

Table 230 – continued from previous page

Name	Type	Description
<a href="astradb_vector_store_driver.mdx#griptape.drivers.vector.astradb_vector_store_driver.AstraDbVectorStoreDriver.collection_name" title="collection_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.astradb_vector_store_driver.AstraDbVectorStoreDriver.collection_name)"><code class="language-python">	*str*	the name of the collection on Astra DB. The collection must have been created beforehand, and support vectors with a vector dimension matching the embeddings being used by this driver.
<a href="astradb_vector_store_driver.mdx#griptape.drivers.vector.astradb_vector_store_driver.AstraDbVectorStoreDriver.environment" title="environment = field(kw_only=True, default=None, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.astradb_vector_store_driver.AstraDbVectorStoreDriver.environment)"><code class="language-python">	*Optional[str]*	the environment ("prod", "hcd", ...) hosting the target Data API. It can be omitted for production Astra DB targets. See *astrapy.constants.Environment* for allowed values.

continues on next page

Table 230 – continued from previous page

Name	Type	Description
<a href="astradb_vector_store_driver.mdx#griptape.drivers.vector.astradb_vector_store_driver.AstraDbVectorStoreDriver.astra_db_namespace" title="astra_db_namespace = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.astradb_vector_store_driver.AstraDbVectorStoreDriver.astra_db_namespace)"><code class="language-python">	*Optional[str]*	optional specification of the namespace (in the Astra database) for the data. *Note* : not to be confused with the "namespace" mentioned elsewhere, which is a grouping within this vector store.
<a href="astradb_vector_store_driver.mdx#griptape.drivers.vector.astradb_vector_store_driver.AstraDbVectorStoreDriver.caller_name" title="caller_name = field(default='griptape', kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.vector.astradb_vector_store_driver.AstraDbVectorStoreDriver.caller_name)" class="language-python">	*str*	the name of the caller for the Astra DB client. Defaults to "griptape".

continues on next page

Table 230 – continued from previous page

Name	Type	Description
<a href="astradb_vector_store_driver.mdx#griptape.drivers.vector.astradb_vector_store_driver.AstraDbVectorStoreDriver.client" title="client() (griptape.drivers.vector.astradb_vector_store_driver.AstraDbVectorStoreDriver.client)"><code class="language-python">	*DataAPIClient*	an instance of *astrapy.DataAPIClient* for the Astra DB.
<a href="astradb_vector_store_driver.mdx#griptape.drivers.vector.astradb_vector_store_driver.AstraDbVectorStoreDriver.collection" title="collection() (griptape.drivers.vector.astradb_vector_store_driver.AstraDbVectorStoreDriver.collection)"><code class="language-python">	*Collection*	an instance of *astrapy.Collection* for the Astra DB.

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
@define
class AstraDbVectorStoreDriver(BaseVectorStoreDriver):
    """A Vector Store Driver for Astra DB.

    Attributes:
        embedding_driver: a `griptape.drivers.BaseEmbeddingDriver` for embedding␣
↪computations within the store
        api_endpoint: the "API Endpoint" for the Astra DB instance.
        token: a Database Token ("AstraCS:...") secret to access Astra DB. An␣
↪instance of `astrapy.authentication.TokenProvider` is also accepted.
        collection_name: the name of the collection on Astra DB. The collection must␣
↪have been created beforehand,
            and support vectors with a vector dimension matching the embeddings being␣
↪used by this driver.
        environment: the environment ("prod", "hcd", ...) hosting the target Data API.
            It can be omitted for production Astra DB targets. See `astrapy.constants.
↪Environment` for allowed values.
        astra_db_namespace: optional specification of the namespace (in the Astra␣
↪database) for the data.
```

(continues on next page)

(continued from previous page)

```
            *Note*: not to be confused with the "namespace" mentioned elsewhere,␣
↪which is a grouping within this vector store.
        caller_name: the name of the caller for the Astra DB client. Defaults to
↪"griptape".
        client: an instance of `astrapy.DataAPIClient` for the Astra DB.
        collection: an instance of `astrapy.Collection` for the Astra DB.
    """

    api_endpoint: str = field(kw_only=True, metadata={"serializable": True})
    token: Optional[str | astrapy.authentication.TokenProvider] = field(
        kw_only=True, default=None, metadata={"serializable": False}
    )
    collection_name: str = field(kw_only=True, metadata={"serializable": True})
    environment: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": True})
    astra_db_namespace: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    caller_name: str = field(default="griptape", kw_only=True, metadata={"serializable
↪": False})
    _client: Optional[astrapy.DataAPIClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )
    _collection: Optional[astrapy.Collection] = field(
        default=None, kw_only=True, alias="collection", metadata={"serializable":␣
↪False}
    )

    @lazy_property()
    def client(self) -> astrapy.DataAPIClient:
        astrapy = import_optional_dependency("astrapy")

        return astrapy.DataAPIClient(
            callers=[(self.caller_name, None)],
            environment=self.environment or astrapy.utils.unset.UnsetType(),
        )

    @lazy_property()
    def collection(self) -> astrapy.Collection:
        if self.token is None:
            raise ValueError("Astra DB token must be provided.")
        return self.client.get_database(
            self.api_endpoint, token=self.token, keyspace=self.astra_db_namespace
        ).get_collection(self.collection_name)

    def delete_vector(self, vector_id: str) -> None:
        """Delete a vector from Astra DB store.

        The method succeeds regardless of whether a vector with the provided ID
        was actually stored or not in the first place.

        Args:
            vector_id: ID of the vector to delete.
```

(continues on next page)

(continued from previous page)

```
        """
        self.collection.delete_one({"_id": vector_id})

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs: Any,
    ) -> str:
        """Write a vector to the Astra DB store.

        In case the provided ID exists already, an overwrite will take place.

        Args:
            vector: the vector to be upserted.
            vector_id: the ID for the vector to store. If omitted, a server-provided␣
↪new ID will be employed.
            namespace: a namespace (a grouping within the vector store) to assign the␣
↪vector to.
            meta: a metadata dictionary associated to the vector.
            kwargs: additional keyword arguments. Currently none is used: if they are␣
↪passed, they will be ignored with a warning.

        Returns:
            the ID of the written vector (str).
        """
        document = {
            k: v
            for k, v in {"$vector": vector, "_id": vector_id, "keyspace": namespace,␣
↪"meta": meta}.items()
            if v is not None
        }
        if vector_id is not None:
            self.collection.find_one_and_replace({"_id": vector_id}, document,␣
↪upsert=True)
            return vector_id
        insert_result = self.collection.insert_one(document)
        return insert_result.inserted_id

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        """Load a single vector entry from the Astra DB store given its ID.

        Args:
            vector_id: the ID of the required vector.
            namespace: a namespace, within the vector store, to constrain the search.

        Returns:
            The vector entry (a `BaseVectorStoreDriver.Entry`) if found, otherwise␣
```

(continues on next page)

(continued from previous page)

```
↪None.
        """
        find_filter = {k: v for k, v in {"_id": vector_id, "keyspace": namespace}.
↪items() if v is not None}
        match = self.collection.find_one(filter=find_filter, projection={"*": 1})
        if match is not None:
            return BaseVectorStoreDriver.Entry(
                id=match["_id"], vector=match.get("$vector"), meta=match.get("meta"),␣
↪namespace=match.get("keyspace")
            )
        return None

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Load entries from the Astra DB store.

        Args:
            namespace: a namespace, within the vector store, to constrain the search.

        Returns:
            A list of vector (`BaseVectorStoreDriver.Entry`) entries.
        """
        find_filter: dict[str, str] = {} if namespace is None else {"keyspace":␣
↪namespace}
        return [
            BaseVectorStoreDriver.Entry(
                id=match["_id"], vector=match.get("$vector"), meta=match.get("meta"),␣
↪namespace=match.get("keyspace")
            )
            for match in self.collection.find(filter=find_filter, projection={"*": 1})
        ]

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs: Any,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Run a similarity search on the Astra DB store, based on a vector list.

        Args:
            vector: the vector to be queried.
            count: the maximum number of results to return. If omitted, defaults will␣
↪apply.
            namespace: the namespace to filter results by.
            include_vectors: whether to include vector data in the results.
            kwargs: additional keyword arguments. Currently only the free-form dict␣
↪`filter`
                is recognized (and goes straight to the Data API query);
```

(continues on next page)

(continued from previous page)

```
                others will generate a warning and be ignored.

        Returns:
            A list of vector (`BaseVectorStoreDriver.Entry`) entries,
            with their `score` attribute set to the vector similarity to the query.
        """
        query_filter: Optional[dict[str, Any]] = kwargs.get("filter")
        find_filter_ns: dict[str, Any] = {} if namespace is None else {"keyspace":␣
↪namespace}
        find_filter = {**(query_filter or {}), **find_filter_ns}
        find_projection: Optional[dict[str, int]] = {"*": 1} if include_vectors else␣
↪None
        ann_limit = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
        matches = self.collection.find(
            filter=find_filter,
            sort={"$vector": vector},
            limit=ann_limit,
            projection=find_projection,
            include_similarity=True,
        )
        return [
            BaseVectorStoreDriver.Entry(
                id=match["_id"],
                vector=match.get("$vector"),
                score=match["$similarity"],
                meta=match.get("meta"),
                namespace=match.get("keyspace"),
            )
            for match in matches
        ]
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `_collection = field(default=None, kw_only=True, alias='collection', metadata={'serializable': False})` class-attribute instance-attribute
- `api_endpoint = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `astra_db_namespace = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `caller_name = field(default='griptape', kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `collection_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `environment = field(kw_only=True, default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `token = field(kw_only=True, default=None, metadata={'serializable': False})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
@lazy_property()
def client(self) -> astrapy.DataAPIClient:
    astrapy = import_optional_dependency("astrapy")

    return astrapy.DataAPIClient(
        callers=[(self.caller_name, None)],
        environment=self.environment or astrapy.utils.unset.UnsetType(),
    )
```

collection()

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
@lazy_property()
def collection(self) -> astrapy.Collection:
    if self.token is None:
        raise ValueError("Astra DB token must be provided.")
    return self.client.get_database(
        self.api_endpoint, token=self.token, keyspace=self.astra_db_namespace
    ).get_collection(self.collection_name)
```

delete_vector(vector_id)

Delete a vector from Astra DB store.

The method succeeds regardless of whether a vector with the provided ID was actually stored or not in the first place.

Parameters

Name	Type	Description	Default
vector_id	*str*	ID of the vector to delete. 	*required*

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> None:
    """Delete a vector from Astra DB store.

    The method succeeds regardless of whether a vector with the provided ID
    was actually stored or not in the first place.

    Args:
        vector_id: ID of the vector to delete.
    """
    self.collection.delete_one({"_id": vector_id})
```

load_entries(*, namespace=None)

Load entries from the Astra DB store.

Parameters

Name	Type	Description	Default
namespace	*Optional[str]*	a namespace, within the vector store, to constrain the search. 	*None*

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	A list of vector (*BaseVectorStoreDriver.Entry*) entries.

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Load entries from the Astra DB store.

    Args:
        namespace: a namespace, within the vector store, to constrain the search.

    Returns:
        A list of vector (`BaseVectorStoreDriver.Entry`) entries.
    """
    find_filter: dict[str, str] = {} if namespace is None else {"keyspace": namespace}
    return [
        BaseVectorStoreDriver.Entry(
            id=match["_id"], vector=match.get("$vector"), meta=match.get("meta"),␣
↪namespace=match.get("keyspace")
        )
        for match in self.collection.find(filter=find_filter, projection={"*": 1})
    ]
```

load_entry(vector_id, *, namespace=None)

Load a single vector entry from the Astra DB store given its ID.

Parameters

Name	Type	Description	Default
vector_id	*str*	the ID of the required vector. 	*required*
namespace	*Optional[str]*	a namespace, within the vector store, to constrain the search. 	*None*

Returns

Type	Description
<a href="./#griptape.drivers.vector.BaseVectorStoreDriver.Entry" title="Entry (griptape.drivers.vector.BaseVectorStoreDriver.Entry)"><code class="language-python">C	The vector entry (a *BaseVectorStoreDriver.Entry*) if found, otherwise None.

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    """Load a single vector entry from the Astra DB store given its ID.

    Args:
        vector_id: the ID of the required vector.
        namespace: a namespace, within the vector store, to constrain the search.

    Returns:
        The vector entry (a `BaseVectorStoreDriver.Entry`) if found, otherwise None.
    """
    find_filter = {k: v for k, v in {"_id": vector_id, "keyspace": namespace}.items()␣
↪if v is not None}
    match = self.collection.find_one(filter=find_filter, projection={"*": 1})
    if match is not None:
        return BaseVectorStoreDriver.Entry(
            id=match["_id"], vector=match.get("$vector"), meta=match.get("meta"),␣
↪namespace=match.get("keyspace")
        )
    return None
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs)

Run a similarity search on the Astra DB store, based on a vector list.

Parameters

Name	Type	Description	Default
vector	<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	the vector to be queried. 	*required*
count	*Optional[int]*	the maximum number of results to return. If omitted, defaults will apply. 	*None*
namespace	*Optional[str]*	the namespace to filter results by. 	*None*
include_vectors	*bool*	whether to include vector data in the results. 	*False*
kwargs	*Any*	additional keyword arguments. Currently only the free-form dict *filter* is recognized (and goes straight to the Data API query); others will generate a warning and be ignored. 	*{}*

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	A list of vector (*BaseVectorStoreDriver.Entry*) entries,
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	with their *score* attribute set to the vector similarity to the query.

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
def query_vector(
    self,
```

(continues on next page)

(continued from previous page)

```
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs: Any,
) -> list[BaseVectorStoreDriver.Entry]:
    """Run a similarity search on the Astra DB store, based on a vector list.

    Args:
        vector: the vector to be queried.
        count: the maximum number of results to return. If omitted, defaults will␣
↪apply.
        namespace: the namespace to filter results by.
        include_vectors: whether to include vector data in the results.
        kwargs: additional keyword arguments. Currently only the free-form dict␣
↪`filter`
            is recognized (and goes straight to the Data API query);
            others will generate a warning and be ignored.

    Returns:
        A list of vector (`BaseVectorStoreDriver.Entry`) entries,
        with their `score` attribute set to the vector similarity to the query.
    """
    query_filter: Optional[dict[str, Any]] = kwargs.get("filter")
    find_filter_ns: dict[str, Any] = {} if namespace is None else {"keyspace":␣
↪namespace}
    find_filter = {**(query_filter or {}), **find_filter_ns}
    find_projection: Optional[dict[str, int]] = {"*": 1} if include_vectors else None
    ann_limit = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
    matches = self.collection.find(
        filter=find_filter,
        sort={"$vector": vector},
        limit=ann_limit,
        projection=find_projection,
        include_similarity=True,
    )
    return [
        BaseVectorStoreDriver.Entry(
            id=match["_id"],
            vector=match.get("$vector"),
            score=match["$similarity"],
            meta=match.get("meta"),
            namespace=match.get("keyspace"),
        )
        for match in matches
    ]
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Write a vector to the Astra DB store.

In case the provided ID exists already, an overwrite will take place.

Parameters

Name	Type	Description	Default
vector	<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	the vector to be upserted. 	*required*
vector_id	*Optional[str]*	the ID for the vector to store. If omitted, a server-provided new ID will be employed. 	*None*
namespace	*Optional[str]*	a namespace (a grouping within the vector store) to assign the vector to. 	*None*
meta	*Optional[dict]*	a metadata dictionary associated to the vector. 	*None*
kwargs	*Any*	additional keyword arguments. Currently none is used: if they are passed, they will be ignored with a warning. 	*{}*

Returns

Type	Description
str	the ID of the written vector (str).

Source Code in griptape/drivers/vector/astradb_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs: Any,
```

(continues on next page)

(continued from previous page)

```
) -> str:
    """Write a vector to the Astra DB store.

    In case the provided ID exists already, an overwrite will take place.

    Args:
        vector: the vector to be upserted.
        vector_id: the ID for the vector to store. If omitted, a server-provided new␣
↪ID will be employed.
        namespace: a namespace (a grouping within the vector store) to assign the␣
↪vector to.
        meta: a metadata dictionary associated to the vector.
        kwargs: additional keyword arguments. Currently none is used: if they are␣
↪passed, they will be ignored with a warning.

    Returns:
        the ID of the written vector (str).
    """
    document = {
        k: v
        for k, v in {"$vector": vector, "_id": vector_id, "keyspace": namespace, "meta
↪": meta}.items()
        if v is not None
    }
    if vector_id is not None:
        self.collection.find_one_and_replace({"_id": vector_id}, document,␣
↪upsert=True)
        return vector_id
    insert_result = self.collection.insert_one(document)
    return insert_result.inserted_id
```

azure_mongodb

- `__all__ = ['AzureMongoDbVectorStoreDriver']` module-attribute

Bases: *MongoDbAtlasVectorStoreDriver* “)

Source Code in griptape/drivers/vector/azure_mongodb_vector_store_driver.py

```
@define
class AzureMongoDbVectorStoreDriver(MongoDbAtlasVectorStoreDriver):
    """A Vector Store Driver for CosmosDB with MongoDB vCore API."""

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        offset: Optional[int] = None,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
```

(continues on next page)

(continued from previous page)

```
        """Queries the MongoDB collection for documents that match the provided␣
↪vector list.

        Results can be customized based on parameters like count, namespace,␣
↪inclusion of vectors, offset, and index.
        """
        collection = self.get_collection()

        count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
        offset = offset or 0

        pipeline = []

        pipeline.append(
            {
                "$search": {
                    "cosmosSearch": {
                        "vector": vector,
                        "path": self.vector_path,
                        "k": min(count * self.num_candidates_multiplier, self.MAX_NUM_
↪CANDIDATES),
                    },
                    "returnStoredSource": True,
                },
            },
        )

        if namespace:
            pipeline.append({"$match": {"namespace": namespace}})

        pipeline.append({"$project": {"similarityScore": {"$meta": "searchScore"},
↪"document": "$$ROOT"}})

        return [
            BaseVectorStoreDriver.Entry(
                id=str(doc["_id"]),
                vector=doc[self.vector_path] if include_vectors else [],
                score=doc["similarityScore"],
                meta=doc["document"]["meta"],
                namespace=namespace,
            )
            for doc in collection.aggregate(pipeline)
        ]
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, offset=None, **kwargs)

Source Code in griptape/drivers/vector/azure_mongodb_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
```

(continues on next page)

(continued from previous page)

```
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    offset: Optional[int] = None,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    """Queries the MongoDB collection for documents that match the provided vector␣
↪list.

    Results can be customized based on parameters like count, namespace, inclusion of␣
↪vectors, offset, and index.
    """
    collection = self.get_collection()

    count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
    offset = offset or 0

    pipeline = []

    pipeline.append(
        {
            "$search": {
                "cosmosSearch": {
                    "vector": vector,
                    "path": self.vector_path,
                    "k": min(count * self.num_candidates_multiplier, self.MAX_NUM_
↪CANDIDATES),
                },
                "returnStoredSource": True,
            },
        },
    )

    if namespace:
        pipeline.append({"$match": {"namespace": namespace}})

    pipeline.append({"$project": {"similarityScore": {"$meta": "searchScore"},
↪"document": "$$ROOT"}})

    return [
        BaseVectorStoreDriver.Entry(
            id=str(doc["_id"]),
            vector=doc[self.vector_path] if include_vectors else [],
            score=doc["similarityScore"],
            meta=doc["document"]["meta"],
            namespace=namespace,
        )
        for doc in collection.aggregate(pipeline)
    ]
```

azure_mongodb_vector_store_driver

Bases: *MongoDbAtlasVectorStoreDriver* “)

Source Code in griptape/drivers/vector/azure_mongodb_vector_store_driver.py

```
@define
class AzureMongoDbVectorStoreDriver(MongoDbAtlasVectorStoreDriver):
    """A Vector Store Driver for CosmosDB with MongoDB vCore API."""

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        offset: Optional[int] = None,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Queries the MongoDB collection for documents that match the provided␣
↪vector list.

        Results can be customized based on parameters like count, namespace,␣
↪inclusion of vectors, offset, and index.
        """
        collection = self.get_collection()

        count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
        offset = offset or 0

        pipeline = []

        pipeline.append(
            {
                "$search": {
                    "cosmosSearch": {
                        "vector": vector,
                        "path": self.vector_path,
                        "k": min(count * self.num_candidates_multiplier, self.MAX_NUM_
↪CANDIDATES),
                    },
                    "returnStoredSource": True,
                },
            },
        )

        if namespace:
            pipeline.append({"$match": {"namespace": namespace}})

        pipeline.append({"$project": {"similarityScore": {"$meta": "searchScore"},
↪"document": "$$ROOT"}})

        return [
```

(continues on next page)

(continued from previous page)

```
            BaseVectorStoreDriver.Entry(
                id=str(doc["_id"]),
                vector=doc[self.vector_path] if include_vectors else [],
                score=doc["similarityScore"],
                meta=doc["document"]["meta"],
                namespace=namespace,
            )
            for doc in collection.aggregate(pipeline)
        ]
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, offset=None, **kwargs)

Source Code in griptape/drivers/vector/azure_mongodb_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    offset: Optional[int] = None,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    """Queries the MongoDB collection for documents that match the provided vector␣
↪list.

    Results can be customized based on parameters like count, namespace, inclusion of␣
↪vectors, offset, and index.
    """
    collection = self.get_collection()

    count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
    offset = offset or 0

    pipeline = []

    pipeline.append(
        {
            "$search": {
                "cosmosSearch": {
                    "vector": vector,
                    "path": self.vector_path,
                    "k": min(count * self.num_candidates_multiplier, self.MAX_NUM_
↪CANDIDATES),
                },
                "returnStoredSource": True,
            },
        },
    )
```

(continues on next page)

(continued from previous page)

```
    if namespace:
        pipeline.append({"$match": {"namespace": namespace}})

    pipeline.append({"$project": {"similarityScore": {"$meta": "searchScore"},
↪"document": "$$ROOT"}})

    return [
        BaseVectorStoreDriver.Entry(
            id=str(doc["_id"]),
            vector=doc[self.vector_path] if include_vectors else [],
            score=doc["similarityScore"],
            meta=doc["document"]["meta"],
            namespace=namespace,
        )
        for doc in collection.aggregate(pipeline)
    ]
```

base_vector_store_driver

Bases: *SerializableMixin* “) , *FuturesExecutorMixin* ”) , ABC

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
@define
class BaseVectorStoreDriver(SerializableMixin, FuturesExecutorMixin, ABC):
    DEFAULT_QUERY_COUNT = 5

    @define
    class Entry(SerializableMixin):
        id: str = field(metadata={"serializable": True})
        vector: Optional[list[float]] = field(default=None, metadata={"serializable":␣
↪True})
        score: Optional[float] = field(default=None, metadata={"serializable": True})
        meta: Optional[dict] = field(default=None, metadata={"serializable": True})
        namespace: Optional[str] = field(default=None, metadata={"serializable": True}
↪)

        def to_artifact(self) -> BaseArtifact:
            return BaseArtifact.from_json(self.meta["artifact"])  # pyright:␣
↪ignore[reportOptionalSubscript]

    embedding_driver: BaseEmbeddingDriver = field(kw_only=True, metadata={
↪"serializable": True})

    def upsert_text_artifacts(
        self,
        artifacts: list[TextArtifact] | dict[str, list[TextArtifact]],
        *,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> list[str] | dict[str, list[str]]:
        warnings.warn(
```

(continues on next page)

(continued from previous page)

```
            "`BaseVectorStoreDriver.upsert_text_artifacts` is deprecated and will be␣
↪removed in a future release. `BaseVectorStoreDriver.upsert_collection` is a drop-in␣
↪replacement.",
            DeprecationWarning,
            stacklevel=2,
        )
        return self.upsert_collection(artifacts, meta=meta, **kwargs)

    def upsert_text_artifact(
        self,
        artifact: TextArtifact,
        *,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        vector_id: Optional[str] = None,
        **kwargs,
    ) -> str:
        warnings.warn(
            "`BaseVectorStoreDriver.upsert_text_artifacts` is deprecated and will be␣
↪removed in a future release. `BaseVectorStoreDriver.upsert` is a drop-in␣
↪replacement.",
            DeprecationWarning,
            stacklevel=2,
        )
        return self.upsert(artifact, namespace=namespace, meta=meta, vector_id=vector_
↪id, **kwargs)

    def upsert_text(
        self,
        string: str,
        *,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        vector_id: Optional[str] = None,
        **kwargs,
    ) -> str:
        warnings.warn(
            "`BaseVectorStoreDriver.upsert_text` is deprecated and will be removed in␣
↪a future release. `BaseVectorStoreDriver.upsert` is a drop-in replacement.",
            DeprecationWarning,
            stacklevel=2,
        )
        return self.upsert(string, namespace=namespace, meta=meta, vector_id=vector_
↪id, **kwargs)

    @overload
    def upsert_collection(
        self,
        artifacts: list[TextArtifact] | list[ImageArtifact],
        *,
        meta: Optional[dict] = None,
        **kwargs,
```

(continues on next page)

(continued from previous page)

```
    ) -> list[str]: ...

    @overload
    def upsert_collection(
        self,
        artifacts: dict[str, list[TextArtifact]] | dict[str, list[ImageArtifact]],
        *,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> dict[str, list[str]]: ...

    def upsert_collection(
        self,
        artifacts: list[TextArtifact]
        | list[ImageArtifact]
        | dict[str, list[TextArtifact]]
        | dict[str, list[ImageArtifact]],
        *,
        meta: Optional[dict] = None,
        **kwargs,
    ):
        with self.create_futures_executor() as futures_executor:
            if isinstance(artifacts, list):
                return utils.execute_futures_list(
                    [
                        futures_executor.submit(with_contextvars(self.upsert), a,␣
↪namespace=None, meta=meta, **kwargs)
                        for a in artifacts
                    ],
                )
            futures_dict = {}

            for namespace, artifact_list in artifacts.items():
                for a in artifact_list:
                    if not futures_dict.get(namespace):
                        futures_dict[namespace] = []

                    futures_dict[namespace].append(
                        futures_executor.submit(
                            with_contextvars(self.upsert), a, namespace=namespace,␣
↪meta=meta, **kwargs
                        )
                    )

            return utils.execute_futures_list_dict(futures_dict)

    def upsert(
        self,
        value: str | TextArtifact | ImageArtifact,
        *,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
```

(continues on next page)

(continued from previous page)

```
        vector_id: Optional[str] = None,
        **kwargs,
    ) -> str:
        artifact = TextArtifact(value) if isinstance(value, str) else value

        meta = {} if meta is None else meta

        if vector_id is None:
            value = artifact.to_text() if artifact.reference is None else artifact.to_
↪text() + str(artifact.reference)
            vector_id = self._get_default_vector_id(value)

        if self.does_entry_exist(vector_id, namespace=namespace):
            return vector_id
        meta = {**meta, "artifact": artifact.to_json()}

        vector = self.embedding_driver.embed(artifact, vector_operation="upsert")

        return self.upsert_vector(vector, vector_id=vector_id, namespace=namespace,␣
↪meta=meta, **kwargs)

    def does_entry_exist(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪bool:
        try:
            return self.load_entry(vector_id, namespace=namespace) is not None
        except Exception:
            return False

    def load_artifacts(self, *, namespace: Optional[str] = None) -> ListArtifact:
        result = self.load_entries(namespace=namespace)
        artifacts = [r.to_artifact() for r in result]

        return ListArtifact([a for a in artifacts if isinstance(a, TextArtifact)])

    @abstractmethod
    def delete_vector(self, vector_id: str) -> None: ...

    @abstractmethod
    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str: ...

    @abstractmethod
    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[Entry]: ...
```

(continues on next page)

(continued from previous page)

```
    @abstractmethod
    def load_entries(self, *, namespace: Optional[str] = None) -> list[Entry]: ...

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[Entry]:
        # TODO: Mark as abstract method for griptape 2.0
        raise NotImplementedError(f"{self.__class__.__name__} does not support vector query.")

    def query(
        self,
        query: str | TextArtifact | ImageArtifact,
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[Entry]:
        try:
            vector = self.embedding_driver.embed(query, vector_operation="query")
        except ValueError as e:
            raise ValueError(
                "The Embedding Driver, %s, used by the Vector Store does not support embedding the %s type."
                "To resolve, provide an Embedding Driver that supports this type.",
                self.embedding_driver.__class__.__name__,
                type(query),
            ) from e
        return self.query_vector(vector, count=count, namespace=namespace, include_vectors=include_vectors, **kwargs)

    def _get_default_vector_id(self, value: str) -> str:
        return str(uuid.uuid5(uuid.NAMESPACE_OID, value))
```

- `DEFAULT_QUERY_COUNT = 5` class-attribute instance-attribute
- `embedding_driver = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

Entry

Bases: *SerializableMixin* “)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
@define
```

(continues on next page)

(continued from previous page)

```
class Entry(SerializableMixin):
    id: str = field(metadata={"serializable": True})
    vector: Optional[list[float]] = field(default=None, metadata={"serializable":␣
↪True})
    score: Optional[float] = field(default=None, metadata={"serializable": True})
    meta: Optional[dict] = field(default=None, metadata={"serializable": True})
    namespace: Optional[str] = field(default=None, metadata={"serializable": True})

    def to_artifact(self) -> BaseArtifact:
        return BaseArtifact.from_json(self.meta["artifact"])  # pyright:␣
↪ignore[reportOptionalSubscript]
```

- `id = field(metadata={'serializable': True})` class-attribute instance-attribute
- `meta = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `namespace = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `score = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `vector = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute

_get_default_vector_id(value)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def _get_default_vector_id(self, value: str) -> str:
    return str(uuid.uuid5(uuid.NAMESPACE_OID, value))
```

delete_vector(vector_id)abstractmethod

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
@abstractmethod
def delete_vector(self, vector_id: str) -> None: ...
```

does_entry_exist(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def does_entry_exist(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪bool:
    try:
        return self.load_entry(vector_id, namespace=namespace) is not None
    except Exception:
        return False
```

load_artifacts(*, namespace=None)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def load_artifacts(self, *, namespace: Optional[str] = None) -> ListArtifact:
    result = self.load_entries(namespace=namespace)
    artifacts = [r.to_artifact() for r in result]

    return ListArtifact([a for a in artifacts if isinstance(a, TextArtifact)])
```

load_entries(*, namespace=None)abstractmethod

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
@abstractmethod
def load_entries(self, *, namespace: Optional[str] = None) -> list[Entry]: ...
```

load_entry(vector_id, *, namespace=None)abstractmethod

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
@abstractmethod
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[Entry]: ...
```

query(query, *, count=None, namespace=None, include_vectors=False, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def query(
    self,
    query: str | TextArtifact | ImageArtifact,
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[Entry]:
    try:
        vector = self.embedding_driver.embed(query, vector_operation="query")
    except ValueError as e:
        raise ValueError(
            "The Embedding Driver, %s, used by the Vector Store does not support␣
↪embedding the %s type."
            "To resolve, provide an Embedding Driver that supports this type.",
            self.embedding_driver.__class__.__name__,
            type(query),
        ) from e
    return self.query_vector(vector, count=count, namespace=namespace, include_
↪vectors=include_vectors, **kwargs)
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[Entry]:
    # TODO: Mark as abstract method for griptape 2.0
    raise NotImplementedError(f"{self.__class__.__name__} does not support vector␣
↪query.")
```

upsert(value, *, namespace=None, meta=None, vector_id=None, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def upsert(
    self,
    value: str | TextArtifact | ImageArtifact,
    *,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    vector_id: Optional[str] = None,
    **kwargs,
) -> str:
    artifact = TextArtifact(value) if isinstance(value, str) else value

    meta = {} if meta is None else meta

    if vector_id is None:
        value = artifact.to_text() if artifact.reference is None else artifact.to_
↪text() + str(artifact.reference)
        vector_id = self._get_default_vector_id(value)

    if self.does_entry_exist(vector_id, namespace=namespace):
        return vector_id
    meta = {**meta, "artifact": artifact.to_json()}

    vector = self.embedding_driver.embed(artifact, vector_operation="upsert")

    return self.upsert_vector(vector, vector_id=vector_id, namespace=namespace,␣
↪meta=meta, **kwargs)
```

upsert_collection(artifacts, *, meta=None, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def upsert_collection(
    self,
    artifacts: list[TextArtifact]
    | list[ImageArtifact]
    | dict[str, list[TextArtifact]]
    | dict[str, list[ImageArtifact]],
    *,
    meta: Optional[dict] = None,
    **kwargs,
):
    with self.create_futures_executor() as futures_executor:
        if isinstance(artifacts, list):
            return utils.execute_futures_list(
                [
                    futures_executor.submit(with_contextvars(self.upsert), a,␣
↪namespace=None, meta=meta, **kwargs)
                    for a in artifacts
                ],
            )
        futures_dict = {}

        for namespace, artifact_list in artifacts.items():
            for a in artifact_list:
                if not futures_dict.get(namespace):
                    futures_dict[namespace] = []

                futures_dict[namespace].append(
                    futures_executor.submit(
                        with_contextvars(self.upsert), a, namespace=namespace,␣
↪meta=meta, **kwargs
                    )
                )

        return utils.execute_futures_list_dict(futures_dict)
```

upsert_text(string, *, namespace=None, meta=None, vector_id=None, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def upsert_text(
    self,
    string: str,
    *,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    vector_id: Optional[str] = None,
    **kwargs,
) -> str:
    warnings.warn(
        "`BaseVectorStoreDriver.upsert_text` is deprecated and will be removed in a␣
↪future release. `BaseVectorStoreDriver.upsert` is a drop-in replacement.",
        DeprecationWarning,
```

(continues on next page)

(continued from previous page)

```
        stacklevel=2,
    )
    return self.upsert(string, namespace=namespace, meta=meta, vector_id=vector_id,␣
↪**kwargs)
```

upsert_text_artifact(artifact, *, namespace=None, meta=None, vector_id=None, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def upsert_text_artifact(
    self,
    artifact: TextArtifact,
    *,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    vector_id: Optional[str] = None,
    **kwargs,
) -> str:
    warnings.warn(
        "`BaseVectorStoreDriver.upsert_text_artifacts` is deprecated and will be␣
↪removed in a future release. `BaseVectorStoreDriver.upsert` is a drop-in␣
↪replacement.",
        DeprecationWarning,
        stacklevel=2,
    )
    return self.upsert(artifact, namespace=namespace, meta=meta, vector_id=vector_id,␣
↪**kwargs)
```

upsert_text_artifacts(artifacts, *, meta=None, **kwargs)

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
def upsert_text_artifacts(
    self,
    artifacts: list[TextArtifact] | dict[str, list[TextArtifact]],
    *,
    meta: Optional[dict] = None,
    **kwargs,
) -> list[str] | dict[str, list[str]]:
    warnings.warn(
        "`BaseVectorStoreDriver.upsert_text_artifacts` is deprecated and will be␣
↪removed in a future release. `BaseVectorStoreDriver.upsert_collection` is a drop-in␣
↪replacement.",
        DeprecationWarning,
        stacklevel=2,
    )
    return self.upsert_collection(artifacts, meta=meta, **kwargs)
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)abstractmethod

Source Code in griptape/drivers/vector/base_vector_store_driver.py

```
@abstractmethod
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str: ...
```

dummy

- `__all__ = ['DummyVectorStoreDriver']` module-attribute

Bases: *BaseVectorStoreDriver* “)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
@define()
class DummyVectorStoreDriver(BaseVectorStoreDriver):
    embedding_driver: BaseEmbeddingDriver = field(
        kw_only=True,
        default=Factory(lambda: DummyEmbeddingDriver()),
        metadata={"serializable": True},
    )

    def delete_vector(self, vector_id: str) -> None:
        raise DummyError(__class__.__name__, "delete_vector")

    def upsert_vector(
        self,
        vector: list[float],
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        raise DummyError(__class__.__name__, "upsert_vector")

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        raise DummyError(__class__.__name__, "load_entry")

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        raise DummyError(__class__.__name__, "load_entries")

    def query_vector(
        self,
```

(continues on next page)

(continued from previous page)

```
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        raise DummyError(__class__.__name__, "query_vector")

    def query(
        self,
        query: str | TextArtifact | ImageArtifact,
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        raise DummyError(__class__.__name__, "query")
```

- `embedding_driver = field(kw_only=True, default=Factory(lambda: DummyEmbeddingDriver()), metadata={'serializable': True})` class-attribute instance-attribute

delete_vector(vector_id)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> None:
    raise DummyError(__class__.__name__, "delete_vector")
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    raise DummyError(__class__.__name__, "load_entries")
```

load_entry(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    raise DummyError(__class__.__name__, "load_entry")
```

query(query, *, count=None, namespace=None, include_vectors=False, **kwargs)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
def query(
    self,
    query: str | TextArtifact | ImageArtifact,
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    raise DummyError(__class__.__name__, "query")
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    raise DummyError(__class__.__name__, "query_vector")
```

upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    raise DummyError(__class__.__name__, "upsert_vector")
```

dummy_vector_store_driver

Bases: *BaseVectorStoreDriver* “)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
@define()
class DummyVectorStoreDriver(BaseVectorStoreDriver):
    embedding_driver: BaseEmbeddingDriver = field(
        kw_only=True,
        default=Factory(lambda: DummyEmbeddingDriver()),
        metadata={"serializable": True},
```

(continues on next page)

(continued from previous page)

```
    )

    def delete_vector(self, vector_id: str) -> None:
        raise DummyError(__class__.__name__, "delete_vector")

    def upsert_vector(
        self,
        vector: list[float],
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        raise DummyError(__class__.__name__, "upsert_vector")

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        raise DummyError(__class__.__name__, "load_entry")

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        raise DummyError(__class__.__name__, "load_entries")

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        raise DummyError(__class__.__name__, "query_vector")

    def query(
        self,
        query: str | TextArtifact | ImageArtifact,
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        raise DummyError(__class__.__name__, "query")
```

- `embedding_driver = field(kw_only=True, default=Factory(lambda: DummyEmbeddingDriver()), metadata={'serializable': True})` class-attribute instance-attribute

delete_vector(vector_id)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> None:
    raise DummyError(__class__.__name__, "delete_vector")
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    raise DummyError(__class__.__name__, "load_entries")
```

load_entry(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    raise DummyError(__class__.__name__, "load_entry")
```

query(query, *, count=None, namespace=None, include_vectors=False, **kwargs)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
def query(
    self,
    query: str | TextArtifact | ImageArtifact,
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    raise DummyError(__class__.__name__, "query")
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    raise DummyError(__class__.__name__, "query_vector")
```

upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/dummy_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    raise DummyError(__class__.__name__, "upsert_vector")
```

griptape_cloud

- `__all__ = ['GriptapeCloudVectorStoreDriver']` module-attribute

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="grip-tape_cloud.mdx#grip-tape.drivers.vector.grip-tape_cloud.Grip-tapeCloudVector-StoreDriver.api_key" title="api_key = field(default=Fac-tory(lambda: os.envi-ron['GT_CLOUD_API_K class-attribute instance-attribute (grip-tape.drivers.vector.grip-tape_cloud.Grip-tapeCloudVector-StoreDriver.api_key)"><c class="language-python">	*str*	API Key for Gen AI Builder.

continues on next page

Table 240 – continued from previous page

Name	Type	Description
<a href="griptape_cloud.mdx#griptape.drivers.vector.griptape_cloud.GriptapeCloudVectorStoreDriver.knowledge_base_id" title="knowledge_base_id = field(kw_only=True, metadata={'serializable': True})">class-attribute instance-attribute (griptape.drivers.vector.griptape_cloud.GriptapeCloudVectorStoreDriver.knowledge_base_id)"><code class="language-python">	*str*	Knowledge Base ID for Gen AI Builder.
<a href="griptape_cloud.mdx#griptape.drivers.vector.griptape_cloud.GriptapeCloudVectorStoreDriver.base_url" title="base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_ 'https://cloud.griptape.ai')))">class-attribute instance-attribute (griptape.drivers.vector.griptape_cloud.GriptapeCloudVectorStoreDriver.base_url)"><class="language-python">	*str*	Base URL for Gen AI Builder.

continues on next page

Table 240 – continued from previous page

Name	Type	Description
<a href="griptape_cloud.mdx#griptape.drivers.vector.griptape_cloud.GriptapeCloudVectorStoreDriver.headers" title="headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.drivers.vector.griptape_cloud.GriptapeCloudVectorStoreDriver.headers)"><code class="language-python">	*dict*	Headers for Gen AI Builder.

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
@define
class GriptapeCloudVectorStoreDriver(BaseVectorStoreDriver):
    """A vector store driver for Gen AI Builder Knowledge Bases.

    Attributes:
        api_key: API Key for Gen AI Builder.
        knowledge_base_id: Knowledge Base ID for Gen AI Builder.
        base_url: Base URL for Gen AI Builder.
        headers: Headers for Gen AI Builder.
    """

    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]))
    knowledge_base_id: str = field(kw_only=True, metadata={"serializable": True})
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        kw_only=True,
    )
    embedding_driver: BaseEmbeddingDriver = field(
        default=Factory(lambda: DummyEmbeddingDriver()),
        metadata={"serializable": True},
        kw_only=True,
```

(continues on next page)

(continued from previous page)

```
        init=False,
    )

    def upsert_vector(
        self,
        vector: list[float],
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        raise NotImplementedError(f"{self.__class__.__name__} does not support vector␣
↪upsert.")

    def upsert_text_artifact(
        self,
        artifact: TextArtifact,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        vector_id: Optional[str] = None,
        **kwargs,
    ) -> str:
        raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪artifact upsert.")

    def upsert_text(
        self,
        string: str,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪upsert.")

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪BaseVectorStoreDriver.Entry:
        raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")

    def load_artifacts(self, *, namespace: Optional[str] = None) -> ListArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪Artifact loading.")

    def query(
        self,
```

(continues on next page)

(continued from previous page)

```
        query: str | TextArtifact | ImageArtifact,
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: Optional[bool] = None,
        distance_metric: Optional[str] = None,
        # GriptapeCloudVectorStoreDriver-specific params:
        filter: Optional[dict] = None,  # noqa: A002
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Performs a query on the Knowledge Base.

        Performs a query on the Knowledge Base and returns Artifacts with close␣
↪vector proximity to the query, optionally filtering to only those that match the␣
↪provided filter(s).
        """
        if isinstance(query, ImageArtifact):
            raise ValueError(f"{self.__class__.__name__} does not support querying␣
↪with Image Artifacts.")
        url = griptape_cloud_url(self.base_url, f"api/knowledge-bases/{self.knowledge_
↪base_id}/query")

        query_args = {
            "count": count,
            "distance_metric": distance_metric,
            "filter": filter,
            "include_vectors": include_vectors,
        }
        query_args = {k: v for k, v in query_args.items() if v is not None}

        request: dict[str, Any] = {
            "query": str(query),
            "query_args": query_args,
        }

        response = requests.post(url, json=request, headers=self.headers).json()
        entries = response.get("entries", [])
        return [BaseVectorStoreDriver.Entry.from_dict(entry) for entry in entries]

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")
```

- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']))` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')))` class-attribute instance-attribute
- `embedding_driver = field(default=Factory(lambda: DummyEmbeddingDriver()), metadata={'serializable': True}, kw_only=True, init=False)` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute

- `knowledge_base_id = field(kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute

delete_vector(vector_id)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

load_artifacts(*, namespace=None)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def load_artifacts(self, *, namespace: Optional[str] = None) -> ListArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support Artifact␣
↪loading.")
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")
```

load_entry(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪BaseVectorStoreDriver.Entry:
    raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")
```

query(query, *, count=None, namespace=None, include_vectors=None, distance_metric=None, filter=None, **kwargs)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def query(
    self,
    query: str | TextArtifact | ImageArtifact,
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: Optional[bool] = None,
    distance_metric: Optional[str] = None,
    # GriptapeCloudVectorStoreDriver-specific params:
    filter: Optional[dict] = None,  # noqa: A002
    **kwargs,
```

(continues on next page)

(continued from previous page)

```
) -> list[BaseVectorStoreDriver.Entry]:
    """Performs a query on the Knowledge Base.

    Performs a query on the Knowledge Base and returns Artifacts with close vector␣
↪proximity to the query, optionally filtering to only those that match the provided␣
↪filter(s).
    """
    if isinstance(query, ImageArtifact):
        raise ValueError(f"{self.__class__.__name__} does not support querying with␣
↪Image Artifacts.")
    url = griptape_cloud_url(self.base_url, f"api/knowledge-bases/{self.knowledge_
↪base_id}/query")

    query_args = {
        "count": count,
        "distance_metric": distance_metric,
        "filter": filter,
        "include_vectors": include_vectors,
    }
    query_args = {k: v for k, v in query_args.items() if v is not None}

    request: dict[str, Any] = {
        "query": str(query),
        "query_args": query_args,
    }

    response = requests.post(url, json=request, headers=self.headers).json()
    entries = response.get("entries", [])
    return [BaseVectorStoreDriver.Entry.from_dict(entry) for entry in entries]
```

upsert_text(string, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def upsert_text(
    self,
    string: str,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪upsert.")
```

upsert_text_artifact(artifact, namespace=None, meta=None, vector_id=None, **kwargs)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def upsert_text_artifact(
    self,
```

(continues on next page)

(continued from previous page)

```
    artifact: TextArtifact,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    vector_id: Optional[str] = None,
    **kwargs,
) -> str:
    raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪artifact upsert.")
```

upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    raise NotImplementedError(f"{self.__class__.__name__} does not support vector␣
↪upsert.")
```

griptape_cloud_vector_store_driver

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="griptape_cloud_vector_store_driver.mdx#griptape.drivers.vector.griptape_cloud_vector_store_driver.GriptapeCloudVectorStoreDriver.api_key" title="api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_K class-attribute instance-attribute (griptape.drivers.vector.griptape_cloud_vector_store_driver.GriptapeCloudVectorStoreDriver.api_key)"><c class="language-python">	*str*	API Key for Gen AI Builder.
<a href="griptape_cloud_vector_store_driver.mdx#griptape.drivers.vector.griptape_cloud_vector_store_driver.GriptapeCloudVectorStoreDriver.knowledge_base_id" title="knowledge_base_id = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.griptape_cloud_vector_store_driver.GriptapeCloudVectorStoreDriver.knowledge_base_id)"><code class="language-python">	*str*	Knowledge Base ID for Gen AI Builder.

continues on next page

Table 241 – continued from previous page

Name	Type	Description
<a href="griptape_cloud_vector_store_driver.mdx#griptape.drivers.vector.griptape_cloud_vector_store_driver.GriptapeCloudVectorStoreDriver.base_url" title="base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_ 'https://cloud.griptape.ai'))) class-attribute instance-attribute (griptape.drivers.vector.griptape_cloud_vector_store_driver.GriptapeCloudVectorStoreDriver.base_url)"><class="language-python">	*str*	Base URL for Gen AI Builder.
<a href="griptape_cloud_vector_store_driver.mdx#griptape.drivers.vector.griptape_cloud_vector_store_driver.GriptapeCloudVectorStoreDriver.headers" title="headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.drivers.vector.griptape_cloud_vector_store_driver.GriptapeCloudVectorStoreDriver.headers)"><code class="language-python">	*dict*	Headers for Gen AI Builder.

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
@define
class GriptapeCloudVectorStoreDriver(BaseVectorStoreDriver):
```

(continues on next page)

(continued from previous page)

```
    """A vector store driver for Gen AI Builder Knowledge Bases.

    Attributes:
        api_key: API Key for Gen AI Builder.
        knowledge_base_id: Knowledge Base ID for Gen AI Builder.
        base_url: Base URL for Gen AI Builder.
        headers: Headers for Gen AI Builder.
    """

    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
    )
    api_key: str = field(default=Factory(lambda: os.environ["GT_CLOUD_API_KEY"]))
    knowledge_base_id: str = field(kw_only=True, metadata={"serializable": True})
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        kw_only=True,
    )
    embedding_driver: BaseEmbeddingDriver = field(
        default=Factory(lambda: DummyEmbeddingDriver()),
        metadata={"serializable": True},
        kw_only=True,
        init=False,
    )

    def upsert_vector(
        self,
        vector: list[float],
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        raise NotImplementedError(f"{self.__class__.__name__} does not support vector␣
↪upsert.")

    def upsert_text_artifact(
        self,
        artifact: TextArtifact,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        vector_id: Optional[str] = None,
        **kwargs,
    ) -> str:
        raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪artifact upsert.")

    def upsert_text(
        self,
        string: str,
```

(continues on next page)

(continued from previous page)

```
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪upsert.")

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪BaseVectorStoreDriver.Entry:
        raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")

    def load_artifacts(self, *, namespace: Optional[str] = None) -> ListArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪Artifact loading.")

    def query(
        self,
        query: str | TextArtifact | ImageArtifact,
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: Optional[bool] = None,
        distance_metric: Optional[str] = None,
        # GriptapeCloudVectorStoreDriver-specific params:
        filter: Optional[dict] = None,  # noqa: A002
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Performs a query on the Knowledge Base.

        Performs a query on the Knowledge Base and returns Artifacts with close␣
↪vector proximity to the query, optionally filtering to only those that match the␣
↪provided filter(s).
        """
        if isinstance(query, ImageArtifact):
            raise ValueError(f"{self.__class__.__name__} does not support querying␣
↪with Image Artifacts.")
        url = griptape_cloud_url(self.base_url, f"api/knowledge-bases/{self.knowledge_
↪base_id}/query")

        query_args = {
            "count": count,
            "distance_metric": distance_metric,
            "filter": filter,
            "include_vectors": include_vectors,
        }
```

(continues on next page)

(continued from previous page)

```
        query_args = {k: v for k, v in query_args.items() if v is not None}

        request: dict[str, Any] = {
            "query": str(query),
            "query_args": query_args,
        }

        response = requests.post(url, json=request, headers=self.headers).json()
        entries = response.get("entries", [])
        return [BaseVectorStoreDriver.Entry.from_dict(entry) for entry in entries]

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")
```

- `api_key = field(default=Factory(lambda: os.environ['GT_CLOUD_API_KEY']))` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')))` class-attribute instance-attribute
- `embedding_driver = field(default=Factory(lambda: DummyEmbeddingDriver()), metadata={'serializable': True}, kw_only=True, init=False)` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `knowledge_base_id = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

delete_vector(vector_id)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

load_artifacts(*, namespace=None)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def load_artifacts(self, *, namespace: Optional[str] = None) -> ListArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support Artifact␣
↪loading.")
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")
```

load_entry(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪BaseVectorStoreDriver.Entry:
    raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")
```

query(query, *, count=None, namespace=None, include_vectors=None, distance_metric=None, filter=None, **kwargs)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def query(
    self,
    query: str | TextArtifact | ImageArtifact,
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: Optional[bool] = None,
    distance_metric: Optional[str] = None,
    # GriptapeCloudVectorStoreDriver-specific params:
    filter: Optional[dict] = None,  # noqa: A002
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    """Performs a query on the Knowledge Base.

    Performs a query on the Knowledge Base and returns Artifacts with close vector␣
↪proximity to the query, optionally filtering to only those that match the provided␣
↪filter(s).
    """
    if isinstance(query, ImageArtifact):
        raise ValueError(f"{self.__class__.__name__} does not support querying with␣
↪Image Artifacts.")
    url = griptape_cloud_url(self.base_url, f"api/knowledge-bases/{self.knowledge_
↪base_id}/query")

    query_args = {
        "count": count,
        "distance_metric": distance_metric,
        "filter": filter,
        "include_vectors": include_vectors,
    }
    query_args = {k: v for k, v in query_args.items() if v is not None}

    request: dict[str, Any] = {
        "query": str(query),
        "query_args": query_args,
    }

    response = requests.post(url, json=request, headers=self.headers).json()
    entries = response.get("entries", [])
    return [BaseVectorStoreDriver.Entry.from_dict(entry) for entry in entries]
```

upsert_text(string, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def upsert_text(
    self,
    string: str,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪upsert.")
```

upsert_text_artifact(artifact, namespace=None, meta=None, vector_id=None, **kwargs)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def upsert_text_artifact(
    self,
    artifact: TextArtifact,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    vector_id: Optional[str] = None,
    **kwargs,
) -> str:
    raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪artifact upsert.")
```

upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/griptape_cloud_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    raise NotImplementedError(f"{self.__class__.__name__} does not support vector␣
↪upsert.")
```

local

- `__all__ = ['LocalVectorStoreDriver']` module-attribute

Bases: *BaseVectorStoreDriver* “)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
@define(kw_only=True)
class LocalVectorStoreDriver(BaseVectorStoreDriver):
    entries: dict[str, BaseVectorStoreDriver.Entry] = field(factory=dict)
    persist_file: Optional[str] = field(default=None)
    calculate_relatedness: Callable = field(default=lambda x, y: dot(x, y) / (norm(x)␣
↪* norm(y)))
    thread_lock: threading.Lock = field(default=Factory(lambda: threading.Lock()))

    def __attrs_post_init__(self) -> None:
        if self.persist_file is not None:
            directory = os.path.dirname(self.persist_file)

            if directory and not os.path.exists(directory):
                os.makedirs(directory)

            if not os.path.isfile(self.persist_file):
                with open(self.persist_file, "w") as file:
                    self.__save_entries_to_file(file)

            with open(self.persist_file, "r+") as file:
                if os.path.getsize(self.persist_file) > 0:
                    self.entries = self.load_entries_from_file(file)
                else:
                    self.__save_entries_to_file(file)

    def load_entries_from_file(self, json_file: TextIO) -> dict[str,␣
↪BaseVectorStoreDriver.Entry]:
        with self.thread_lock:
            data = json.load(json_file)

            return {k: BaseVectorStoreDriver.Entry.from_dict(v) for k, v in data.
↪items()}

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        vector_id = vector_id or utils.str_to_hash(str(vector))

        with self.thread_lock:
            self.entries[self.__namespaced_vector_id(vector_id, namespace=namespace)]␣
↪= self.Entry(
                id=vector_id,
                vector=vector,
                meta=meta,
                namespace=namespace,
            )
```

(continues on next page)

(continued from previous page)

```
        if self.persist_file is not None:
            # TODO: optimize later since it reserializes all entries from memory and␣
↪stores them in the JSON file
            #  every time a new vector is inserted
            with open(self.persist_file, "w") as file:
                self.__save_entries_to_file(file)

        return vector_id

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        return self.entries.get(self.__namespaced_vector_id(vector_id,␣
↪namespace=namespace), None)

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        return [entry for key, entry in self.entries.items() if namespace is None or␣
↪entry.namespace == namespace]

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        if namespace:
            entries = {k: v for (k, v) in self.entries.items() if k.startswith(f"
↪{namespace}-")}
        else:
            entries = self.entries

        entries_and_relatednesses = [
            (entry, self.calculate_relatedness(vector, entry.vector)) for entry in␣
↪list(entries.values())
        ]

        entries_and_relatednesses.sort(key=operator.itemgetter(1), reverse=True)

        result = [
            BaseVectorStoreDriver.Entry(
                id=er[0].id, vector=er[0].vector, score=er[1], meta=er[0].meta,␣
↪namespace=er[0].namespace
            )
            for er in entries_and_relatednesses
        ][:count]

        if include_vectors:
            return result
        return [
```

(continues on next page)

(continued from previous page)

```
            BaseVectorStoreDriver.Entry(id=r.id, vector=[], score=r.score, meta=r.
↪meta, namespace=r.namespace)
            for r in result
        ]

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")

    def __save_entries_to_file(self, json_file: TextIO) -> None:
        with self.thread_lock:
            serialized_data = {k: v.to_dict() for k, v in self.entries.items()}

            json.dump(serialized_data, json_file)

    def __namespaced_vector_id(self, vector_id: str, *, namespace: Optional[str]) ->␣
↪str:
        return vector_id if namespace is None else f"{namespace}-{vector_id}"
```

- `calculate_relatedness = field(default=lambda x, y: dot(x, y) / norm(x) * norm(y))` class-attribute instance-attribute
- `entries = field(factory=dict)` class-attribute instance-attribute
- `persist_file = field(default=None)` class-attribute instance-attribute
- `thread_lock = field(default=Factory(lambda: threading.Lock()))` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def __attrs_post_init__(self) -> None:
    if self.persist_file is not None:
        directory = os.path.dirname(self.persist_file)

        if directory and not os.path.exists(directory):
            os.makedirs(directory)

        if not os.path.isfile(self.persist_file):
            with open(self.persist_file, "w") as file:
                self.__save_entries_to_file(file)

        with open(self.persist_file, "r+") as file:
            if os.path.getsize(self.persist_file) > 0:
                self.entries = self.load_entries_from_file(file)
            else:
                self.__save_entries_to_file(file)
```

__namespaced_vector_id(vector_id, *, namespace)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def __namespaced_vector_id(self, vector_id: str, *, namespace: Optional[str]) -> str:
    return vector_id if namespace is None else f"{namespace}-{vector_id}"
```

__save_entries_to_file(json_file)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def __save_entries_to_file(self, json_file: TextIO) -> None:
    with self.thread_lock:
        serialized_data = {k: v.to_dict() for k, v in self.entries.items()}

        json.dump(serialized_data, json_file)
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    return [entry for key, entry in self.entries.items() if namespace is None or␣
↪entry.namespace == namespace]
```

load_entries_from_file(json_file)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def load_entries_from_file(self, json_file: TextIO) -> dict[str,␣
↪BaseVectorStoreDriver.Entry]:
    with self.thread_lock:
        data = json.load(json_file)

        return {k: BaseVectorStoreDriver.Entry.from_dict(v) for k, v in data.items()}
```

load_entry(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    return self.entries.get(self.__namespaced_vector_id(vector_id,␣
↪namespace=namespace), None)
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    if namespace:
        entries = {k: v for (k, v) in self.entries.items() if k.startswith(f"
↪{namespace}-")}
    else:
        entries = self.entries

    entries_and_relatednesses = [
        (entry, self.calculate_relatedness(vector, entry.vector)) for entry in␣
↪list(entries.values())
    ]

    entries_and_relatednesses.sort(key=operator.itemgetter(1), reverse=True)

    result = [
        BaseVectorStoreDriver.Entry(
            id=er[0].id, vector=er[0].vector, score=er[1], meta=er[0].meta,␣
↪namespace=er[0].namespace
        )
        for er in entries_and_relatednesses
    ][:count]

    if include_vectors:
        return result
    return [
        BaseVectorStoreDriver.Entry(id=r.id, vector=[], score=r.score, meta=r.meta,␣
↪namespace=r.namespace)
        for r in result
    ]
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
```

(continues on next page)

(continued from previous page)

```
    **kwargs,
) -> str:
    vector_id = vector_id or utils.str_to_hash(str(vector))

    with self.thread_lock:
        self.entries[self.__namespaced_vector_id(vector_id, namespace=namespace)] =␣
↪self.Entry(
            id=vector_id,
            vector=vector,
            meta=meta,
            namespace=namespace,
        )

    if self.persist_file is not None:
        # TODO: optimize later since it reserializes all entries from memory and␣
↪stores them in the JSON file
        #  every time a new vector is inserted
        with open(self.persist_file, "w") as file:
            self.__save_entries_to_file(file)

    return vector_id
```

local_vector_store_driver

Bases: *BaseVectorStoreDriver* “)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
@define(kw_only=True)
class LocalVectorStoreDriver(BaseVectorStoreDriver):
    entries: dict[str, BaseVectorStoreDriver.Entry] = field(factory=dict)
    persist_file: Optional[str] = field(default=None)
    calculate_relatedness: Callable = field(default=lambda x, y: dot(x, y) / (norm(x)␣
↪* norm(y)))
    thread_lock: threading.Lock = field(default=Factory(lambda: threading.Lock()))

    def __attrs_post_init__(self) -> None:
        if self.persist_file is not None:
            directory = os.path.dirname(self.persist_file)

            if directory and not os.path.exists(directory):
                os.makedirs(directory)

            if not os.path.isfile(self.persist_file):
                with open(self.persist_file, "w") as file:
                    self.__save_entries_to_file(file)

            with open(self.persist_file, "r+") as file:
                if os.path.getsize(self.persist_file) > 0:
                    self.entries = self.load_entries_from_file(file)
                else:
                    self.__save_entries_to_file(file)
```

(continues on next page)

(continued from previous page)

```
    def load_entries_from_file(self, json_file: TextIO) -> dict[str,␣
↪BaseVectorStoreDriver.Entry]:
        with self.thread_lock:
            data = json.load(json_file)

            return {k: BaseVectorStoreDriver.Entry.from_dict(v) for k, v in data.
↪items()}

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        vector_id = vector_id or utils.str_to_hash(str(vector))

        with self.thread_lock:
            self.entries[self.__namespaced_vector_id(vector_id, namespace=namespace)]␣
↪= self.Entry(
                id=vector_id,
                vector=vector,
                meta=meta,
                namespace=namespace,
            )

        if self.persist_file is not None:
            # TODO: optimize later since it reserializes all entries from memory and␣
↪stores them in the JSON file
            #  every time a new vector is inserted
            with open(self.persist_file, "w") as file:
                self.__save_entries_to_file(file)

        return vector_id

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        return self.entries.get(self.__namespaced_vector_id(vector_id,␣
↪namespace=namespace), None)

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        return [entry for key, entry in self.entries.items() if namespace is None or␣
↪entry.namespace == namespace]

    def query_vector(
        self,
        vector: list[float],
        *,
```

(continues on next page)

(continued from previous page)

```
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        if namespace:
            entries = {k: v for (k, v) in self.entries.items() if k.startswith(f"
↪{namespace}-")}
        else:
            entries = self.entries

        entries_and_relatednesses = [
            (entry, self.calculate_relatedness(vector, entry.vector)) for entry in␣
↪list(entries.values())
        ]

        entries_and_relatednesses.sort(key=operator.itemgetter(1), reverse=True)

        result = [
            BaseVectorStoreDriver.Entry(
                id=er[0].id, vector=er[0].vector, score=er[1], meta=er[0].meta,␣
↪namespace=er[0].namespace
            )
            for er in entries_and_relatednesses
        ][:count]

        if include_vectors:
            return result
        return [
            BaseVectorStoreDriver.Entry(id=r.id, vector=[], score=r.score, meta=r.
↪meta, namespace=r.namespace)
            for r in result
        ]

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")

    def __save_entries_to_file(self, json_file: TextIO) -> None:
        with self.thread_lock:
            serialized_data = {k: v.to_dict() for k, v in self.entries.items()}

            json.dump(serialized_data, json_file)

    def __namespaced_vector_id(self, vector_id: str, *, namespace: Optional[str]) ->␣
↪str:
        return vector_id if namespace is None else f"{namespace}-{vector_id}"
```

- `calculate_relatedness = field(default=lambda x, y: dot(x, y) / norm(x) * norm(y))` class-attribute instance-attribute
- `entries = field(factory=dict)` class-attribute instance-attribute
- `persist_file = field(default=None)` class-attribute instance-attribute

- `thread_lock = field(default=Factory(lambda: threading.Lock()))` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def __attrs_post_init__(self) -> None:
    if self.persist_file is not None:
        directory = os.path.dirname(self.persist_file)

        if directory and not os.path.exists(directory):
            os.makedirs(directory)

        if not os.path.isfile(self.persist_file):
            with open(self.persist_file, "w") as file:
                self.__save_entries_to_file(file)

        with open(self.persist_file, "r+") as file:
            if os.path.getsize(self.persist_file) > 0:
                self.entries = self.load_entries_from_file(file)
            else:
                self.__save_entries_to_file(file)
```

__namespaced_vector_id(vector_id, *, namespace)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def __namespaced_vector_id(self, vector_id: str, *, namespace: Optional[str]) -> str:
    return vector_id if namespace is None else f"{namespace}-{vector_id}"
```

__save_entries_to_file(json_file)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def __save_entries_to_file(self, json_file: TextIO) -> None:
    with self.thread_lock:
        serialized_data = {k: v.to_dict() for k, v in self.entries.items()}

        json.dump(serialized_data, json_file)
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    return [entry for key, entry in self.entries.items() if namespace is None or␣
↪entry.namespace == namespace]
```

load_entries_from_file(json_file)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def load_entries_from_file(self, json_file: TextIO) -> dict[str,␣
↪BaseVectorStoreDriver.Entry]:
    with self.thread_lock:
        data = json.load(json_file)

        return {k: BaseVectorStoreDriver.Entry.from_dict(v) for k, v in data.items()}
```

load_entry(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    return self.entries.get(self.__namespaced_vector_id(vector_id,␣
↪namespace=namespace), None)
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    if namespace:
        entries = {k: v for (k, v) in self.entries.items() if k.startswith(f"
↪{namespace}-")}
    else:
        entries = self.entries

    entries_and_relatednesses = [
        (entry, self.calculate_relatedness(vector, entry.vector)) for entry in␣
↪list(entries.values())
    ]
```

(continues on next page)

(continued from previous page)

```
    entries_and_relatednesses.sort(key=operator.itemgetter(1), reverse=True)

    result = [
        BaseVectorStoreDriver.Entry(
            id=er[0].id, vector=er[0].vector, score=er[1], meta=er[0].meta,␣
↪namespace=er[0].namespace
        )
        for er in entries_and_relatednesses
    ][:count]

    if include_vectors:
        return result
    return [
        BaseVectorStoreDriver.Entry(id=r.id, vector=[], score=r.score, meta=r.meta,␣
↪namespace=r.namespace)
        for r in result
    ]
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/local_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    vector_id = vector_id or utils.str_to_hash(str(vector))

    with self.thread_lock:
        self.entries[self.__namespaced_vector_id(vector_id, namespace=namespace)] =␣
↪self.Entry(
            id=vector_id,
            vector=vector,
            meta=meta,
            namespace=namespace,
        )

    if self.persist_file is not None:
        # TODO: optimize later since it reserializes all entries from memory and␣
↪stores them in the JSON file
        #  every time a new vector is inserted
        with open(self.persist_file, "w") as file:
            self.__save_entries_to_file(file)

    return vector_id
```

marqo

- `__all__ = ['MarqoVectorStoreDriver']` module-attribute

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="marqo.mdx#griptape.drivers.vector.marqo.MarqoVectorStoreDriver.api_key" title="api_key = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.marqo.MarqoVectorStoreDriver.api_key)"><c class="language-python">	*str*	The API key for the Marqo API.
<a href="marqo.mdx#griptape.drivers.vector.marqo.MarqoVectorStoreDriver.url" title="url = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.marqo.MarqoVectorStoreDriver.url)"><code class="language-python">	*str*	The URL to the Marqo API.
<a href="marqo.mdx#griptape.drivers.vector.marqo.MarqoVectorStoreDriver.client" title="client() (griptape.drivers.vector.marqo.MarqoVectorStoreDriver.client)"><co class="language-python">	*Client*	An optional Marqo client. Defaults to a new client with the given URL and API key.

continues on next page

Table 242 – continued from previous page

Name	Type	Description
<a href="marqo.mdx#griptape.drivers.vector.marqo.MarqoVectorStoreDriver.index" title="index = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.marqo.MarqoVectorStoreDriver.index)"><code class="language-python">	*str*	The name of the index to use.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
@define
class MarqoVectorStoreDriver(BaseVectorStoreDriver):
    """A Vector Store Driver for Marqo.

    Attributes:
        api_key: The API key for the Marqo API.
        url: The URL to the Marqo API.
        client: An optional Marqo client. Defaults to a new client with the given URL
↪and API key.
        index: The name of the index to use.
    """

    api_key: str = field(kw_only=True, metadata={"serializable": True})
    url: str = field(kw_only=True, metadata={"serializable": True})
    index: str = field(kw_only=True, metadata={"serializable": True})
    _client: Optional[marqo.Client] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> marqo.Client:
        return import_optional_dependency("marqo").Client(self.url, api_key=self.api_
↪key)

    def upsert(
        self,
        value: str | TextArtifact | ImageArtifact,
        *,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        vector_id: Optional[str] = None,
```

(continues on next page)

(continued from previous page)

```
        **kwargs: Any,
    ) -> str:
        """Upsert a text document into the Marqo index.

        Args:
            value: The value to be indexed.
            namespace: An optional namespace for the document.
            meta: An optional dictionary of metadata for the document.
            vector_id: The ID for the vector. If None, Marqo will generate an ID.
            kwargs: Additional keyword arguments to pass to the Marqo client.

        Returns:
            str: The ID of the document that was added.
        """
        if isinstance(value, TextArtifact):
            artifact_json = value.to_json()
            vector_id = utils.str_to_hash(value.value) if vector_id is None else␣
↪vector_id

            doc = {
                "_id": vector_id,
                "Description": value.value,
                "artifact": str(artifact_json),
            }
        elif isinstance(value, ImageArtifact):
            raise NotImplementedError("`MarqoVectorStoreDriver` does not upserting␣
↪Image Artifacts.")
        else:
            doc = {"_id": vector_id, "Description": value}

        # Non-tensor fields
        if meta:
            doc["meta"] = str(meta)
        if namespace:
            doc["namespace"] = namespace

        response = self.client.index(self.index).add_documents([doc], tensor_fields=[
↪"Description"])
        if isinstance(response, dict) and "items" in response and response["items"]:
            return response["items"][0]["_id"]
        raise ValueError(f"Failed to upsert text: {response}")

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        """Load a document entry from the Marqo index.

        Args:
            vector_id: The ID of the vector to load.
            namespace: The namespace of the vector to load.

        Returns:
            The loaded Entry if found, otherwise None.
```

(continues on next page)

(continued from previous page)

```
        """
        result = self.client.index(self.index).get_document(document_id=vector_id,␣
↪expose_facets=True)

        if result and "_tensor_facets" in result and len(result["_tensor_facets"]) >␣
↪0:
            return BaseVectorStoreDriver.Entry(
                id=result["_id"],
                meta={k: v for k, v in result.items() if k != "_id"},
                vector=result["_tensor_facets"][0]["_embedding"],
            )
        return None

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Load all document entries from the Marqo index.

        Args:
            namespace: The namespace to filter entries by.

        Returns:
            The list of loaded Entries.
        """
        filter_string = f"namespace:{namespace}" if namespace else None

        if filter_string is not None:
            results = self.client.index(self.index).search("", limit=10000, filter_
↪string=filter_string)
        else:
            results = self.client.index(self.index).search("", limit=10000)

        # get all _ids from search results
        ids = [r["_id"] for r in results["hits"]]

        # get documents corresponding to the ids
        documents = self.client.index(self.index).get_documents(document_ids=ids,␣
↪expose_facets=True)

        # for each document, if its found, create an Entry object
        entries = []
        for doc in documents["results"]:
            if doc["_found"]:
                entries.append(
                    BaseVectorStoreDriver.Entry(
                        id=doc["_id"],
                        vector=doc["_tensor_facets"][0]["_embedding"],
                        meta={k: v for k, v in doc.items() if k not in ["_id", "_
↪tensor_facets", "_found"]},
                        namespace=doc.get("namespace"),
                    ),
                )
```

(continues on next page)

(continued from previous page)

```
        return entries

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        include_metadata: bool = True,
        **kwargs: Any,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Query the Marqo index for documents.

        Args:
            vector: The vector to query by.
            count: The maximum number of results to return.
            namespace: The namespace to filter results by.
            include_vectors: Whether to include vector data in the results.
            include_metadata: Whether to include metadata in the results.
            kwargs: Additional keyword arguments to pass to the Marqo client.

        Returns:
            The list of query results.
        """
        params = {
            "limit": count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
            "attributes_to_retrieve": None if include_metadata else ["_id"],
            "filter_string": f"namespace:{namespace}" if namespace else None,
        } | kwargs

        results = self.client.index(self.index).search(**params, context={"tensor":␣
↪[vector], "weight": 1})

        return self.__process_results(results, include_vectors=include_vectors)

    def query(
        self,
        query: str | TextArtifact | ImageArtifact,
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        include_metadata: bool = True,
        **kwargs: Any,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Query the Marqo index for documents.

        Args:
            query: The query string.
            count: The maximum number of results to return.
            namespace: The namespace to filter results by.
```

(continues on next page)

(continued from previous page)

```
            include_vectors: Whether to include vector data in the results.
            include_metadata: Whether to include metadata in the results.
            kwargs: Additional keyword arguments to pass to the Marqo client.

        Returns:
            The list of query results.
        """
        params = {
            "limit": count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
            "attributes_to_retrieve": None if include_metadata else ["_id"],
            "filter_string": f"namespace:{namespace}" if namespace else None,
        } | kwargs

        results = self.client.index(self.index).search(str(query), **params)
        return self.__process_results(results, include_vectors=include_vectors)

    def delete_index(self, name: str) -> dict[str, Any]:
        """Delete an index in the Marqo client.

        Args:
            name: The name of the index to delete.
        """
        return self.client.delete_index(name)

    def get_indexes(self) -> list[str]:
        """Get a list of all indexes in the Marqo client.

        Returns:
            The list of all indexes.
        """
        return [index["index"] for index in self.client.get_indexes()["results"]]

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs: Any,
    ) -> str:
        """Upsert a vector into the Marqo index.

        Args:
            vector: The vector to be indexed.
            vector_id: The ID for the vector. If None, Marqo will generate an ID.
            namespace: An optional namespace for the vector.
            meta: An optional dictionary of metadata for the vector.
            kwargs: Additional keyword arguments to pass to the Marqo client.

        Raises:
            Exception: This function is not yet implemented.
```

(continues on next page)

(continued from previous page)

```
        Returns:
            The ID of the vector that was added.
        """
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪upserting a vector.")

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")

    def __process_results(self, results: dict, *, include_vectors: bool) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        if include_vectors:
            results["hits"] = [
                {**r, **self.client.index(self.index).get_document(r["_id"], expose_
↪facets=True)}
                for r in results["hits"]
            ]

        return [
            BaseVectorStoreDriver.Entry(
                id=r["_id"],
                vector=r["_tensor_facets"][0]["_embedding"] if include_vectors else␣
↪[],
                score=r["_score"],
                meta={k: v for k, v in r.items() if k not in ["_score", "_tensor_
↪facets"]},
            )
            for r in results["hits"]
        ]
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `index = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `url = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__process_results(results, *, include_vectors)

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def __process_results(self, results: dict, *, include_vectors: bool) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    if include_vectors:
        results["hits"] = [
            {**r, **self.client.index(self.index).get_document(r["_id"], expose_
↪facets=True)}
            for r in results["hits"]
```

(continues on next page)

(continued from previous page)

```
        ]

    return [
        BaseVectorStoreDriver.Entry(
            id=r["_id"],
            vector=r["_tensor_facets"][0]["_embedding"] if include_vectors else [],
            score=r["_score"],
            meta={k: v for k, v in r.items() if k not in ["_score", "_tensor_facets"]}
↪,
        )
        for r in results["hits"]
    ]
```

client()

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
@lazy_property()
def client(self) -> marqo.Client:
    return import_optional_dependency("marqo").Client(self.url, api_key=self.api_key)
```

delete_index(name)

Delete an index in the Marqo client.

Parameters

Name	Type	Description	Default
name	*str*	The name of the index to delete. 	*required*

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def delete_index(self, name: str) -> dict[str, Any]:
    """Delete an index in the Marqo client.

    Args:
        name: The name of the index to delete.
    """
    return self.client.delete_index(name)
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

get_indexes()

Get a list of all indexes in the Marqo client.

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	The list of all indexes.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def get_indexes(self) -> list[str]:
    """Get a list of all indexes in the Marqo client.

    Returns:
        The list of all indexes.
    """
    return [index["index"] for index in self.client.get_indexes()["results"]]
```

load_entries(*, namespace=None)

Load all document entries from the Marqo index.

Parameters

Name	Type	Description	Default
namespace	*Optional[str]*	The namespace to filter entries by. 	*None*

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	The list of loaded Entries.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Load all document entries from the Marqo index.
```

(continues on next page)

(continued from previous page)

```
    Args:
        namespace: The namespace to filter entries by.

    Returns:
        The list of loaded Entries.
    """
    filter_string = f"namespace:{namespace}" if namespace else None

    if filter_string is not None:
        results = self.client.index(self.index).search("", limit=10000, filter_
↪string=filter_string)
    else:
        results = self.client.index(self.index).search("", limit=10000)

    # get all _ids from search results
    ids = [r["_id"] for r in results["hits"]]

    # get documents corresponding to the ids
    documents = self.client.index(self.index).get_documents(document_ids=ids, expose_
↪facets=True)

    # for each document, if its found, create an Entry object
    entries = []
    for doc in documents["results"]:
        if doc["_found"]:
            entries.append(
                BaseVectorStoreDriver.Entry(
                    id=doc["_id"],
                    vector=doc["_tensor_facets"][0]["_embedding"],
                    meta={k: v for k, v in doc.items() if k not in ["_id", "_tensor_
↪facets", "_found"]},
                    namespace=doc.get("namespace"),
                ),
            )

    return entries
```

load_entry(vector_id, *, namespace=None)

Load a document entry from the Marqo index.

Parameters

Name	Type	Description	Default
vector_id	*str*	The ID of the vector to load. 	*required*
namespace	*Optional[str]*	The namespace of the vector to load. 	*None*

Returns

Type	Description
<a href="./#griptape.drivers.vector.BaseVectorStoreDriver.Entry" title="Entry (griptape.drivers.vector.BaseVectorStoreDriver.Entry)"><code class="language-python">C	The loaded Entry if found, otherwise None.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    """Load a document entry from the Marqo index.

    Args:
        vector_id: The ID of the vector to load.
        namespace: The namespace of the vector to load.

    Returns:
        The loaded Entry if found, otherwise None.
    """
    result = self.client.index(self.index).get_document(document_id=vector_id, expose_
↪facets=True)

    if result and "_tensor_facets" in result and len(result["_tensor_facets"]) > 0:
        return BaseVectorStoreDriver.Entry(
            id=result["_id"],
            meta={k: v for k, v in result.items() if k != "_id"},
            vector=result["_tensor_facets"][0]["_embedding"],
        )
    return None
```

query(query, *, count=None, namespace=None, include_vectors=False, include_metadata=True, **kwargs)

Query the Marqo index for documents.

Parameters

Name	Type	Description	Default
query	<code class="language-pythc I TextArtifact I ImageArtifact</code>	The query string. 	*required*
count	*Optional[int]*	The maximum number of results to return. 	*None*
namespace	*Optional[str]*	The namespace to filter results by. 	*None*
include_vectors	*bool*	Whether to include vector data in the results. 	*False*
include_metadata	*bool*	Whether to include metadata in the results. 	*True*
kwargs	*Any*	Additional keyword arguments to pass to the Marqo client. 	*{}*

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	The list of query results.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def query(
    self,
    query: str | TextArtifact | ImageArtifact,
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    include_metadata: bool = True,
    **kwargs: Any,
) -> list[BaseVectorStoreDriver.Entry]:
    """Query the Marqo index for documents.
```

(continues on next page)

(continued from previous page)

```
    Args:
        query: The query string.
        count: The maximum number of results to return.
        namespace: The namespace to filter results by.
        include_vectors: Whether to include vector data in the results.
        include_metadata: Whether to include metadata in the results.
        kwargs: Additional keyword arguments to pass to the Marqo client.

    Returns:
        The list of query results.
    """
    params = {
        "limit": count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
        "attributes_to_retrieve": None if include_metadata else ["_id"],
        "filter_string": f"namespace:{namespace}" if namespace else None,
    } | kwargs

    results = self.client.index(self.index).search(str(query), **params)
    return self.__process_results(results, include_vectors=include_vectors)
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, include_metadata=True, **kwargs)

Query the Marqo index for documents.

Parameters

Name	Type	Description	Default
vector	<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	The vector to query by. 	*required*
count	*Optional[int]*	The maximum number of results to return. 	*None*
namespace	*Optional[str]*	The namespace to filter results by. 	*None*
include_vectors	*bool*	Whether to include vector data in the results. 	*False*
include_metadata	*bool*	Whether to include metadata in the results. 	*True*
kwargs	*Any*	Additional keyword arguments to pass to the Marqo client. 	*{}*

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	The list of query results.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    include_metadata: bool = True,
    **kwargs: Any,
) -> list[BaseVectorStoreDriver.Entry]:
    """Query the Marqo index for documents.

    Args:
        vector: The vector to query by.
        count: The maximum number of results to return.
        namespace: The namespace to filter results by.
        include_vectors: Whether to include vector data in the results.
        include_metadata: Whether to include metadata in the results.
        kwargs: Additional keyword arguments to pass to the Marqo client.

    Returns:
        The list of query results.
    """
    params = {
        "limit": count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
        "attributes_to_retrieve": None if include_metadata else ["_id"],
        "filter_string": f"namespace:{namespace}" if namespace else None,
    } | kwargs

    results = self.client.index(self.index).search(**params, context={"tensor":␣
↪[vector], "weight": 1})

    return self.__process_results(results, include_vectors=include_vectors)
```

upsert(value, *, namespace=None, meta=None, vector_id=None, **kwargs)

Upsert a text document into the Marqo index.

Parameters

Name	Type	Description	Default
value	<code class="language-pythc I TextArtifact I ImageArtifact</code>	The value to be indexed. 	*required*
namespace	*Optional[str]*	An optional namespace for the document. 	*None*
meta	*Optional[dict]*	An optional dictionary of metadata for the document. 	*None*
vector_id	*Optional[str]*	The ID for the vector. If None, Marqo will generate an ID. 	*None*
kwargs	*Any*	Additional keyword arguments to pass to the Marqo client. 	*{}*

Returns

Name	Type	Description
str	*str*	The ID of the document that was added.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def upsert(
    self,
    value: str | TextArtifact | ImageArtifact,
    *,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    vector_id: Optional[str] = None,
    **kwargs: Any,
) -> str:
    """Upsert a text document into the Marqo index.

    Args:
        value: The value to be indexed.
```

(continues on next page)

(continued from previous page)

```
        namespace: An optional namespace for the document.
        meta: An optional dictionary of metadata for the document.
        vector_id: The ID for the vector. If None, Marqo will generate an ID.
        kwargs: Additional keyword arguments to pass to the Marqo client.

    Returns:
        str: The ID of the document that was added.
    """
    if isinstance(value, TextArtifact):
        artifact_json = value.to_json()
        vector_id = utils.str_to_hash(value.value) if vector_id is None else vector_id

        doc = {
            "_id": vector_id,
            "Description": value.value,
            "artifact": str(artifact_json),
        }
    elif isinstance(value, ImageArtifact):
        raise NotImplementedError("`MarqoVectorStoreDriver` does not upserting Image␣
↪Artifacts.")
    else:
        doc = {"_id": vector_id, "Description": value}

    # Non-tensor fields
    if meta:
        doc["meta"] = str(meta)
    if namespace:
        doc["namespace"] = namespace

    response = self.client.index(self.index).add_documents([doc], tensor_fields=[
↪"Description"])
    if isinstance(response, dict) and "items" in response and response["items"]:
        return response["items"][0]["_id"]
    raise ValueError(f"Failed to upsert text: {response}")
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Upsert a vector into the Marqo index.

Parameters

Name	Type	Description	Default
vector	<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	The vector to be indexed. 	*required*

continues on next page

Table 255 – continued from previous page

Name	Type	Description	Default
vector_id	*Optional[str]*	The ID for the vector. If None, Marqo will generate an ID. 	*None*
namespace	*Optional[str]*	An optional namespace for the vector. 	*None*
meta	*Optional[dict]*	An optional dictionary of metadata for the vector. 	*None*
kwargs	*Any*	Additional keyword arguments to pass to the Marqo client. 	*{}*

Raises

Returns

Type	Description
str	The ID of the vector that was added.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs: Any,
) -> str:
    """Upsert a vector into the Marqo index.

    Args:
        vector: The vector to be indexed.
        vector_id: The ID for the vector. If None, Marqo will generate an ID.
        namespace: An optional namespace for the vector.
        meta: An optional dictionary of metadata for the vector.
        kwargs: Additional keyword arguments to pass to the Marqo client.

    Raises:
        Exception: This function is not yet implemented.

    Returns:
        The ID of the vector that was added.
    """
```

(continues on next page)

(continued from previous page)

```
    raise NotImplementedError(f"{self.__class__.__name__} does not support upserting␣
↪a vector.")
```

marqo_vector_store_driver

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="marqo_vector_store_driver.mdx#griptape.drivers.vector.marqo_vector_store_driver.MarqoVectorStoreDriver.api_key" title="api_key = field(kw_only=True, metadata={'serializable': True})">class-attribute instance-attribute (griptape.drivers.vector.marqo_vector_store_driver.MarqoVectorStoreDriver.api_key)"><c class="language-python">	*str*	The API key for the Marqo API.
<a href="marqo_vector_store_driver.mdx#griptape.drivers.vector.marqo_vector_store_driver.MarqoVectorStoreDriver.url" title="url = field(kw_only=True, metadata={'serializable': True})">class-attribute instance-attribute (griptape.drivers.vector.marqo_vector_store_driver.MarqoVectorStoreDriver.url)"><code class="language-python">	*str*	The URL to the Marqo API.

continues on next page

Table 257 – continued from previous page

Name	Type	Description
<a href="marqo_vector_store_driver.mdx#griptape.drivers.vector.marqo_vector_store_driver.MarqoVectorStoreDriver.client" title="client() (griptape.drivers.vector.marqo_vector_store_driver.MarqoVectorStoreDriver.client)"><code class="language-python">	*Client*	An optional Marqo client. Defaults to a new client with the given URL and API key.
<a href="marqo_vector_store_driver.mdx#griptape.drivers.vector.marqo_vector_store_driver.MarqoVectorStoreDriver.index" title="index = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.marqo_vector_store_driver.MarqoVectorStoreDriver.index)"><code class="language-python">	*str*	The name of the index to use.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
@define
class MarqoVectorStoreDriver(BaseVectorStoreDriver):
    """A Vector Store Driver for Marqo.

    Attributes:
        api_key: The API key for the Marqo API.
        url: The URL to the Marqo API.
        client: An optional Marqo client. Defaults to a new client with the given URL
↪and API key.
        index: The name of the index to use.
    """

    api_key: str = field(kw_only=True, metadata={"serializable": True})
    url: str = field(kw_only=True, metadata={"serializable": True})
    index: str = field(kw_only=True, metadata={"serializable": True})
```

(continues on next page)

(continued from previous page)

```
    _client: Optional[marqo.Client] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> marqo.Client:
        return import_optional_dependency("marqo").Client(self.url, api_key=self.api_
↪key)

    def upsert(
        self,
        value: str | TextArtifact | ImageArtifact,
        *,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        vector_id: Optional[str] = None,
        **kwargs: Any,
    ) -> str:
        """Upsert a text document into the Marqo index.

        Args:
            value: The value to be indexed.
            namespace: An optional namespace for the document.
            meta: An optional dictionary of metadata for the document.
            vector_id: The ID for the vector. If None, Marqo will generate an ID.
            kwargs: Additional keyword arguments to pass to the Marqo client.

        Returns:
            str: The ID of the document that was added.
        """
        if isinstance(value, TextArtifact):
            artifact_json = value.to_json()
            vector_id = utils.str_to_hash(value.value) if vector_id is None else␣
↪vector_id

            doc = {
                "_id": vector_id,
                "Description": value.value,
                "artifact": str(artifact_json),
            }
        elif isinstance(value, ImageArtifact):
            raise NotImplementedError("`MarqoVectorStoreDriver` does not upserting␣
↪Image Artifacts.")
        else:
            doc = {"_id": vector_id, "Description": value}

        # Non-tensor fields
        if meta:
            doc["meta"] = str(meta)
        if namespace:
            doc["namespace"] = namespace
```

(continues on next page)

(continued from previous page)

```
        response = self.client.index(self.index).add_documents([doc], tensor_fields=[
↪"Description"])
        if isinstance(response, dict) and "items" in response and response["items"]:
            return response["items"][0]["_id"]
        raise ValueError(f"Failed to upsert text: {response}")

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        """Load a document entry from the Marqo index.

        Args:
            vector_id: The ID of the vector to load.
            namespace: The namespace of the vector to load.

        Returns:
            The loaded Entry if found, otherwise None.
        """
        result = self.client.index(self.index).get_document(document_id=vector_id,␣
↪expose_facets=True)

        if result and "_tensor_facets" in result and len(result["_tensor_facets"]) >␣
↪0:
            return BaseVectorStoreDriver.Entry(
                id=result["_id"],
                meta={k: v for k, v in result.items() if k != "_id"},
                vector=result["_tensor_facets"][0]["_embedding"],
            )
        return None

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Load all document entries from the Marqo index.

        Args:
            namespace: The namespace to filter entries by.

        Returns:
            The list of loaded Entries.
        """
        filter_string = f"namespace:{namespace}" if namespace else None

        if filter_string is not None:
            results = self.client.index(self.index).search("", limit=10000, filter_
↪string=filter_string)
        else:
            results = self.client.index(self.index).search("", limit=10000)

        # get all _ids from search results
        ids = [r["_id"] for r in results["hits"]]

        # get documents corresponding to the ids
        documents = self.client.index(self.index).get_documents(document_ids=ids,␣
```

(continues on next page)

(continued from previous page)

```
↪expose_facets=True)

        # for each document, if its found, create an Entry object
        entries = []
        for doc in documents["results"]:
            if doc["_found"]:
                entries.append(
                    BaseVectorStoreDriver.Entry(
                        id=doc["_id"],
                        vector=doc["_tensor_facets"][0]["_embedding"],
                        meta={k: v for k, v in doc.items() if k not in ["_id", "_
↪tensor_facets", "_found"]},
                        namespace=doc.get("namespace"),
                    ),
                )

        return entries

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        include_metadata: bool = True,
        **kwargs: Any,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Query the Marqo index for documents.

        Args:
            vector: The vector to query by.
            count: The maximum number of results to return.
            namespace: The namespace to filter results by.
            include_vectors: Whether to include vector data in the results.
            include_metadata: Whether to include metadata in the results.
            kwargs: Additional keyword arguments to pass to the Marqo client.

        Returns:
            The list of query results.
        """
        params = {
            "limit": count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
            "attributes_to_retrieve": None if include_metadata else ["_id"],
            "filter_string": f"namespace:{namespace}" if namespace else None,
        } | kwargs

        results = self.client.index(self.index).search(**params, context={"tensor":␣
↪[vector], "weight": 1})

        return self.__process_results(results, include_vectors=include_vectors)
```

(continues on next page)

(continued from previous page)

```
    def query(
        self,
        query: str | TextArtifact | ImageArtifact,
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        include_metadata: bool = True,
        **kwargs: Any,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Query the Marqo index for documents.

        Args:
            query: The query string.
            count: The maximum number of results to return.
            namespace: The namespace to filter results by.
            include_vectors: Whether to include vector data in the results.
            include_metadata: Whether to include metadata in the results.
            kwargs: Additional keyword arguments to pass to the Marqo client.

        Returns:
            The list of query results.
        """
        params = {
            "limit": count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
            "attributes_to_retrieve": None if include_metadata else ["_id"],
            "filter_string": f"namespace:{namespace}" if namespace else None,
        } | kwargs

        results = self.client.index(self.index).search(str(query), **params)
        return self.__process_results(results, include_vectors=include_vectors)

    def delete_index(self, name: str) -> dict[str, Any]:
        """Delete an index in the Marqo client.

        Args:
            name: The name of the index to delete.
        """
        return self.client.delete_index(name)

    def get_indexes(self) -> list[str]:
        """Get a list of all indexes in the Marqo client.

        Returns:
            The list of all indexes.
        """
        return [index["index"] for index in self.client.get_indexes()["results"]]

    def upsert_vector(
        self,
        vector: list[float],
        *,
```

(continues on next page)

(continued from previous page)

```
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs: Any,
    ) -> str:
        """Upsert a vector into the Marqo index.

        Args:
            vector: The vector to be indexed.
            vector_id: The ID for the vector. If None, Marqo will generate an ID.
            namespace: An optional namespace for the vector.
            meta: An optional dictionary of metadata for the vector.
            kwargs: Additional keyword arguments to pass to the Marqo client.

        Raises:
            Exception: This function is not yet implemented.

        Returns:
            The ID of the vector that was added.
        """
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪upserting a vector.")

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")

    def __process_results(self, results: dict, *, include_vectors: bool) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        if include_vectors:
            results["hits"] = [
                {**r, **self.client.index(self.index).get_document(r["_id"], expose_
↪facets=True)}
                for r in results["hits"]
            ]

        return [
            BaseVectorStoreDriver.Entry(
                id=r["_id"],
                vector=r["_tensor_facets"][0]["_embedding"] if include_vectors else␣
↪[],
                score=r["_score"],
                meta={k: v for k, v in r.items() if k not in ["_score", "_tensor_
↪facets"]},
            )
            for r in results["hits"]
        ]
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

- `index = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `url = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

__process_results(results, *, include_vectors)

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def __process_results(self, results: dict, *, include_vectors: bool) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    if include_vectors:
        results["hits"] = [
            {**r, **self.client.index(self.index).get_document(r["_id"], expose_
↪facets=True)}
            for r in results["hits"]
        ]

    return [
        BaseVectorStoreDriver.Entry(
            id=r["_id"],
            vector=r["_tensor_facets"][0]["_embedding"] if include_vectors else [],
            score=r["_score"],
            meta={k: v for k, v in r.items() if k not in ["_score", "_tensor_facets"]}
↪,
        )
        for r in results["hits"]
    ]
```

client()

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
@lazy_property()
def client(self) -> marqo.Client:
    return import_optional_dependency("marqo").Client(self.url, api_key=self.api_key)
```

delete_index(name)

Delete an index in the Marqo client.

Parameters

Name	Type	Description	Default
name	*str*	The name of the index to delete. 	*required*

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def delete_index(self, name: str) -> dict[str, Any]:
    """Delete an index in the Marqo client.
```

(continues on next page)

(continued from previous page)

```
    Args:
        name: The name of the index to delete.
    """
    return self.client.delete_index(name)
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

get_indexes()

Get a list of all indexes in the Marqo client.

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	The list of all indexes.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def get_indexes(self) -> list[str]:
    """Get a list of all indexes in the Marqo client.

    Returns:
        The list of all indexes.
    """
    return [index["index"] for index in self.client.get_indexes()["results"]]
```

load_entries(*, namespace=None)

Load all document entries from the Marqo index.

Parameters

Name	Type	Description	Default
namespace	*Optional[str]*	The namespace to filter entries by. 	*None*

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	The list of loaded Entries.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Load all document entries from the Marqo index.

    Args:
        namespace: The namespace to filter entries by.

    Returns:
        The list of loaded Entries.
    """
    filter_string = f"namespace:{namespace}" if namespace else None

    if filter_string is not None:
        results = self.client.index(self.index).search("", limit=10000, filter_
↪string=filter_string)
    else:
        results = self.client.index(self.index).search("", limit=10000)

    # get all _ids from search results
    ids = [r["_id"] for r in results["hits"]]

    # get documents corresponding to the ids
    documents = self.client.index(self.index).get_documents(document_ids=ids, expose_
↪facets=True)

    # for each document, if its found, create an Entry object
    entries = []
    for doc in documents["results"]:
        if doc["_found"]:
            entries.append(
                BaseVectorStoreDriver.Entry(
                    id=doc["_id"],
                    vector=doc["_tensor_facets"][0]["_embedding"],
                    meta={k: v for k, v in doc.items() if k not in ["_id", "_tensor_
↪facets", "_found"]},
                    namespace=doc.get("namespace"),
                ),
            )

    return entries
```

load_entry(vector_id, *, namespace=None)

Load a document entry from the Marqo index.

Parameters

Name	Type	Description	Default
vector_id	*str*	The ID of the vector to load. 	*required*
namespace	*Optional[str]*	The namespace of the vector to load. 	*None*

Returns

Type	Description
<a href="./#griptape.drivers.vector.BaseVectorStoreDriver.Entry" title="Entry (griptape.drivers.vector.BaseVectorStoreDriver.Entry)"><code class="language-python">C	The loaded Entry if found, otherwise None.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    """Load a document entry from the Marqo index.

    Args:
        vector_id: The ID of the vector to load.
        namespace: The namespace of the vector to load.

    Returns:
        The loaded Entry if found, otherwise None.
    """
    result = self.client.index(self.index).get_document(document_id=vector_id, expose_
↪facets=True)

    if result and "_tensor_facets" in result and len(result["_tensor_facets"]) > 0:
        return BaseVectorStoreDriver.Entry(
            id=result["_id"],
            meta={k: v for k, v in result.items() if k != "_id"},
            vector=result["_tensor_facets"][0]["_embedding"],
        )
    return None
```

query(query, *, count=None, namespace=None, include_vectors=False, include_metadata=True, **kwargs)

Query the Marqo index for documents.

Parameters

Name	Type	Description	Default
query	<code class="language-pythc I TextArtifact I ImageArtifact</code>	The query string. 	*required*
count	*Optional[int]*	The maximum number of results to return. 	*None*
namespace	*Optional[str]*	The namespace to filter results by. 	*None*
include_vectors	*bool*	Whether to include vector data in the results. 	*False*
include_metadata	*bool*	Whether to include metadata in the results. 	*True*
kwargs	*Any*	Additional keyword arguments to pass to the Marqo client. 	*{}*

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	The list of query results.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def query(
    self,
    query: str | TextArtifact | ImageArtifact,
    *,
    count: Optional[int] = None,
```

(continues on next page)

(continued from previous page)

```
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    include_metadata: bool = True,
    **kwargs: Any,
) -> list[BaseVectorStoreDriver.Entry]:
    """Query the Marqo index for documents.

    Args:
        query: The query string.
        count: The maximum number of results to return.
        namespace: The namespace to filter results by.
        include_vectors: Whether to include vector data in the results.
        include_metadata: Whether to include metadata in the results.
        kwargs: Additional keyword arguments to pass to the Marqo client.

    Returns:
        The list of query results.
    """
    params = {
        "limit": count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
        "attributes_to_retrieve": None if include_metadata else ["_id"],
        "filter_string": f"namespace:{namespace}" if namespace else None,
    } | kwargs

    results = self.client.index(self.index).search(str(query), **params)
    return self.__process_results(results, include_vectors=include_vectors)
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, include_metadata=True, **kwargs)

Query the Marqo index for documents.

Parameters

Name	Type	Description	Default
vector	<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	The vector to query by. 	*required*
count	*Optional[int]*	The maximum number of results to return. 	*None*
namespace	*Optional[str]*	The namespace to filter results by. 	*None*
include_vectors	*bool*	Whether to include vector data in the results. 	*False*

continues on next page

Table 266 – continued from previous page

Name	Type	Description	Default
include_metadata	*bool*	Whether to include metadata in the results. 	*True*
kwargs	*Any*	Additional keyword arguments to pass to the Marqo client. 	*{}*

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	The list of query results.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    include_metadata: bool = True,
    **kwargs: Any,
) -> list[BaseVectorStoreDriver.Entry]:
    """Query the Marqo index for documents.

    Args:
        vector: The vector to query by.
        count: The maximum number of results to return.
        namespace: The namespace to filter results by.
        include_vectors: Whether to include vector data in the results.
        include_metadata: Whether to include metadata in the results.
        kwargs: Additional keyword arguments to pass to the Marqo client.

    Returns:
        The list of query results.
    """
    params = {
        "limit": count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
        "attributes_to_retrieve": None if include_metadata else ["_id"],
        "filter_string": f"namespace:{namespace}" if namespace else None,
    } | kwargs

    results = self.client.index(self.index).search(**params, context={"tensor":␣
```

(continues on next page)

(continued from previous page)

```
↪[vector], "weight": 1})

    return self.__process_results(results, include_vectors=include_vectors)
```

upsert(value, *, namespace=None, meta=None, vector_id=None, **kwargs)

Upsert a text document into the Marqo index.

Parameters

Name	Type	Description	Default
value	<code class="language-pythc I TextArtifact I ImageArtifact</code>	The value to be indexed. 	*required*
namespace	*Optional[str]*	An optional namespace for the document. 	*None*
meta	*Optional[dict]*	An optional dictionary of metadata for the document. 	*None*
vector_id	*Optional[str]*	The ID for the vector. If None, Marqo will generate an ID. 	*None*
kwargs	*Any*	Additional keyword arguments to pass to the Marqo client. 	*{}*

Returns

Name	Type	Description
str	*str*	The ID of the document that was added.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def upsert(
    self,
    value: str | TextArtifact | ImageArtifact,
```

(continues on next page)

(continued from previous page)

```
    *,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    vector_id: Optional[str] = None,
    **kwargs: Any,
) -> str:
    """Upsert a text document into the Marqo index.

    Args:
        value: The value to be indexed.
        namespace: An optional namespace for the document.
        meta: An optional dictionary of metadata for the document.
        vector_id: The ID for the vector. If None, Marqo will generate an ID.
        kwargs: Additional keyword arguments to pass to the Marqo client.

    Returns:
        str: The ID of the document that was added.
    """
    if isinstance(value, TextArtifact):
        artifact_json = value.to_json()
        vector_id = utils.str_to_hash(value.value) if vector_id is None else vector_id

        doc = {
            "_id": vector_id,
            "Description": value.value,
            "artifact": str(artifact_json),
        }
    elif isinstance(value, ImageArtifact):
        raise NotImplementedError("`MarqoVectorStoreDriver` does not upserting Image␣
↪Artifacts.")
    else:
        doc = {"_id": vector_id, "Description": value}

    # Non-tensor fields
    if meta:
        doc["meta"] = str(meta)
    if namespace:
        doc["namespace"] = namespace

    response = self.client.index(self.index).add_documents([doc], tensor_fields=[
↪"Description"])
    if isinstance(response, dict) and "items" in response and response["items"]:
        return response["items"][0]["_id"]
    raise ValueError(f"Failed to upsert text: {response}")
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Upsert a vector into the Marqo index.

Parameters

Name	Type	Description	Default
vector	<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	The vector to be indexed. 	*required*
vector_id	*Optional[str]*	The ID for the vector. If None, Marqo will generate an ID. 	*None*
namespace	*Optional[str]*	An optional namespace for the vector. 	*None*
meta	*Optional[dict]*	An optional dictionary of metadata for the vector. 	*None*
kwargs	*Any*	Additional keyword arguments to pass to the Marqo client. 	*{}*

Raises

Returns

Type	Description
str	The ID of the vector that was added.

Source Code in griptape/drivers/vector/marqo_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs: Any,
) -> str:
    """Upsert a vector into the Marqo index.

    Args:
        vector: The vector to be indexed.
        vector_id: The ID for the vector. If None, Marqo will generate an ID.
        namespace: An optional namespace for the vector.
        meta: An optional dictionary of metadata for the vector.
        kwargs: Additional keyword arguments to pass to the Marqo client.
```

(continues on next page)

(continued from previous page)

```
    Raises:
        Exception: This function is not yet implemented.

    Returns:
        The ID of the vector that was added.
    """
    raise NotImplementedError(f"{self.__class__.__name__} does not support upserting␣
↪a vector.")
```

mongodb_atlas

- `__all__ = ['MongoDbAtlasVectorStoreDriver']` module-attribute

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="mongodb_atlas.mdx#griptape.drivers.vector.mongodb_atlas.MongoDbAtlasVectorStoreDriver.connection_string" title="connection_string = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.mongodb_atlas.MongoDbAtlasVectorStoreDriver.connection_string)"><code class="language-python">	*str*	The connection string for the MongoDb Atlas cluster.

continues on next page

Table 272 – continued from previous page

Name	Type	Description
<a href="mongodb_atlas.mdx#griptape.drivers.vector.mongodb_atlas.MongoDbAtlasVectorStoreDriver.database_nan title="database_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.mongodb_atlas.MongoDbAtlasVectorStoreDriver.database_nan class="language-python">	*str*	The name of the database to use.
<a href="mongodb_atlas.mdx#griptape.drivers.vector.mongodb_atlas.MongoDbAtlasVectorStoreDriver.collection_name" title="collection_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.mongodb_atlas.MongoDbAtlasVectorStoreDriver.collection_name)"><code class="language-python">	*str*	The name of the collection to use.

continues on next page

Table 272 – continued from previous page

Name	Type	Description
<a href="mongodb_atlas.mdx#griptape.drivers.vector.mongodb_atlas.MongoDbAtlasVectorStoreDriver.index_name" title="index_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.mongodb_atlas.MongoDbAtlasVectorStoreDriver.index_name)"><code class="language-python">	*str*	The name of the index to use.
<a href="mongodb_atlas.mdx#griptape.drivers.vector.mongodb_atlas.MongoDbAtlasVectorStoreDriver.vector_path" title="vector_path = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.mongodb_atlas.MongoDbAtlasVectorStoreDriver.vector_path)"><code class="language-python">	*str*	The path to the vector field in the collection.

continues on next page

Table 272 – continued from previous page

Name	Type	Description
<a href="mongodb_atlas.mdx#griptape.drivers.vector.mongodb_atlas.MongoDbAtlasVectorStoreDriver.client" title="client() (griptape.drivers.vector.mongodb_atlas.MongoDbAtlasVectorStoreDriver.client)"><coc class="language-python">	*MongoClient*	An optional MongoDb client to use. Defaults to a new client using the connection string.

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
@define
class MongoDbAtlasVectorStoreDriver(BaseVectorStoreDriver):
    """A Vector Store Driver for MongoDb Atlas.

    Attributes:
        connection_string: The connection string for the MongoDb Atlas cluster.
        database_name: The name of the database to use.
        collection_name: The name of the collection to use.
        index_name: The name of the index to use.
        vector_path: The path to the vector field in the collection.
        client: An optional MongoDb client to use. Defaults to a new client using the␣
↪connection string.
    """

    MAX_NUM_CANDIDATES = 10000

    connection_string: str = field(kw_only=True, metadata={"serializable": True})
    database_name: str = field(kw_only=True, metadata={"serializable": True})
    collection_name: str = field(kw_only=True, metadata={"serializable": True})
    index_name: str = field(kw_only=True, metadata={"serializable": True})
    vector_path: str = field(kw_only=True, metadata={"serializable": True})
    num_candidates_multiplier: int = field(
        default=10,
        kw_only=True,
        metadata={"serializable": True},
    )  # https://www.mongodb.com/docs/atlas/atlas-vector-search/vector-search-stage/
↪#fields
    _client: Optional[MongoClient] = field(default=None, kw_only=True, alias="client",
↪ metadata={"serializable": False})

    @lazy_property()
    def client(self) -> MongoClient:
        return import_optional_dependency("pymongo").MongoClient(self.connection_
↪string)
```

(continues on next page)

(continued from previous page)

```
    def get_collection(self) -> Collection:
        """Returns the MongoDB Collection instance for the specified database and␣
↪collection name."""
        return self.client[self.database_name][self.collection_name]

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        """Inserts or updates a vector in the collection.

        If a vector with the given vector ID already exists, it is updated; otherwise,
↪ a new vector is inserted.
        """
        collection = self.get_collection()

        if vector_id is None:
            result = collection.insert_one({self.vector_path: vector, "namespace":␣
↪namespace, "meta": meta})
            vector_id = str(result.inserted_id)
        else:
            collection.replace_one(
                {"_id": vector_id},
                {self.vector_path: vector, "namespace": namespace, "meta": meta},
                upsert=True,
            )
        return vector_id

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        """Loads a document entry from the MongoDB collection based on the vector ID.

        Returns:
            The loaded Entry if found; otherwise, None is returned.
        """
        collection = self.get_collection()
        if namespace:
            doc = collection.find_one({"_id": vector_id, "namespace": namespace})
        else:
            doc = collection.find_one({"_id": vector_id})

        if doc is None:
            return doc
        return BaseVectorStoreDriver.Entry(
            id=str(doc["_id"]),
            vector=doc[self.vector_path],
```

(continues on next page)

(continued from previous page)

```
            namespace=doc["namespace"],
            meta=doc["meta"],
        )

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Loads all document entries from the MongoDB collection.

        Entries can optionally be filtered by namespace.
        """
        collection = self.get_collection()
        cursor = collection.find() if namespace is None else collection.find({
↪"namespace": namespace})

        return [
            BaseVectorStoreDriver.Entry(
                id=str(doc["_id"]),
                vector=doc[self.vector_path],
                namespace=doc["namespace"],
                meta=doc["meta"],
            )
            for doc in cursor
        ]

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        offset: Optional[int] = None,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Queries the MongoDB collection for documents that match the provided␣
↪vector list.

        Results can be customized based on parameters like count, namespace,␣
↪inclusion of vectors, offset, and index.
        """
        collection = self.get_collection()

        count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
        offset = offset or 0

        pipeline = [
            {
                "$vectorSearch": {
                    "index": self.index_name,
                    "path": self.vector_path,
                    "queryVector": vector,
                    "numCandidates": min(count * self.num_candidates_multiplier, self.
```

(continues on next page)

(continued from previous page)

```
↪MAX_NUM_CANDIDATES),
                    "limit": count,
                },
            },
            {
                "$project": {
                    "_id": 1,
                    self.vector_path: 1,
                    "namespace": 1,
                    "meta": 1,
                    "score": {"$meta": "vectorSearchScore"},
                },
            },
        ]

        if namespace:
            pipeline[0]["$vectorSearch"]["filter"] = {"namespace": namespace}

        return [
            BaseVectorStoreDriver.Entry(
                id=str(doc["_id"]),
                vector=doc[self.vector_path] if include_vectors else [],
                score=doc["score"],
                meta=doc["meta"],
                namespace=namespace,
            )
            for doc in collection.aggregate(pipeline)
        ]

    def delete_vector(self, vector_id: str) -> None:
        """Deletes the vector from the collection."""
        collection = self.get_collection()
        collection.delete_one({"_id": vector_id})
```

- `MAX_NUM_CANDIDATES = 10000` class-attribute instance-attribute
- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `collection_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `connection_string = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `database_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `index_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `num_candidates_multiplier = field(default=10, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `vector_path = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
@lazy_property()
def client(self) -> MongoClient:
    return import_optional_dependency("pymongo").MongoClient(self.connection_string)
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> None:
    """Deletes the vector from the collection."""
    collection = self.get_collection()
    collection.delete_one({"_id": vector_id})
```

get_collection()

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
def get_collection(self) -> Collection:
    """Returns the MongoDB Collection instance for the specified database and␣
↪collection name."""
    return self.client[self.database_name][self.collection_name]
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Loads all document entries from the MongoDB collection.

    Entries can optionally be filtered by namespace.
    """
    collection = self.get_collection()
    cursor = collection.find() if namespace is None else collection.find({"namespace
↪": namespace})

    return [
        BaseVectorStoreDriver.Entry(
            id=str(doc["_id"]),
            vector=doc[self.vector_path],
            namespace=doc["namespace"],
            meta=doc["meta"],
        )
        for doc in cursor
    ]
```

load_entry(vector_id, *, namespace=None)

Loads a document entry from the MongoDB collection based on the vector ID.

Returns

Type	Description
<a href="./#griptape.drivers.vector.BaseVectorStoreDriver.Entry" title="Entry (griptape.drivers.vector.BaseVectorStoreDriver.Entry)"><code class="language-python">C	The loaded Entry if found; otherwise, None is returned.

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    """Loads a document entry from the MongoDB collection based on the vector ID.

    Returns:
        The loaded Entry if found; otherwise, None is returned.
    """
    collection = self.get_collection()
    if namespace:
        doc = collection.find_one({"_id": vector_id, "namespace": namespace})
    else:
        doc = collection.find_one({"_id": vector_id})

    if doc is None:
        return doc
    return BaseVectorStoreDriver.Entry(
        id=str(doc["_id"]),
        vector=doc[self.vector_path],
        namespace=doc["namespace"],
        meta=doc["meta"],
    )
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, offset=None, **kwargs)

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
```

(continues on next page)

(continued from previous page)

```
    include_vectors: bool = False,
    offset: Optional[int] = None,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    """Queries the MongoDB collection for documents that match the provided vector␣
↪list.

    Results can be customized based on parameters like count, namespace, inclusion of␣
↪vectors, offset, and index.
    """
    collection = self.get_collection()

    count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
    offset = offset or 0

    pipeline = [
        {
            "$vectorSearch": {
                "index": self.index_name,
                "path": self.vector_path,
                "queryVector": vector,
                "numCandidates": min(count * self.num_candidates_multiplier, self.MAX_
↪NUM_CANDIDATES),
                "limit": count,
            },
        },
        {
            "$project": {
                "_id": 1,
                self.vector_path: 1,
                "namespace": 1,
                "meta": 1,
                "score": {"$meta": "vectorSearchScore"},
            },
        },
    ]

    if namespace:
        pipeline[0]["$vectorSearch"]["filter"] = {"namespace": namespace}

    return [
        BaseVectorStoreDriver.Entry(
            id=str(doc["_id"]),
            vector=doc[self.vector_path] if include_vectors else [],
            score=doc["score"],
            meta=doc["meta"],
            namespace=namespace,
        )
        for doc in collection.aggregate(pipeline)
    ]
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    """Inserts or updates a vector in the collection.

    If a vector with the given vector ID already exists, it is updated; otherwise, a new vector is inserted.
    """
    collection = self.get_collection()

    if vector_id is None:
        result = collection.insert_one({self.vector_path: vector, "namespace": namespace, "meta": meta})
        vector_id = str(result.inserted_id)
    else:
        collection.replace_one(
            {"_id": vector_id},
            {self.vector_path: vector, "namespace": namespace, "meta": meta},
            upsert=True,
        )
    return vector_id
```

mongodb_atlas_vector_store_driver

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="mongodb_atlas_vector_store_driver.mdx#griptape.drivers.vector.mongodb_atlas_vector_store_driver.MongoDbAtlasVectorStoreDriver.connection_string" title="connection_string = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.mongodb_atlas_vector_store_driver.MongoDbAtlasVectorStoreDriver.connection_string)"><code class="language-python">	*str*	The connection string for the MongoDb Atlas cluster.
<a href="mongodb_atlas_vector_store_driver.mdx#griptape.drivers.vector.mongodb_atlas_vector_store_driver.MongoDbAtlasVectorStoreDriver.database_nam title="database_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.mongodb_atlas_vector_store_driver.MongoDbAtlasVectorStoreDriver.database_nam class="language-python">	*str*	The name of the database to use.

continues on next page

Table 274 – continued from previous page

Name	Type	Description
<a href="mongodb_atlas_vector_store_driver.mdx#griptape.drivers.vector.mongodb_atlas_vector_store_driver.MongoDbAtlasVectorStoreDriver.collection_name" title="collection_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.mongodb_atlas_vector_store_driver.MongoDbAtlasVectorStoreDriver.collection_name)"><code class="language-python">	*str*	The name of the collection to use.
<a href="mongodb_atlas_vector_store_driver.mdx#griptape.drivers.vector.mongodb_atlas_vector_store_driver.MongoDbAtlasVectorStoreDriver.index_name" title="index_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.mongodb_atlas_vector_store_driver.MongoDbAtlasVectorStoreDriver.index_name)"><code class="language-python">	*str*	The name of the index to use.

continues on next page

Table 274 – continued from previous page

Name	Type	Description
<a href="mongodb_atlas_vector_store_driver.mdx#griptape.drivers.vector.mongodb_atlas_vector_store_driver.MongoDbAtlasVectorStoreDriver.vector_path" title="vector_path = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.mongodb_atlas_vector_store_driver.MongoDbAtlasVectorStoreDriver.vector_path)"><code class="language-python">	*str*	The path to the vector field in the collection.
<a href="mongodb_atlas_vector_store_driver.mdx#griptape.drivers.vector.mongodb_atlas_vector_store_driver.MongoDbAtlasVectorStoreDriver.client" title="client() (griptape.drivers.vector.mongodb_atlas_vector_store_driver.MongoDbAtlasVectorStoreDriver.client)"><coc class="language-python">	*MongoClient*	An optional MongoDb client to use. Defaults to a new client using the connection string.

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
@define
class MongoDbAtlasVectorStoreDriver(BaseVectorStoreDriver):
    """A Vector Store Driver for MongoDb Atlas.

    Attributes:
        connection_string: The connection string for the MongoDb Atlas cluster.
        database_name: The name of the database to use.
        collection_name: The name of the collection to use.
        index_name: The name of the index to use.
        vector_path: The path to the vector field in the collection.
```

(continues on next page)

(continued from previous page)

```
        client: An optional MongoDb client to use. Defaults to a new client using the␣
↪connection string.
    """

    MAX_NUM_CANDIDATES = 10000

    connection_string: str = field(kw_only=True, metadata={"serializable": True})
    database_name: str = field(kw_only=True, metadata={"serializable": True})
    collection_name: str = field(kw_only=True, metadata={"serializable": True})
    index_name: str = field(kw_only=True, metadata={"serializable": True})
    vector_path: str = field(kw_only=True, metadata={"serializable": True})
    num_candidates_multiplier: int = field(
        default=10,
        kw_only=True,
        metadata={"serializable": True},
    )  # https://www.mongodb.com/docs/atlas/atlas-vector-search/vector-search-stage/
↪#fields
    _client: Optional[MongoClient] = field(default=None, kw_only=True, alias="client",
↪ metadata={"serializable": False})

    @lazy_property()
    def client(self) -> MongoClient:
        return import_optional_dependency("pymongo").MongoClient(self.connection_
↪string)

    def get_collection(self) -> Collection:
        """Returns the MongoDB Collection instance for the specified database and␣
↪collection name."""
        return self.client[self.database_name][self.collection_name]

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        """Inserts or updates a vector in the collection.

        If a vector with the given vector ID already exists, it is updated; otherwise,
↪ a new vector is inserted.
        """
        collection = self.get_collection()

        if vector_id is None:
            result = collection.insert_one({self.vector_path: vector, "namespace":␣
↪namespace, "meta": meta})
            vector_id = str(result.inserted_id)
        else:
            collection.replace_one(
```

(continues on next page)

(continued from previous page)

```
                {"_id": vector_id},
                {self.vector_path: vector, "namespace": namespace, "meta": meta},
                upsert=True,
            )
        return vector_id

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        """Loads a document entry from the MongoDB collection based on the vector ID.

        Returns:
            The loaded Entry if found; otherwise, None is returned.
        """
        collection = self.get_collection()
        if namespace:
            doc = collection.find_one({"_id": vector_id, "namespace": namespace})
        else:
            doc = collection.find_one({"_id": vector_id})

        if doc is None:
            return doc
        return BaseVectorStoreDriver.Entry(
            id=str(doc["_id"]),
            vector=doc[self.vector_path],
            namespace=doc["namespace"],
            meta=doc["meta"],
        )

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Loads all document entries from the MongoDB collection.

        Entries can optionally be filtered by namespace.
        """
        collection = self.get_collection()
        cursor = collection.find() if namespace is None else collection.find({
↪"namespace": namespace})

        return [
            BaseVectorStoreDriver.Entry(
                id=str(doc["_id"]),
                vector=doc[self.vector_path],
                namespace=doc["namespace"],
                meta=doc["meta"],
            )
            for doc in cursor
        ]

    def query_vector(
        self,
        vector: list[float],
        *,
```

(continues on next page)

(continued from previous page)

```
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        offset: Optional[int] = None,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Queries the MongoDB collection for documents that match the provided␣
↪vector list.

        Results can be customized based on parameters like count, namespace,␣
↪inclusion of vectors, offset, and index.
        """
        collection = self.get_collection()

        count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
        offset = offset or 0

        pipeline = [
            {
                "$vectorSearch": {
                    "index": self.index_name,
                    "path": self.vector_path,
                    "queryVector": vector,
                    "numCandidates": min(count * self.num_candidates_multiplier, self.
↪MAX_NUM_CANDIDATES),
                    "limit": count,
                },
            },
            {
                "$project": {
                    "_id": 1,
                    self.vector_path: 1,
                    "namespace": 1,
                    "meta": 1,
                    "score": {"$meta": "vectorSearchScore"},
                },
            },
        ]

        if namespace:
            pipeline[0]["$vectorSearch"]["filter"] = {"namespace": namespace}

        return [
            BaseVectorStoreDriver.Entry(
                id=str(doc["_id"]),
                vector=doc[self.vector_path] if include_vectors else [],
                score=doc["score"],
                meta=doc["meta"],
                namespace=namespace,
            )
            for doc in collection.aggregate(pipeline)
        ]
```

(continues on next page)

(continued from previous page)

```
    def delete_vector(self, vector_id: str) -> None:
        """Deletes the vector from the collection."""
        collection = self.get_collection()
        collection.delete_one({"_id": vector_id})
```

- `MAX_NUM_CANDIDATES = 10000` class-attribute instance-attribute
- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `collection_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `connection_string = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `database_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `index_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `num_candidates_multiplier = field(default=10, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `vector_path = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
@lazy_property()
def client(self) -> MongoClient:
    return import_optional_dependency("pymongo").MongoClient(self.connection_string)
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> None:
    """Deletes the vector from the collection."""
    collection = self.get_collection()
    collection.delete_one({"_id": vector_id})
```

get_collection()

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
def get_collection(self) -> Collection:
    """Returns the MongoDB Collection instance for the specified database and␣
↪collection name."""
    return self.client[self.database_name][self.collection_name]
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Loads all document entries from the MongoDB collection.

    Entries can optionally be filtered by namespace.
    """
    collection = self.get_collection()
    cursor = collection.find() if namespace is None else collection.find({"namespace
↪": namespace})

    return [
        BaseVectorStoreDriver.Entry(
            id=str(doc["_id"]),
            vector=doc[self.vector_path],
            namespace=doc["namespace"],
            meta=doc["meta"],
        )
        for doc in cursor
    ]
```

load_entry(vector_id, *, namespace=None)

Loads a document entry from the MongoDB collection based on the vector ID.

Returns

Type	Description
<a href="./#griptape.drivers.vector.BaseVectorStoreDriver.Entry" title="Entry (griptape.drivers.vector.BaseVectorStoreDriver.Entry)"><code class="language-python">C	The loaded Entry if found; otherwise, None is returned.

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    """Loads a document entry from the MongoDB collection based on the vector ID.

    Returns:
        The loaded Entry if found; otherwise, None is returned.
    """
    collection = self.get_collection()
    if namespace:
```

(continues on next page)

(continued from previous page)

```
        doc = collection.find_one({"_id": vector_id, "namespace": namespace})
    else:
        doc = collection.find_one({"_id": vector_id})

    if doc is None:
        return doc
    return BaseVectorStoreDriver.Entry(
        id=str(doc["_id"]),
        vector=doc[self.vector_path],
        namespace=doc["namespace"],
        meta=doc["meta"],
    )
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, offset=None, **kwargs)

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    offset: Optional[int] = None,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    """Queries the MongoDB collection for documents that match the provided vector␣
↪list.

    Results can be customized based on parameters like count, namespace, inclusion of␣
↪vectors, offset, and index.
    """
    collection = self.get_collection()

    count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
    offset = offset or 0

    pipeline = [
        {
            "$vectorSearch": {
                "index": self.index_name,
                "path": self.vector_path,
                "queryVector": vector,
                "numCandidates": min(count * self.num_candidates_multiplier, self.MAX_
↪NUM_CANDIDATES),
                "limit": count,
            },
        },
        {
            "$project": {
```

(continues on next page)

(continued from previous page)

```
                "_id": 1,
                self.vector_path: 1,
                "namespace": 1,
                "meta": 1,
                "score": {"$meta": "vectorSearchScore"},
            },
        },
    ]

    if namespace:
        pipeline[0]["$vectorSearch"]["filter"] = {"namespace": namespace}

    return [
        BaseVectorStoreDriver.Entry(
            id=str(doc["_id"]),
            vector=doc[self.vector_path] if include_vectors else [],
            score=doc["score"],
            meta=doc["meta"],
            namespace=namespace,
        )
        for doc in collection.aggregate(pipeline)
    ]
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/mongodb_atlas_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    """Inserts or updates a vector in the collection.

    If a vector with the given vector ID already exists, it is updated; otherwise, a␣
↪new vector is inserted.
    """
    collection = self.get_collection()

    if vector_id is None:
        result = collection.insert_one({self.vector_path: vector, "namespace":␣
↪namespace, "meta": meta})
        vector_id = str(result.inserted_id)
    else:
        collection.replace_one(
            {"_id": vector_id},
            {self.vector_path: vector, "namespace": namespace, "meta": meta},
            upsert=True,
```

(continues on next page)

(continued from previous page)

```
        )
    return vector_id
```

opensearch

- `__all__ = ['OpenSearchVectorStoreDriver']` module-attribute

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="opensearch.mdx#griptape.drivers.vector.opensearch.OpenSearchVectorStoreDriver.host" title="host = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.opensearch.OpenSearchVectorStoreDriver.host)"><code class="language-python">	*str*	The host of the OpenSearch cluster.
<a href="opensearch.mdx#griptape.drivers.vector.opensearch.OpenSearchVectorStoreDriver.port" title="port = field(default=443, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.opensearch.OpenSearchVectorStoreDriver.port)"><code class="language-python">	*int*	The port of the OpenSearch cluster.

continues on next page

Table 276 – continued from previous page

Name	Type	Description
<a href="opensearch.mdx#griptape.drivers.vector.opensearch.OpenSearchVectorStoreDriver.http_auth" title="http_auth = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.opensearch.OpenSearchVectorStoreDriver.http_auth)"><code class="language-python">	*str I tuple[str, Optional[str]]*	The HTTP authentication credentials to use.
<a href="opensearch.mdx#griptape.drivers.vector.opensearch.OpenSearchVectorStoreDriver.use_ssl" title="use_ssl = field(default=True, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.opensearch.OpenSearchVectorStoreDriver.use_ssl)"><co class="language-python">	*bool*	Whether to use SSL.
<a href="opensearch.mdx#griptape.drivers.vector.opensearch.OpenSearchVectorStoreDriver.verify_certs" title="verify_certs = field(default=True, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.opensearch.OpenSearchVectorStoreDriver.verify_certs)"><code class="language-python">	*bool*	Whether to verify SSL certificates.

continues on next page

Table 276 – continued from previous page

Name	Type	Description
class-attribute instance-attribute (griptape.drivers.vector.opensearch.OpenSearchVectorStoreDriver.index_name)	*str*	The name of the index to use.

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
@define
class OpenSearchVectorStoreDriver(BaseVectorStoreDriver):
    """A Vector Store Driver for OpenSearch.

    Attributes:
        host: The host of the OpenSearch cluster.
        port: The port of the OpenSearch cluster.
        http_auth: The HTTP authentication credentials to use.
        use_ssl: Whether to use SSL.
        verify_certs: Whether to verify SSL certificates.
        index_name: The name of the index to use.
    """

    host: str = field(kw_only=True, metadata={"serializable": True})
    port: int = field(default=443, kw_only=True, metadata={"serializable": True})
    http_auth: str | tuple[str, Optional[str]] = field(default=None, kw_only=True,␣
↪metadata={"serializable": True})
    use_ssl: bool = field(default=True, kw_only=True, metadata={"serializable": True})
    verify_certs: bool = field(default=True, kw_only=True, metadata={"serializable":␣
↪True})
    index_name: str = field(kw_only=True, metadata={"serializable": True})
    _client: Optional[OpenSearch] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> OpenSearch:
        opensearchpy = import_optional_dependency("opensearchpy")

        return opensearchpy.OpenSearch(
            hosts=[{"host": self.host, "port": self.port}],
```

(continues on next page)

(continued from previous page)

```
            http_auth=self.http_auth,
            use_ssl=self.use_ssl,
            verify_certs=self.verify_certs,
            connection_class=opensearchpy.RequestsHttpConnection,
        )

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        """Inserts or updates a vector in OpenSearch.

        If a vector with the given vector ID already exists, it is updated; otherwise,
↪ a new vector is inserted.
        Metadata associated with the vector can also be provided.
        """
        vector_id = vector_id or utils.str_to_hash(str(vector))
        doc = {"vector": vector, "namespace": namespace, "metadata": meta}
        doc.update(kwargs)
        response = self.client.index(index=self.index_name, id=vector_id, body=doc)

        return response["_id"]

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        """Retrieves a specific vector entry from OpenSearch based on its identifier␣
↪and optional namespace.

        Returns:
            If the entry is found, it returns an instance of BaseVectorStoreDriver.
↪Entry; otherwise, None is returned.
        """
        try:
            query = {"bool": {"must": [{"term": {"_id": vector_id}}]}}

            if namespace:
                query["bool"]["must"].append({"term": {"namespace": namespace}})

            response = self.client.search(index=self.index_name, body={"query": query,
↪ "size": 1})

            if response["hits"]["total"]["value"] > 0:
                vector_data = response["hits"]["hits"][0]["_source"]
                return BaseVectorStoreDriver.Entry(
                    id=vector_id,
                    meta=vector_data.get("metadata"),
                    vector=vector_data.get("vector"),
```

(continues on next page)

(continued from previous page)

```
                    namespace=vector_data.get("namespace"),
                )
            return None
        except Exception as e:
            logging.exception("Error while loading entry: %s", e)
            return None

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Retrieves all vector entries from OpenSearch that match the optional␣
↪namespace.

        Returns:
            A list of BaseVectorStoreDriver.Entry objects.
        """
        query_body = {"size": 10000, "query": {"match_all": {}}}

        if namespace:
            query_body["query"] = {"match": {"namespace": namespace}}

        response = self.client.search(index=self.index_name, body=query_body)

        return [
            BaseVectorStoreDriver.Entry(
                id=hit["_id"],
                vector=hit["_source"].get("vector"),
                meta=hit["_source"].get("metadata"),
                namespace=hit["_source"].get("namespace"),
            )
            for hit in response["hits"]["hits"]
        ]

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        include_metadata: bool = True,
        field_name: str = "vector",
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Performs a nearest neighbor search on OpenSearch to find vectors similar␣
↪to the provided vector list.

        Results can be limited using the count parameter and optionally filtered by a␣
↪namespace.

        Returns:
            A list of BaseVectorStoreDriver.Entry objects, each encapsulating the␣
↪retrieved vector, its similarity score, metadata, and namespace.
```

(continues on next page)

(continued from previous page)

```
        """
        count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
        # Base k-NN query
        query_body = {"size": count, "query": {"knn": {field_name: {"vector": vector,
↪"k": count}}}}

        if namespace:
            query_body["query"] = {
                "bool": {
                    "must": [
                        {"match": {"namespace": namespace}},
                        {"knn": {field_name: {"vector": vector, "k": count}}},
                    ],
                },
            }

        response = self.client.search(index=self.index_name, body=query_body)

        return [
            BaseVectorStoreDriver.Entry(
                id=hit["_id"],
                namespace=hit["_source"].get("namespace") if namespace else None,
                score=hit["_score"],
                vector=hit["_source"].get("vector") if include_vectors else None,
                meta=hit["_source"].get("metadata") if include_metadata else None,
            )
            for hit in response["hits"]["hits"]
        ]

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `host = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `http_auth = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `index_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `port = field(default=443, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `use_ssl = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `verify_certs = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
@lazy_property()
def client(self) -> OpenSearch:
    opensearchpy = import_optional_dependency("opensearchpy")

    return opensearchpy.OpenSearch(
        hosts=[{"host": self.host, "port": self.port}],
        http_auth=self.http_auth,
        use_ssl=self.use_ssl,
        verify_certs=self.verify_certs,
        connection_class=opensearchpy.RequestsHttpConnection,
    )
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

load_entries(*, namespace=None)

Retrieves all vector entries from OpenSearch that match the optional namespace.

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	A list of BaseVectorStoreDriver.Entry objects.

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Retrieves all vector entries from OpenSearch that match the optional namespace.

    Returns:
        A list of BaseVectorStoreDriver.Entry objects.
    """
    query_body = {"size": 10000, "query": {"match_all": {}}}

    if namespace:
        query_body["query"] = {"match": {"namespace": namespace}}

    response = self.client.search(index=self.index_name, body=query_body)
```

(continues on next page)

(continued from previous page)

```
    return [
        BaseVectorStoreDriver.Entry(
            id=hit["_id"],
            vector=hit["_source"].get("vector"),
            meta=hit["_source"].get("metadata"),
            namespace=hit["_source"].get("namespace"),
        )
        for hit in response["hits"]["hits"]
    ]
```

load_entry(vector_id, *, namespace=None)

Retrieves a specific vector entry from OpenSearch based on its identifier and optional namespace.

Returns

Type	Description
<a href="./#griptape.drivers.vector.BaseVectorStoreDriver.Entry" title="Entry (griptape.drivers.vector.BaseVectorStoreDriver.Entry)"><code class="language-python">C	If the entry is found, it returns an instance of BaseVectorStoreDriver.Entry; otherwise, None is returned.

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    """Retrieves a specific vector entry from OpenSearch based on its identifier and␣
↪optional namespace.

    Returns:
        If the entry is found, it returns an instance of BaseVectorStoreDriver.Entry;␣
↪otherwise, None is returned.
    """
    try:
        query = {"bool": {"must": [{"term": {"_id": vector_id}}]}}

        if namespace:
            query["bool"]["must"].append({"term": {"namespace": namespace}})

        response = self.client.search(index=self.index_name, body={"query": query,
↪"size": 1})

        if response["hits"]["total"]["value"] > 0:
            vector_data = response["hits"]["hits"][0]["_source"]
            return BaseVectorStoreDriver.Entry(
                id=vector_id,
```

(continues on next page)

(continued from previous page)

```
                meta=vector_data.get("metadata"),
                vector=vector_data.get("vector"),
                namespace=vector_data.get("namespace"),
            )
        return None
    except Exception as e:
        logging.exception("Error while loading entry: %s", e)
        return None
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, include_metadata=True, field_name='vector', **kwargs)

Performs a nearest neighbor search on OpenSearch to find vectors similar to the provided vector list.

Results can be limited using the count parameter and optionally filtered by a namespace.

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	A list of BaseVectorStoreDriver.Entry objects, each encapsulating the retrieved vector, its similarity score, metadata, and namespace.

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    include_metadata: bool = True,
    field_name: str = "vector",
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    """Performs a nearest neighbor search on OpenSearch to find vectors similar to␣
↪the provided vector list.

    Results can be limited using the count parameter and optionally filtered by a␣
↪namespace.

    Returns:
        A list of BaseVectorStoreDriver.Entry objects, each encapsulating the␣
↪retrieved vector, its similarity score, metadata, and namespace.
    """
    count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
    # Base k-NN query
    query_body = {"size": count, "query": {"knn": {field_name: {"vector": vector, "k
```

(continues on next page)

(continued from previous page)

```
↪": count}}}}

    if namespace:
        query_body["query"] = {
            "bool": {
                "must": [
                    {"match": {"namespace": namespace}},
                    {"knn": {field_name: {"vector": vector, "k": count}}},
                ],
            },
        }

    response = self.client.search(index=self.index_name, body=query_body)

    return [
        BaseVectorStoreDriver.Entry(
            id=hit["_id"],
            namespace=hit["_source"].get("namespace") if namespace else None,
            score=hit["_score"],
            vector=hit["_source"].get("vector") if include_vectors else None,
            meta=hit["_source"].get("metadata") if include_metadata else None,
        )
        for hit in response["hits"]["hits"]
    ]
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    """Inserts or updates a vector in OpenSearch.

    If a vector with the given vector ID already exists, it is updated; otherwise, a␣
↪new vector is inserted.
    Metadata associated with the vector can also be provided.
    """
    vector_id = vector_id or utils.str_to_hash(str(vector))
    doc = {"vector": vector, "namespace": namespace, "metadata": meta}
    doc.update(kwargs)
    response = self.client.index(index=self.index_name, id=vector_id, body=doc)

    return response["_id"]
```

opensearch_vector_store_driver

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="opensearch_vector_store_driver.mdx#griptape.drivers.vector.opensearch_vector_store_driver.OpenSearchVectorStoreDriver.host" title="host = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.opensearch_vector_store_driver.OpenSearchVectorStoreDriver.host)"><code class="language-python">	*str*	The host of the OpenSearch cluster.
<a href="opensearch_vector_store_driver.mdx#griptape.drivers.vector.opensearch_vector_store_driver.OpenSearchVectorStoreDriver.port" title="port = field(default=443, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.opensearch_vector_store_driver.OpenSearchVectorStoreDriver.port)"><code class="language-python">	*int*	The port of the OpenSearch cluster.

continues on next page

Table 280 – continued from previous page

Name	Type	Description
<a href="opensearch_vector_store_driver.mdx#griptape.drivers.vector.opensearch_vector_store_driver.OpenSearchVectorStoreDriver.http_auth" title="http_auth = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.opensearch_vector_store_driver.OpenSearchVectorStoreDriver.http_auth)"><c class="language-python">	*str I tuple[str, Optional[str]]*	The HTTP authentication credentials to use.
<a href="opensearch_vector_store_driver.mdx#griptape.drivers.vector.opensearch_vector_store_driver.OpenSearchVectorStoreDriver.use_ssl" title="use_ssl = field(default=True, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.opensearch_vector_store_driver.OpenSearchVectorStoreDriver.use_ssl)"><c class="language-python">	*bool*	Whether to use SSL.

continues on next page

Table 280 – continued from previous page

Name	Type	Description
<a href="opensearch_vector_store_driver.mdx#griptape.drivers.vector.opensearch_vector_store_driver.OpenSearchVectorStoreDriver.verify_certs" title="verify_certs = field(default=True, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.opensearch_vector_store_driver.OpenSearchVectorStoreDriver.verify_certs)"><code class="language-python">	*bool*	Whether to verify SSL certificates.
<a href="opensearch_vector_store_driver.mdx#griptape.drivers.vector.opensearch_vector_store_driver.OpenSearchVectorStoreDriver.index_name" title="index_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.opensearch_vector_store_driver.OpenSearchVectorStoreDriver.index_name)"><code class="language-python">	*str*	The name of the index to use.

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
@define
class OpenSearchVectorStoreDriver(BaseVectorStoreDriver):
    """A Vector Store Driver for OpenSearch.

    Attributes:
        host: The host of the OpenSearch cluster.
        port: The port of the OpenSearch cluster.
```

(continues on next page)

(continued from previous page)

```
        http_auth: The HTTP authentication credentials to use.
        use_ssl: Whether to use SSL.
        verify_certs: Whether to verify SSL certificates.
        index_name: The name of the index to use.
    """

    host: str = field(kw_only=True, metadata={"serializable": True})
    port: int = field(default=443, kw_only=True, metadata={"serializable": True})
    http_auth: str | tuple[str, Optional[str]] = field(default=None, kw_only=True,␣
↪metadata={"serializable": True})
    use_ssl: bool = field(default=True, kw_only=True, metadata={"serializable": True})
    verify_certs: bool = field(default=True, kw_only=True, metadata={"serializable":␣
↪True})
    index_name: str = field(kw_only=True, metadata={"serializable": True})
    _client: Optional[OpenSearch] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> OpenSearch:
        opensearchpy = import_optional_dependency("opensearchpy")

        return opensearchpy.OpenSearch(
            hosts=[{"host": self.host, "port": self.port}],
            http_auth=self.http_auth,
            use_ssl=self.use_ssl,
            verify_certs=self.verify_certs,
            connection_class=opensearchpy.RequestsHttpConnection,
        )

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        """Inserts or updates a vector in OpenSearch.

        If a vector with the given vector ID already exists, it is updated; otherwise,
↪ a new vector is inserted.
        Metadata associated with the vector can also be provided.
        """
        vector_id = vector_id or utils.str_to_hash(str(vector))
        doc = {"vector": vector, "namespace": namespace, "metadata": meta}
        doc.update(kwargs)
        response = self.client.index(index=self.index_name, id=vector_id, body=doc)

        return response["_id"]

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
```

(continues on next page)

(continued from previous page)

```
↪Optional[BaseVectorStoreDriver.Entry]:
        """Retrieves a specific vector entry from OpenSearch based on its identifier␣
↪and optional namespace.

        Returns:
            If the entry is found, it returns an instance of BaseVectorStoreDriver.
↪Entry; otherwise, None is returned.
        """
        try:
            query = {"bool": {"must": [{"term": {"_id": vector_id}}]}}

            if namespace:
                query["bool"]["must"].append({"term": {"namespace": namespace}})

            response = self.client.search(index=self.index_name, body={"query": query,
↪ "size": 1})

            if response["hits"]["total"]["value"] > 0:
                vector_data = response["hits"]["hits"][0]["_source"]
                return BaseVectorStoreDriver.Entry(
                    id=vector_id,
                    meta=vector_data.get("metadata"),
                    vector=vector_data.get("vector"),
                    namespace=vector_data.get("namespace"),
                )
            return None
        except Exception as e:
            logging.exception("Error while loading entry: %s", e)
            return None

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Retrieves all vector entries from OpenSearch that match the optional␣
↪namespace.

        Returns:
            A list of BaseVectorStoreDriver.Entry objects.
        """
        query_body = {"size": 10000, "query": {"match_all": {}}}

        if namespace:
            query_body["query"] = {"match": {"namespace": namespace}}

        response = self.client.search(index=self.index_name, body=query_body)

        return [
            BaseVectorStoreDriver.Entry(
                id=hit["_id"],
                vector=hit["_source"].get("vector"),
                meta=hit["_source"].get("metadata"),
                namespace=hit["_source"].get("namespace"),
            )
```

(continues on next page)

(continued from previous page)

```
            for hit in response["hits"]["hits"]
        ]

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        include_metadata: bool = True,
        field_name: str = "vector",
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Performs a nearest neighbor search on OpenSearch to find vectors similar to the provided vector list.

        Results can be limited using the count parameter and optionally filtered by a namespace.

        Returns:
            A list of BaseVectorStoreDriver.Entry objects, each encapsulating the retrieved vector, its similarity score, metadata, and namespace.
        """
        count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
        # Base k-NN query
        query_body = {"size": count, "query": {"knn": {field_name: {"vector": vector, "k": count}}}}

        if namespace:
            query_body["query"] = {
                "bool": {
                    "must": [
                        {"match": {"namespace": namespace}},
                        {"knn": {field_name: {"vector": vector, "k": count}}},
                    ],
                },
            }

        response = self.client.search(index=self.index_name, body=query_body)

        return [
            BaseVectorStoreDriver.Entry(
                id=hit["_id"],
                namespace=hit["_source"].get("namespace") if namespace else None,
                score=hit["_score"],
                vector=hit["_source"].get("vector") if include_vectors else None,
                meta=hit["_source"].get("metadata") if include_metadata else None,
            )
            for hit in response["hits"]["hits"]
        ]
```

(continues on next page)

(continued from previous page)

```
    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `host = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `http_auth = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `index_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `port = field(default=443, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `use_ssl = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `verify_certs = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
@lazy_property()
def client(self) -> OpenSearch:
    opensearchpy = import_optional_dependency("opensearchpy")

    return opensearchpy.OpenSearch(
        hosts=[{"host": self.host, "port": self.port}],
        http_auth=self.http_auth,
        use_ssl=self.use_ssl,
        verify_certs=self.verify_certs,
        connection_class=opensearchpy.RequestsHttpConnection,
    )
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

load_entries(*, namespace=None)

Retrieves all vector entries from OpenSearch that match the optional namespace.

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	A list of BaseVectorStoreDriver.Entry objects.

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Retrieves all vector entries from OpenSearch that match the optional namespace.

    Returns:
        A list of BaseVectorStoreDriver.Entry objects.
    """
    query_body = {"size": 10000, "query": {"match_all": {}}}

    if namespace:
        query_body["query"] = {"match": {"namespace": namespace}}

    response = self.client.search(index=self.index_name, body=query_body)

    return [
        BaseVectorStoreDriver.Entry(
            id=hit["_id"],
            vector=hit["_source"].get("vector"),
            meta=hit["_source"].get("metadata"),
            namespace=hit["_source"].get("namespace"),
        )
        for hit in response["hits"]["hits"]
    ]
```

load_entry(vector_id, *, namespace=None)

Retrieves a specific vector entry from OpenSearch based on its identifier and optional namespace.

Returns

Type	Description
<a href="./#griptape.drivers.vector.BaseVectorStoreDriver.Entry" title="Entry (griptape.drivers.vector.BaseVectorStoreDriver.Entry)"><code class="language-python">C	If the entry is found, it returns an instance of BaseVectorStoreDriver.Entry; otherwise, None is returned.

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    """Retrieves a specific vector entry from OpenSearch based on its identifier and␣
↪optional namespace.

    Returns:
        If the entry is found, it returns an instance of BaseVectorStoreDriver.Entry;␣
↪otherwise, None is returned.
    """
    try:
        query = {"bool": {"must": [{"term": {"_id": vector_id}}]}}

        if namespace:
            query["bool"]["must"].append({"term": {"namespace": namespace}})

        response = self.client.search(index=self.index_name, body={"query": query,
↪"size": 1})

        if response["hits"]["total"]["value"] > 0:
            vector_data = response["hits"]["hits"][0]["_source"]
            return BaseVectorStoreDriver.Entry(
                id=vector_id,
                meta=vector_data.get("metadata"),
                vector=vector_data.get("vector"),
                namespace=vector_data.get("namespace"),
            )
        return None
    except Exception as e:
        logging.exception("Error while loading entry: %s", e)
        return None
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, include_metadata=True, field_name='vector', **kwargs)

Performs a nearest neighbor search on OpenSearch to find vectors similar to the provided vector list.

Results can be limited using the count parameter and optionally filtered by a namespace.

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	A list of BaseVectorStoreDriver.Entry objects, each encapsulating the retrieved vector, its similarity score, metadata, and namespace.

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    include_metadata: bool = True,
    field_name: str = "vector",
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    """Performs a nearest neighbor search on OpenSearch to find vectors similar to␣
↪the provided vector list.

    Results can be limited using the count parameter and optionally filtered by a␣
↪namespace.

    Returns:
        A list of BaseVectorStoreDriver.Entry objects, each encapsulating the␣
↪retrieved vector, its similarity score, metadata, and namespace.
    """
    count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT
    # Base k-NN query
    query_body = {"size": count, "query": {"knn": {field_name: {"vector": vector, "k
↪": count}}}}

    if namespace:
        query_body["query"] = {
            "bool": {
                "must": [
                    {"match": {"namespace": namespace}},
                    {"knn": {field_name: {"vector": vector, "k": count}}},
                ],
            },
        }

    response = self.client.search(index=self.index_name, body=query_body)

    return [
        BaseVectorStoreDriver.Entry(
            id=hit["_id"],
            namespace=hit["_source"].get("namespace") if namespace else None,
            score=hit["_score"],
            vector=hit["_source"].get("vector") if include_vectors else None,
            meta=hit["_source"].get("metadata") if include_metadata else None,
        )
        for hit in response["hits"]["hits"]
    ]
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/opensearch_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    """Inserts or updates a vector in OpenSearch.

    If a vector with the given vector ID already exists, it is updated; otherwise, a␣
↪new vector is inserted.
    Metadata associated with the vector can also be provided.
    """
    vector_id = vector_id or utils.str_to_hash(str(vector))
    doc = {"vector": vector, "namespace": namespace, "metadata": meta}
    doc.update(kwargs)
    response = self.client.index(index=self.index_name, id=vector_id, body=doc)

    return response["_id"]
```

pgai

- `__all__ = ['PgAiKnowledgeBaseVectorStoreDriver']` module-attribute

Bases: *BaseVectorStoreDriver* “)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
@define
class PgAiKnowledgeBaseVectorStoreDriver(BaseVectorStoreDriver):
    connection_string: str = field(kw_only=True, metadata={"serializable": True})
    knowledge_base_name: str = field(kw_only=True, metadata={"serializable": True})
    embedding_driver: BaseEmbeddingDriver = field(
        default=Factory(lambda: DummyEmbeddingDriver()),
        metadata={"serializable": True},
        kw_only=True,
        init=False,
    )
    _engine: sqlalchemy.Engine = field(default=None, kw_only=True, alias="engine",␣
↪metadata={"serializable": False})

    @lazy_property()
    def engine(self) -> sqlalchemy.Engine:
        return import_optional_dependency("sqlalchemy").create_engine(self.connection_
↪string)

    def query(
        self,
```

(continues on next page)

(continued from previous page)

```
        query: str | TextArtifact | ImageArtifact,
        *,
        count: Optional[int] = BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        if isinstance(query, ImageArtifact):
            raise ValueError(f"{self.__class__.__name__} does not support querying␣
↪with Image Artifacts.")

        sqlalchemy = import_optional_dependency("sqlalchemy")

        with sqlalchemy.orm.Session(self.engine) as session:
            rows = session.query(sqlalchemy.func.aidb.retrieve_text(self.knowledge_
↪base_name, query, count)).all()

        entries = []
        for (row,) in rows:
            # Remove the first and last parentheses from the row and list by commas
            # Example: (foo,bar) -> [foo, bar]
            row_list = "".join(row.replace("(", "", 1).rsplit(")", 1)).split(",")
            entries.append(
                BaseVectorStoreDriver.Entry(
                    id=row_list[0],
                    score=float(row_list[2]),
                    meta={"artifact": TextArtifact(row_list[1]).to_json()},
                )
            )

        return entries

    def upsert_vector(
        self,
        vector: list[float],
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        raise NotImplementedError(f"{self.__class__.__name__} does not support vector␣
↪upsert.")

    def upsert_text_artifact(
        self,
        artifact: TextArtifact,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        vector_id: Optional[str] = None,
        **kwargs,
    ) -> str:
        raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪artifact upsert.")
```

(continues on next page)

(continued from previous page)

```
    def upsert_text(
        self,
        string: str,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪upsert.")

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪BaseVectorStoreDriver.Entry:
        raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")

    def load_artifacts(self, *, namespace: Optional[str] = None) -> ListArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪Artifact loading.")

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")
```

- `_engine = field(default=None, kw_only=True, alias='engine', metadata={'serializable': False})` class-attribute instance-attribute
- `connection_string = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `embedding_driver = field(default=Factory(lambda: DummyEmbeddingDriver()), metadata={'serializable': True}, kw_only=True, init=False)` class-attribute instance-attribute
- `knowledge_base_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

delete_vector(vector_id)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

engine()

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
@lazy_property()
def engine(self) -> sqlalchemy.Engine:
    return import_optional_dependency("sqlalchemy").create_engine(self.connection_
↪string)
```

load_artifacts(*, namespace=None)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def load_artifacts(self, *, namespace: Optional[str] = None) -> ListArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support Artifact␣
↪loading.")
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")
```

load_entry(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪BaseVectorStoreDriver.Entry:
    raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")
```

query(query, *, count=BaseVectorStoreDriver.DEFAULT_QUERY_COUNT, **kwargs)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def query(
    self,
    query: str | TextArtifact | ImageArtifact,
    *,
    count: Optional[int] = BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    if isinstance(query, ImageArtifact):
        raise ValueError(f"{self.__class__.__name__} does not support querying with␣
↪Image Artifacts.")

    sqlalchemy = import_optional_dependency("sqlalchemy")

    with sqlalchemy.orm.Session(self.engine) as session:
        rows = session.query(sqlalchemy.func.aidb.retrieve_text(self.knowledge_base_
↪name, query, count)).all()
```

(continues on next page)

(continued from previous page)

```
    entries = []
    for (row,) in rows:
        # Remove the first and last parentheses from the row and list by commas
        # Example: (foo,bar) -> [foo, bar]
        row_list = "".join(row.replace("(", "", 1).rsplit(")", 1)).split(",")
        entries.append(
            BaseVectorStoreDriver.Entry(
                id=row_list[0],
                score=float(row_list[2]),
                meta={"artifact": TextArtifact(row_list[1]).to_json()},
            )
        )

    return entries
```

upsert_text(string, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def upsert_text(
    self,
    string: str,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪upsert.")
```

upsert_text_artifact(artifact, namespace=None, meta=None, vector_id=None, **kwargs)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def upsert_text_artifact(
    self,
    artifact: TextArtifact,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    vector_id: Optional[str] = None,
    **kwargs,
) -> str:
    raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪artifact upsert.")
```

upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    raise NotImplementedError(f"{self.__class__.__name__} does not support vector␣
↪upsert.")
```

pgai_knowledge_base_vector_store_driver

- `logger = logging.getLogger(__name__)` module-attribute

Bases: *BaseVectorStoreDriver* “)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
@define
class PgAiKnowledgeBaseVectorStoreDriver(BaseVectorStoreDriver):
    connection_string: str = field(kw_only=True, metadata={"serializable": True})
    knowledge_base_name: str = field(kw_only=True, metadata={"serializable": True})
    embedding_driver: BaseEmbeddingDriver = field(
        default=Factory(lambda: DummyEmbeddingDriver()),
        metadata={"serializable": True},
        kw_only=True,
        init=False,
    )
    _engine: sqlalchemy.Engine = field(default=None, kw_only=True, alias="engine",␣
↪metadata={"serializable": False})

    @lazy_property()
    def engine(self) -> sqlalchemy.Engine:
        return import_optional_dependency("sqlalchemy").create_engine(self.connection_
↪string)

    def query(
        self,
        query: str | TextArtifact | ImageArtifact,
        *,
        count: Optional[int] = BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        if isinstance(query, ImageArtifact):
            raise ValueError(f"{self.__class__.__name__} does not support querying␣
↪with Image Artifacts.")

        sqlalchemy = import_optional_dependency("sqlalchemy")

        with sqlalchemy.orm.Session(self.engine) as session:
            rows = session.query(sqlalchemy.func.aidb.retrieve_text(self.knowledge_
↪base_name, query, count)).all()
```

(continues on next page)

(continued from previous page)

```
        entries = []
        for (row,) in rows:
            # Remove the first and last parentheses from the row and list by commas
            # Example: (foo,bar) -> [foo, bar]
            row_list = "".join(row.replace("(", "", 1).rsplit(")", 1)).split(",")
            entries.append(
                BaseVectorStoreDriver.Entry(
                    id=row_list[0],
                    score=float(row_list[2]),
                    meta={"artifact": TextArtifact(row_list[1]).to_json()},
                )
            )

        return entries

    def upsert_vector(
        self,
        vector: list[float],
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        raise NotImplementedError(f"{self.__class__.__name__} does not support vector␣
↪upsert.")

    def upsert_text_artifact(
        self,
        artifact: TextArtifact,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        vector_id: Optional[str] = None,
        **kwargs,
    ) -> str:
        raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪artifact upsert.")

    def upsert_text(
        self,
        string: str,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪upsert.")

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪BaseVectorStoreDriver.Entry:
        raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
```

(continues on next page)

(continued from previous page)

```
↪loading.")

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")

    def load_artifacts(self, *, namespace: Optional[str] = None) -> ListArtifact:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪Artifact loading.")

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")
```

- `_engine = field(default=None, kw_only=True, alias='engine', metadata={'serializable': False})` class-attribute instance-attribute
- `connection_string = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `embedding_driver = field(default=Factory(lambda: DummyEmbeddingDriver()), metadata={'serializable': True}, kw_only=True, init=False)` class-attribute instance-attribute
- `knowledge_base_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

delete_vector(vector_id)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

engine()

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
@lazy_property()
def engine(self) -> sqlalchemy.Engine:
    return import_optional_dependency("sqlalchemy").create_engine(self.connection_
↪string)
```

load_artifacts(*, namespace=None)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def load_artifacts(self, *, namespace: Optional[str] = None) -> ListArtifact:
    raise NotImplementedError(f"{self.__class__.__name__} does not support Artifact␣
↪loading.")
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")
```

load_entry(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪BaseVectorStoreDriver.Entry:
    raise NotImplementedError(f"{self.__class__.__name__} does not support entry␣
↪loading.")
```

query(query, *, count=BaseVectorStoreDriver.DEFAULT_QUERY_COUNT, **kwargs)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def query(
    self,
    query: str | TextArtifact | ImageArtifact,
    *,
    count: Optional[int] = BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    if isinstance(query, ImageArtifact):
        raise ValueError(f"{self.__class__.__name__} does not support querying with␣
↪Image Artifacts.")

    sqlalchemy = import_optional_dependency("sqlalchemy")

    with sqlalchemy.orm.Session(self.engine) as session:
        rows = session.query(sqlalchemy.func.aidb.retrieve_text(self.knowledge_base_
↪name, query, count)).all()

    entries = []
    for (row,) in rows:
        # Remove the first and last parentheses from the row and list by commas
        # Example: (foo,bar) -> [foo, bar]
        row_list = "".join(row.replace("(", "", 1).rsplit(")", 1)).split(",")
        entries.append(
            BaseVectorStoreDriver.Entry(
                id=row_list[0],
                score=float(row_list[2]),
                meta={"artifact": TextArtifact(row_list[1]).to_json()},
            )
        )

    return entries
```

upsert_text(string, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def upsert_text(
    self,
    string: str,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪upsert.")
```

upsert_text_artifact(artifact, namespace=None, meta=None, vector_id=None, **kwargs)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def upsert_text_artifact(
    self,
    artifact: TextArtifact,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    vector_id: Optional[str] = None,
    **kwargs,
) -> str:
    raise NotImplementedError(f"{self.__class__.__name__} does not support text␣
↪artifact upsert.")
```

upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/pgai_knowledge_base_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    raise NotImplementedError(f"{self.__class__.__name__} does not support vector␣
↪upsert.")
```

pgvector

- `__all__ = ['PgVectorVectorStoreDriver']` module-attribute

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="pgvector.mdx#griptape.drivers.vector.pgvector.PgVectorVectorStoreDriver.connection_string" title="connection_string = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.pgvector.PgVectorVectorStoreDriver.connection_string)"><code class="language-python">	*Optional[str]*	An optional string describing the target Postgres database instance.
<a href="pgvector.mdx#griptape.drivers.vector.pgvector.PgVectorVectorStoreDriver.create_engine_params" title="create_engine_params = field(factory=dict, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.pgvector.PgVectorVectorStoreDriver.create_engine_params)"><code class="language-python">	*dict*	Additional configuration params passed when creating the database connection.

continues on next page

Table 284 – continued from previous page

Name	Type	Description
<a href="pgvector.mdx#griptape.drivers.vector.pgvector.PgVectorVectorStoreDriver.engine" title="engine() (griptape.drivers.vector.pgvector.PgVectorVectorStoreDriver.engine)"><code class="language-python">	*Engine*	An optional sqlalchemy Postgres engine to use.
<a href="pgvector.mdx#griptape.drivers.vector.pgvector.PgVectorVectorStoreDriver.table_name" title="table_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.pgvector.PgVectorVectorStoreDriver.table_name)"><code class="language-python">	*str*	Optionally specify the name of the table to used to store vectors.

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
@define
class PgVectorVectorStoreDriver(BaseVectorStoreDriver):
    """A vector store driver to Postgres using the PGVector extension.

    Attributes:
        connection_string: An optional string describing the target Postgres database instance.
        create_engine_params: Additional configuration params passed when creating the database connection.
        engine: An optional sqlalchemy Postgres engine to use.
        table_name: Optionally specify the name of the table to used to store vectors.
    """

    connection_string: Optional[str] = field(default=None, kw_only=True, metadata={"serializable": True})
```

(continues on next page)

(continued from previous page)

```
    create_engine_params: dict = field(factory=dict, kw_only=True, metadata={
↪"serializable": True})
    table_name: str = field(kw_only=True, metadata={"serializable": True})
    _model: Any = field(default=Factory(lambda self: self.default_vector_model(),␣
↪takes_self=True))
    _engine: Optional[sqlalchemy.Engine] = field(
        default=None, kw_only=True, alias="engine", metadata={"serializable": False}
    )

    @connection_string.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_connection_string(self, _: Attribute, connection_string:␣
↪Optional[str]) -> None:
        # If an engine is provided, the connection string is not used.
        if self._engine is not None:
            return

        # If an engine is not provided, a connection string is required.
        if connection_string is None:
            raise ValueError("An engine or connection string is required")

        if not connection_string.startswith("postgresql://"):
            raise ValueError("The connection string must describe a Postgres database␣
↪connection")

    @lazy_property()
    def engine(self) -> sqlalchemy.Engine:
        return import_optional_dependency("sqlalchemy").create_engine(
            self.connection_string, **self.create_engine_params
        )

    def setup(
        self,
        *,
        create_schema: bool = True,
        install_uuid_extension: bool = True,
        install_vector_extension: bool = True,
    ) -> None:
        """Provides a mechanism to initialize the database schema and extensions."""
        sqlalchemy_sql = import_optional_dependency("sqlalchemy.sql")

        if install_uuid_extension:
            with self.engine.begin() as conn:
                conn.execute(sqlalchemy_sql.text(CREATE EXTENSION IF NOT EXISTS "uuid-
↪ossp";))

        if install_vector_extension:
            with self.engine.begin() as conn:
                conn.execute(sqlalchemy_sql.text(CREATE EXTENSION IF NOT EXISTS
↪"vector";))

        if create_schema:
```

(continues on next page)

(continued from previous page)

```
            self._model.metadata.create_all(self.engine)

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        """Inserts or updates a vector in the collection."""
        sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

        with sqlalchemy_orm.Session(self.engine) as session:
            obj = self._model(id=vector_id, vector=vector, namespace=namespace,␣
↪meta=meta, **kwargs)

            obj = session.merge(obj)
            session.commit()

            return str(obj.id)

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪BaseVectorStoreDriver.Entry:
        """Retrieves a specific vector entry from the collection based on its␣
↪identifier and optional namespace."""
        sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

        with sqlalchemy_orm.Session(self.engine) as session:
            result = session.get(self._model, vector_id)

            return BaseVectorStoreDriver.Entry(
                id=result.id,
                vector=result.vector,
                namespace=result.namespace,
                meta=result.meta,
            )

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Retrieves all vector entries from the collection, optionally filtering to␣
↪only those that match the provided namespace."""
        sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

        with sqlalchemy_orm.Session(self.engine) as session:
            query = session.query(self._model)
            if namespace:
                query = query.filter_by(namespace=namespace)

            results = query.all()
```

(continues on next page)

(continued from previous page)

```
            return [
                BaseVectorStoreDriver.Entry(
                    id=str(result.id),
                    vector=result.vector,
                    namespace=result.namespace,
                    meta=result.meta,
                )
                for result in results
            ]

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        distance_metric: str = "cosine_distance",
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Performs a search on the collection to find vectors similar to the␣
↪provided input vector, optionally filtering to only those that match the provided␣
↪namespace."""
        sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

        count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT

        distance_metrics = {
            "cosine_distance": self._model.vector.cosine_distance,
            "l2_distance": self._model.vector.l2_distance,
            "inner_product": self._model.vector.max_inner_product,
        }

        if distance_metric not in distance_metrics:
            raise ValueError("Invalid distance metric provided")

        op = distance_metrics[distance_metric]

        with sqlalchemy_orm.Session(self.engine) as session:
            # The query should return both the vector and the distance metric score.
            query_result = session.query(self._model, op(vector).label("score")).
↪order_by(op(vector))  # pyright: ignore[reportOptionalCall]

            filter_kwargs: Optional[OrderedDict] = None

            if namespace is not None:
                filter_kwargs = OrderedDict(namespace=namespace)

            if "filter" in kwargs and isinstance(kwargs["filter"], dict):
                filter_kwargs = filter_kwargs or OrderedDict()
                filter_kwargs.update(kwargs["filter"])
```

(continues on next page)

(continued from previous page)

```
            if filter_kwargs is not None:
                query_result = query_result.filter_by(**filter_kwargs)

            results = query_result.limit(count).all()

            return [
                BaseVectorStoreDriver.Entry(
                    id=str(result[0].id),
                    vector=result[0].vector if include_vectors else None,
                    score=result[1],
                    meta=result[0].meta,
                    namespace=result[0].namespace,
                )
                for result in results
            ]

    def default_vector_model(self) -> Any:
        pgvector_sqlalchemy = import_optional_dependency("pgvector.sqlalchemy")
        sqlalchemy = import_optional_dependency("sqlalchemy")
        sqlalchemy_dialects_postgresql = import_optional_dependency("sqlalchemy.
↪dialects.postgresql")
        sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

        @dataclass
        class VectorModel(sqlalchemy_orm.declarative_base()):
            __tablename__ = self.table_name

            id = sqlalchemy.Column(
                sqlalchemy_dialects_postgresql.UUID(as_uuid=True),
                primary_key=True,
                default=uuid.uuid4,
                unique=True,
                nullable=False,
            )
            vector = sqlalchemy.Column(pgvector_sqlalchemy.Vector())
            namespace = sqlalchemy.Column(sqlalchemy.String)
            meta = sqlalchemy.Column(sqlalchemy.JSON)

        return VectorModel

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")
```

- `_engine = field(default=None, kw_only=True, alias='engine', metadata={'serializable': False})` class-attribute instance-attribute
- `_model = field(default=Factory(lambda self: self.default_vector_model(), takes_self=True))` class-attribute instance-attribute
- `connection_string = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `create_engine_params = field(factory=dict, kw_only=True, metadata={'serializable':`

`True})` class-attribute instance-attribute

- `table_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

default_vector_model()

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def default_vector_model(self) -> Any:
    pgvector_sqlalchemy = import_optional_dependency("pgvector.sqlalchemy")
    sqlalchemy = import_optional_dependency("sqlalchemy")
    sqlalchemy_dialects_postgresql = import_optional_dependency("sqlalchemy.dialects.
↪postgresql")
    sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

    @dataclass
    class VectorModel(sqlalchemy_orm.declarative_base()):
        __tablename__ = self.table_name

        id = sqlalchemy.Column(
            sqlalchemy_dialects_postgresql.UUID(as_uuid=True),
            primary_key=True,
            default=uuid.uuid4,
            unique=True,
            nullable=False,
        )
        vector = sqlalchemy.Column(pgvector_sqlalchemy.Vector())
        namespace = sqlalchemy.Column(sqlalchemy.String)
        meta = sqlalchemy.Column(sqlalchemy.JSON)

    return VectorModel
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

engine()

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
@lazy_property()
def engine(self) -> sqlalchemy.Engine:
    return import_optional_dependency("sqlalchemy").create_engine(
        self.connection_string, **self.create_engine_params
    )
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Retrieves all vector entries from the collection, optionally filtering to only␣
↪those that match the provided namespace."""
    sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

    with sqlalchemy_orm.Session(self.engine) as session:
        query = session.query(self._model)
        if namespace:
            query = query.filter_by(namespace=namespace)

        results = query.all()

        return [
            BaseVectorStoreDriver.Entry(
                id=str(result.id),
                vector=result.vector,
                namespace=result.namespace,
                meta=result.meta,
            )
            for result in results
        ]
```

load_entry(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪BaseVectorStoreDriver.Entry:
    """Retrieves a specific vector entry from the collection based on its identifier␣
↪and optional namespace."""
    sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

    with sqlalchemy_orm.Session(self.engine) as session:
        result = session.get(self._model, vector_id)

        return BaseVectorStoreDriver.Entry(
            id=result.id,
            vector=result.vector,
            namespace=result.namespace,
            meta=result.meta,
        )
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, distance_metric='cosine_distance', **kwargs)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    distance_metric: str = "cosine_distance",
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    """Performs a search on the collection to find vectors similar to the provided input vector, optionally filtering to only those that match the provided namespace."""
    sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

    count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT

    distance_metrics = {
        "cosine_distance": self._model.vector.cosine_distance,
        "l2_distance": self._model.vector.l2_distance,
        "inner_product": self._model.vector.max_inner_product,
    }

    if distance_metric not in distance_metrics:
        raise ValueError("Invalid distance metric provided")

    op = distance_metrics[distance_metric]

    with sqlalchemy_orm.Session(self.engine) as session:
        # The query should return both the vector and the distance metric score.
        query_result = session.query(self._model, op(vector).label("score")).order_by(op(vector))  # pyright: ignore[reportOptionalCall]

        filter_kwargs: Optional[OrderedDict] = None

        if namespace is not None:
            filter_kwargs = OrderedDict(namespace=namespace)

        if "filter" in kwargs and isinstance(kwargs["filter"], dict):
            filter_kwargs = filter_kwargs or OrderedDict()
            filter_kwargs.update(kwargs["filter"])

        if filter_kwargs is not None:
            query_result = query_result.filter_by(**filter_kwargs)

        results = query_result.limit(count).all()

        return [
            BaseVectorStoreDriver.Entry(
                id=str(result[0].id),
                vector=result[0].vector if include_vectors else None,
                score=result[1],
                meta=result[0].meta,
```

(continues on next page)

(continued from previous page)

```
                namespace=result[0].namespace,
            )
            for result in results
        ]
```

setup(*, create_schema=True, install_uuid_extension=True, install_vector_extension=True)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def setup(
    self,
    *,
    create_schema: bool = True,
    install_uuid_extension: bool = True,
    install_vector_extension: bool = True,
) -> None:
    """Provides a mechanism to initialize the database schema and extensions."""
    sqlalchemy_sql = import_optional_dependency("sqlalchemy.sql")

    if install_uuid_extension:
        with self.engine.begin() as conn:
            conn.execute(sqlalchemy_sql.text(CREATE EXTENSION IF NOT EXISTS "uuid-ossp
↪";))

    if install_vector_extension:
        with self.engine.begin() as conn:
            conn.execute(sqlalchemy_sql.text(CREATE EXTENSION IF NOT EXISTS "vector";
↪))

    if create_schema:
        self._model.metadata.create_all(self.engine)
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    """Inserts or updates a vector in the collection."""
    sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

    with sqlalchemy_orm.Session(self.engine) as session:
        obj = self._model(id=vector_id, vector=vector, namespace=namespace, meta=meta,
↪ **kwargs)
```

(continues on next page)

(continued from previous page)

```
        obj = session.merge(obj)
        session.commit()

        return str(obj.id)
```

validate*connection_string(*, connection_string)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
@connection_string.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_connection_string(self, _: Attribute, connection_string: Optional[str]) -
↪> None:
    # If an engine is provided, the connection string is not used.
    if self._engine is not None:
        return

    # If an engine is not provided, a connection string is required.
    if connection_string is None:
        raise ValueError("An engine or connection string is required")

    if not connection_string.startswith("postgresql://"):
        raise ValueError("The connection string must describe a Postgres database␣
↪connection")
```

pgvector_vector_store_driver

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="pgvector_vector_store_driver.mdx#griptape.drivers.vector.pgvector_vector_store_driver.PgVectorVectorStoreDriver.connection_string" title="connection_string = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.pgvector_vector_store_driver.PgVectorVectorStoreDriver.connection_string)"><code class="language-python">	*Optional[str]*	An optional string describing the target Postgres database instance.
<a href="pgvector_vector_store_driver.mdx#griptape.drivers.vector.pgvector_vector_store_driver.PgVectorVectorStoreDriver.create_engine_params" title="create_engine_params = field(factory=dict, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.pgvector_vector_store_driver.PgVectorVectorStoreDriver.create_engine_params)"><code class="language-python">	*dict*	Additional configuration params passed when creating the database connection.

continues on next page

Table 285 – continued from previous page

Name	Type	Description
<a href="pgvector_vector_store_driver.mdx#griptape.drivers.vector.pgvector_vector_store_driver.PgVectorVectorStoreDriver.engine" title="engine() (griptape.drivers.vector.pgvector_vector_store_driver.PgVectorVectorStoreDriver.engine)"><code class="language-python">	*Engine*	An optional sqlalchemy Postgres engine to use.
<a href="pgvector_vector_store_driver.mdx#griptape.drivers.vector.pgvector_vector_store_driver.PgVectorVectorStoreDriver.table_name" title="table_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.pgvector_vector_store_driver.PgVectorVectorStoreDriver.table_name)"><code class="language-python">	*str*	Optionally specify the name of the table to used to store vectors.

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
@define
class PgVectorVectorStoreDriver(BaseVectorStoreDriver):
    """A vector store driver to Postgres using the PGVector extension.

    Attributes:
        connection_string: An optional string describing the target Postgres database instance.
        create_engine_params: Additional configuration params passed when creating the database connection.
        engine: An optional sqlalchemy Postgres engine to use.
        table_name: Optionally specify the name of the table to used to store vectors.
```

(continues on next page)

(continued from previous page)

```
    """

    connection_string: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    create_engine_params: dict = field(factory=dict, kw_only=True, metadata={
↪"serializable": True})
    table_name: str = field(kw_only=True, metadata={"serializable": True})
    _model: Any = field(default=Factory(lambda self: self.default_vector_model(),␣
↪takes_self=True))
    _engine: Optional[sqlalchemy.Engine] = field(
        default=None, kw_only=True, alias="engine", metadata={"serializable": False}
    )

    @connection_string.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_connection_string(self, _: Attribute, connection_string:␣
↪Optional[str]) -> None:
        # If an engine is provided, the connection string is not used.
        if self._engine is not None:
            return

        # If an engine is not provided, a connection string is required.
        if connection_string is None:
            raise ValueError("An engine or connection string is required")

        if not connection_string.startswith("postgresql://"):
            raise ValueError("The connection string must describe a Postgres database␣
↪connection")

    @lazy_property()
    def engine(self) -> sqlalchemy.Engine:
        return import_optional_dependency("sqlalchemy").create_engine(
            self.connection_string, **self.create_engine_params
        )

    def setup(
        self,
        *,
        create_schema: bool = True,
        install_uuid_extension: bool = True,
        install_vector_extension: bool = True,
    ) -> None:
        """Provides a mechanism to initialize the database schema and extensions."""
        sqlalchemy_sql = import_optional_dependency("sqlalchemy.sql")

        if install_uuid_extension:
            with self.engine.begin() as conn:
                conn.execute(sqlalchemy_sql.text(CREATE EXTENSION IF NOT EXISTS "uuid-
↪ossp";))

        if install_vector_extension:
            with self.engine.begin() as conn:
```

(continues on next page)

(continued from previous page)

```
                conn.execute(sqlalchemy_sql.text(CREATE EXTENSION IF NOT EXISTS
↪"vector";))

        if create_schema:
            self._model.metadata.create_all(self.engine)

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        """Inserts or updates a vector in the collection."""
        sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

        with sqlalchemy_orm.Session(self.engine) as session:
            obj = self._model(id=vector_id, vector=vector, namespace=namespace,␣
↪meta=meta, **kwargs)

            obj = session.merge(obj)
            session.commit()

            return str(obj.id)

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪BaseVectorStoreDriver.Entry:
        """Retrieves a specific vector entry from the collection based on its␣
↪identifier and optional namespace."""
        sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

        with sqlalchemy_orm.Session(self.engine) as session:
            result = session.get(self._model, vector_id)

            return BaseVectorStoreDriver.Entry(
                id=result.id,
                vector=result.vector,
                namespace=result.namespace,
                meta=result.meta,
            )

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Retrieves all vector entries from the collection, optionally filtering to␣
↪only those that match the provided namespace."""
        sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

        with sqlalchemy_orm.Session(self.engine) as session:
            query = session.query(self._model)
            if namespace:
```

(continues on next page)

(continued from previous page)

```
                query = query.filter_by(namespace=namespace)

            results = query.all()

            return [
                BaseVectorStoreDriver.Entry(
                    id=str(result.id),
                    vector=result.vector,
                    namespace=result.namespace,
                    meta=result.meta,
                )
                for result in results
            ]

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        distance_metric: str = "cosine_distance",
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Performs a search on the collection to find vectors similar to the
↪provided input vector, optionally filtering to only those that match the provided
↪namespace."""
        sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

        count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT

        distance_metrics = {
            "cosine_distance": self._model.vector.cosine_distance,
            "l2_distance": self._model.vector.l2_distance,
            "inner_product": self._model.vector.max_inner_product,
        }

        if distance_metric not in distance_metrics:
            raise ValueError("Invalid distance metric provided")

        op = distance_metrics[distance_metric]

        with sqlalchemy_orm.Session(self.engine) as session:
            # The query should return both the vector and the distance metric score.
            query_result = session.query(self._model, op(vector).label("score")).
↪order_by(op(vector))  # pyright: ignore[reportOptionalCall]

            filter_kwargs: Optional[OrderedDict] = None

            if namespace is not None:
                filter_kwargs = OrderedDict(namespace=namespace)
```

(continues on next page)

(continued from previous page)

```
            if "filter" in kwargs and isinstance(kwargs["filter"], dict):
                filter_kwargs = filter_kwargs or OrderedDict()
                filter_kwargs.update(kwargs["filter"])

            if filter_kwargs is not None:
                query_result = query_result.filter_by(**filter_kwargs)

            results = query_result.limit(count).all()

            return [
                BaseVectorStoreDriver.Entry(
                    id=str(result[0].id),
                    vector=result[0].vector if include_vectors else None,
                    score=result[1],
                    meta=result[0].meta,
                    namespace=result[0].namespace,
                )
                for result in results
            ]

    def default_vector_model(self) -> Any:
        pgvector_sqlalchemy = import_optional_dependency("pgvector.sqlalchemy")
        sqlalchemy = import_optional_dependency("sqlalchemy")
        sqlalchemy_dialects_postgresql = import_optional_dependency("sqlalchemy.
↪dialects.postgresql")
        sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

        @dataclass
        class VectorModel(sqlalchemy_orm.declarative_base()):
            __tablename__ = self.table_name

            id = sqlalchemy.Column(
                sqlalchemy_dialects_postgresql.UUID(as_uuid=True),
                primary_key=True,
                default=uuid.uuid4,
                unique=True,
                nullable=False,
            )
            vector = sqlalchemy.Column(pgvector_sqlalchemy.Vector())
            namespace = sqlalchemy.Column(sqlalchemy.String)
            meta = sqlalchemy.Column(sqlalchemy.JSON)

        return VectorModel

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")
```

- `_engine = field(default=None, kw_only=True, alias='engine', metadata={'serializable': False})` class-attribute instance-attribute
- `_model = field(default=Factory(lambda self: self.default_vector_model(), takes_self=True))` class-attribute instance-attribute

- `connection_string = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `create_engine_params = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `table_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

default_vector_model()

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def default_vector_model(self) -> Any:
    pgvector_sqlalchemy = import_optional_dependency("pgvector.sqlalchemy")
    sqlalchemy = import_optional_dependency("sqlalchemy")
    sqlalchemy_dialects_postgresql = import_optional_dependency("sqlalchemy.dialects.
↪postgresql")
    sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

    @dataclass
    class VectorModel(sqlalchemy_orm.declarative_base()):
        __tablename__ = self.table_name

        id = sqlalchemy.Column(
            sqlalchemy_dialects_postgresql.UUID(as_uuid=True),
            primary_key=True,
            default=uuid.uuid4,
            unique=True,
            nullable=False,
        )
        vector = sqlalchemy.Column(pgvector_sqlalchemy.Vector())
        namespace = sqlalchemy.Column(sqlalchemy.String)
        meta = sqlalchemy.Column(sqlalchemy.JSON)

    return VectorModel
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

engine()

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
@lazy_property()
def engine(self) -> sqlalchemy.Engine:
    return import_optional_dependency("sqlalchemy").create_engine(
        self.connection_string, **self.create_engine_params
    )
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Retrieves all vector entries from the collection, optionally filtering to only␣
↪those that match the provided namespace."""
    sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

    with sqlalchemy_orm.Session(self.engine) as session:
        query = session.query(self._model)
        if namespace:
            query = query.filter_by(namespace=namespace)

        results = query.all()

        return [
            BaseVectorStoreDriver.Entry(
                id=str(result.id),
                vector=result.vector,
                namespace=result.namespace,
                meta=result.meta,
            )
            for result in results
        ]
```

load_entry(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪BaseVectorStoreDriver.Entry:
    """Retrieves a specific vector entry from the collection based on its identifier␣
↪and optional namespace."""
    sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

    with sqlalchemy_orm.Session(self.engine) as session:
        result = session.get(self._model, vector_id)

        return BaseVectorStoreDriver.Entry(
            id=result.id,
            vector=result.vector,
            namespace=result.namespace,
            meta=result.meta,
        )
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, distance_metric='cosine_distance', **kwargs)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    distance_metric: str = "cosine_distance",
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    """Performs a search on the collection to find vectors similar to the provided␣
↪input vector, optionally filtering to only those that match the provided namespace."
↪""
    sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

    count = count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT

    distance_metrics = {
        "cosine_distance": self._model.vector.cosine_distance,
        "l2_distance": self._model.vector.l2_distance,
        "inner_product": self._model.vector.max_inner_product,
    }

    if distance_metric not in distance_metrics:
        raise ValueError("Invalid distance metric provided")

    op = distance_metrics[distance_metric]

    with sqlalchemy_orm.Session(self.engine) as session:
        # The query should return both the vector and the distance metric score.
        query_result = session.query(self._model, op(vector).label("score")).order_
↪by(op(vector))  # pyright: ignore[reportOptionalCall]

        filter_kwargs: Optional[OrderedDict] = None

        if namespace is not None:
            filter_kwargs = OrderedDict(namespace=namespace)

        if "filter" in kwargs and isinstance(kwargs["filter"], dict):
            filter_kwargs = filter_kwargs or OrderedDict()
            filter_kwargs.update(kwargs["filter"])

        if filter_kwargs is not None:
            query_result = query_result.filter_by(**filter_kwargs)

        results = query_result.limit(count).all()

        return [
            BaseVectorStoreDriver.Entry(
                id=str(result[0].id),
                vector=result[0].vector if include_vectors else None,
                score=result[1],
                meta=result[0].meta,
```

(continues on next page)

(continued from previous page)

```
                namespace=result[0].namespace,
            )
            for result in results
        ]
```

setup(*, create_schema=True, install_uuid_extension=True, install_vector_extension=True)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def setup(
    self,
    *,
    create_schema: bool = True,
    install_uuid_extension: bool = True,
    install_vector_extension: bool = True,
) -> None:
    """Provides a mechanism to initialize the database schema and extensions."""
    sqlalchemy_sql = import_optional_dependency("sqlalchemy.sql")

    if install_uuid_extension:
        with self.engine.begin() as conn:
            conn.execute(sqlalchemy_sql.text(CREATE EXTENSION IF NOT EXISTS "uuid-ossp
↪";))

    if install_vector_extension:
        with self.engine.begin() as conn:
            conn.execute(sqlalchemy_sql.text(CREATE EXTENSION IF NOT EXISTS "vector";
↪))

    if create_schema:
        self._model.metadata.create_all(self.engine)
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    """Inserts or updates a vector in the collection."""
    sqlalchemy_orm = import_optional_dependency("sqlalchemy.orm")

    with sqlalchemy_orm.Session(self.engine) as session:
        obj = self._model(id=vector_id, vector=vector, namespace=namespace, meta=meta,
↪ **kwargs)
```

(continues on next page)

(continued from previous page)

```
        obj = session.merge(obj)
        session.commit()

        return str(obj.id)
```

validate*connection_string(*, connection_string)

Source Code in griptape/drivers/vector/pgvector_vector_store_driver.py

```
@connection_string.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_connection_string(self, _: Attribute, connection_string: Optional[str]) -
↪> None:
    # If an engine is provided, the connection string is not used.
    if self._engine is not None:
        return

    # If an engine is not provided, a connection string is required.
    if connection_string is None:
        raise ValueError("An engine or connection string is required")

    if not connection_string.startswith("postgresql://"):
        raise ValueError("The connection string must describe a Postgres database␣
↪connection")
```

pinecone

- `__all__ = ['PineconeVectorStoreDriver']` module-attribute

Bases: *BaseVectorStoreDriver* “)

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
@define
class PineconeVectorStoreDriver(BaseVectorStoreDriver):
    api_key: str = field(kw_only=True, metadata={"serializable": True})
    index_name: str = field(kw_only=True, metadata={"serializable": True})
    environment: str = field(kw_only=True, metadata={"serializable": True})
    project_name: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    _client: Optional[pinecone.Pinecone] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )
    _index: Optional[pinecone.Index] = field(
        default=None, kw_only=True, alias="index", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> pinecone.Pinecone:
        return import_optional_dependency("pinecone").Pinecone(
            api_key=self.api_key,
```

(continues on next page)

(continued from previous page)

```
            environment=self.environment,
            project_name=self.project_name,
        )

    @lazy_property()
    def index(self) -> pinecone.data.index.Index:
        return self.client.Index(self.index_name)

    def upsert_vector(
        self,
        vector: list[float],
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        vector_id = vector_id or str_to_hash(str(vector))

        params: dict[str, Any] = {"namespace": namespace} | kwargs

        self.index.upsert(vectors=[(vector_id, vector, meta)], **params)  # pyright:␣
↪ignore[reportArgumentType]

        return vector_id

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        result = self.index.fetch(ids=[vector_id], namespace=namespace).to_dict()  #␣
↪pyright: ignore[reportAttributeAccessIssue]
        vectors = list(result["vectors"].values())

        if len(vectors) > 0:
            vector = vectors[0]

            return BaseVectorStoreDriver.Entry(
                id=vector["id"],
                meta=vector["metadata"],
                vector=vector["values"],
                namespace=result["namespace"],
            )
        return None

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        # This is a hacky way to query up to 10,000 values from Pinecone. Waiting on␣
↪an official API for fetching
        # all values from a namespace:
        # https://community.pinecone.io/t/is-there-a-way-to-query-all-the-vectors-and-
↪or-metadata-from-a-namespace/797/5

        results = self.index.query(
            vector=self.embedding_driver.embed("", vector_operation="query"),
```

(continues on next page)

(continued from previous page)

```
            top_k=10000,
            include_metadata=True,
            namespace=namespace,
        )

        return [
            BaseVectorStoreDriver.Entry(
                id=r["id"],
                vector=r["values"],
                meta=r["metadata"],
                namespace=results["namespace"],  # pyright: ignore[reportIndexIssue]
            )
            for r in results["matches"]  # pyright: ignore[reportIndexIssue]
        ]

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        include_metadata: bool = True,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        params = {
            "top_k": count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
            "namespace": namespace,
            "include_values": include_vectors,
            "include_metadata": include_metadata,
        } | kwargs

        results = self.index.query(vector=vector, **params)

        return [
            BaseVectorStoreDriver.Entry(
                id=r["id"],
                vector=r["values"],
                score=r["score"],
                meta=r["metadata"],
                namespace=results["namespace"],  # pyright: ignore[reportIndexIssue]
            )
            for r in results["matches"]  # pyright: ignore[reportIndexIssue]
        ]

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `_index = field(default=None, kw_only=True, alias='index', metadata={'serializable':`

`False})` class-attribute instance-attribute

- `api_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `environment = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `index_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `project_name = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
@lazy_property()
def client(self) -> pinecone.Pinecone:
    return import_optional_dependency("pinecone").Pinecone(
        api_key=self.api_key,
        environment=self.environment,
        project_name=self.project_name,
    )
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

index()

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
@lazy_property()
def index(self) -> pinecone.data.index.Index:
    return self.client.Index(self.index_name)
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    # This is a hacky way to query up to 10,000 values from Pinecone. Waiting on an␣
↪official API for fetching
    # all values from a namespace:
    # https://community.pinecone.io/t/is-there-a-way-to-query-all-the-vectors-and-or-
↪metadata-from-a-namespace/797/5

    results = self.index.query(
```

(continues on next page)

(continued from previous page)

```
        vector=self.embedding_driver.embed("", vector_operation="query"),
        top_k=10000,
        include_metadata=True,
        namespace=namespace,
    )

    return [
        BaseVectorStoreDriver.Entry(
            id=r["id"],
            vector=r["values"],
            meta=r["metadata"],
            namespace=results["namespace"],  # pyright: ignore[reportIndexIssue]
        )
        for r in results["matches"]  # pyright: ignore[reportIndexIssue]
    ]
```

load_entry(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    result = self.index.fetch(ids=[vector_id], namespace=namespace).to_dict()  #␣
↪pyright: ignore[reportAttributeAccessIssue]
    vectors = list(result["vectors"].values())

    if len(vectors) > 0:
        vector = vectors[0]

        return BaseVectorStoreDriver.Entry(
            id=vector["id"],
            meta=vector["metadata"],
            vector=vector["values"],
            namespace=result["namespace"],
        )
    return None
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, include_metadata=True, **kwargs)

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    include_metadata: bool = True,
    **kwargs,
```

(continues on next page)

(continued from previous page)

```
) -> list[BaseVectorStoreDriver.Entry]:
    params = {
        "top_k": count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
        "namespace": namespace,
        "include_values": include_vectors,
        "include_metadata": include_metadata,
    } | kwargs

    results = self.index.query(vector=vector, **params)

    return [
        BaseVectorStoreDriver.Entry(
            id=r["id"],
            vector=r["values"],
            score=r["score"],
            meta=r["metadata"],
            namespace=results["namespace"],  # pyright: ignore[reportIndexIssue]
        )
        for r in results["matches"]  # pyright: ignore[reportIndexIssue]
    ]
```

upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    vector_id = vector_id or str_to_hash(str(vector))

    params: dict[str, Any] = {"namespace": namespace} | kwargs

    self.index.upsert(vectors=[(vector_id, vector, meta)], **params)  # pyright:␣
↪ignore[reportArgumentType]

    return vector_id
```

pinecone_vector_store_driver

Bases: *BaseVectorStoreDriver* “)

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
@define
class PineconeVectorStoreDriver(BaseVectorStoreDriver):
    api_key: str = field(kw_only=True, metadata={"serializable": True})
    index_name: str = field(kw_only=True, metadata={"serializable": True})
```

(continues on next page)

(continued from previous page)

```
    environment: str = field(kw_only=True, metadata={"serializable": True})
    project_name: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    _client: Optional[pinecone.Pinecone] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )
    _index: Optional[pinecone.Index] = field(
        default=None, kw_only=True, alias="index", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> pinecone.Pinecone:
        return import_optional_dependency("pinecone").Pinecone(
            api_key=self.api_key,
            environment=self.environment,
            project_name=self.project_name,
        )

    @lazy_property()
    def index(self) -> pinecone.data.index.Index:
        return self.client.Index(self.index_name)

    def upsert_vector(
        self,
        vector: list[float],
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        vector_id = vector_id or str_to_hash(str(vector))

        params: dict[str, Any] = {"namespace": namespace} | kwargs

        self.index.upsert(vectors=[(vector_id, vector, meta)], **params)  # pyright:␣
↪ignore[reportArgumentType]

        return vector_id

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        result = self.index.fetch(ids=[vector_id], namespace=namespace).to_dict()  #␣
↪pyright: ignore[reportAttributeAccessIssue]
        vectors = list(result["vectors"].values())

        if len(vectors) > 0:
            vector = vectors[0]

            return BaseVectorStoreDriver.Entry(
                id=vector["id"],
                meta=vector["metadata"],
                vector=vector["values"],
```

(continues on next page)

(continued from previous page)

```
                namespace=result["namespace"],
            )
        return None

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        # This is a hacky way to query up to 10,000 values from Pinecone. Waiting on␣
↪an official API for fetching
        # all values from a namespace:
        # https://community.pinecone.io/t/is-there-a-way-to-query-all-the-vectors-and-
↪or-metadata-from-a-namespace/797/5

        results = self.index.query(
            vector=self.embedding_driver.embed("", vector_operation="query"),
            top_k=10000,
            include_metadata=True,
            namespace=namespace,
        )

        return [
            BaseVectorStoreDriver.Entry(
                id=r["id"],
                vector=r["values"],
                meta=r["metadata"],
                namespace=results["namespace"],  # pyright: ignore[reportIndexIssue]
            )
            for r in results["matches"]  # pyright: ignore[reportIndexIssue]
        ]

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        include_metadata: bool = True,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        params = {
            "top_k": count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
            "namespace": namespace,
            "include_values": include_vectors,
            "include_metadata": include_metadata,
        } | kwargs

        results = self.index.query(vector=vector, **params)

        return [
            BaseVectorStoreDriver.Entry(
                id=r["id"],
                vector=r["values"],
```

(continues on next page)

(continued from previous page)

```
                score=r["score"],
                meta=r["metadata"],
                namespace=results["namespace"],  # pyright: ignore[reportIndexIssue]
            )
            for r in results["matches"]  # pyright: ignore[reportIndexIssue]
        ]

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `_index = field(default=None, kw_only=True, alias='index', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `environment = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `index_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `project_name = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
@lazy_property()
def client(self) -> pinecone.Pinecone:
    return import_optional_dependency("pinecone").Pinecone(
        api_key=self.api_key,
        environment=self.environment,
        project_name=self.project_name,
    )
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

index()

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
@lazy_property()
def index(self) -> pinecone.data.index.Index:
    return self.client.Index(self.index_name)
```

load_entries(*, namespace=None)

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    # This is a hacky way to query up to 10,000 values from Pinecone. Waiting on an␣
↪official API for fetching
    # all values from a namespace:
    # https://community.pinecone.io/t/is-there-a-way-to-query-all-the-vectors-and-or-
↪metadata-from-a-namespace/797/5

    results = self.index.query(
        vector=self.embedding_driver.embed("", vector_operation="query"),
        top_k=10000,
        include_metadata=True,
        namespace=namespace,
    )

    return [
        BaseVectorStoreDriver.Entry(
            id=r["id"],
            vector=r["values"],
            meta=r["metadata"],
            namespace=results["namespace"],  # pyright: ignore[reportIndexIssue]
        )
        for r in results["matches"]  # pyright: ignore[reportIndexIssue]
    ]
```

load_entry(vector_id, *, namespace=None)

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    result = self.index.fetch(ids=[vector_id], namespace=namespace).to_dict()  #␣
↪pyright: ignore[reportAttributeAccessIssue]
    vectors = list(result["vectors"].values())

    if len(vectors) > 0:
        vector = vectors[0]

        return BaseVectorStoreDriver.Entry(
            id=vector["id"],
            meta=vector["metadata"],
            vector=vector["values"],
            namespace=result["namespace"],
        )
```

(continues on next page)

(continued from previous page)

```
    return None
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, include_metadata=True, **kwargs)

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    include_metadata: bool = True,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    params = {
        "top_k": count or BaseVectorStoreDriver.DEFAULT_QUERY_COUNT,
        "namespace": namespace,
        "include_values": include_vectors,
        "include_metadata": include_metadata,
    } | kwargs

    results = self.index.query(vector=vector, **params)

    return [
        BaseVectorStoreDriver.Entry(
            id=r["id"],
            vector=r["values"],
            score=r["score"],
            meta=r["metadata"],
            namespace=results["namespace"],  # pyright: ignore[reportIndexIssue]
        )
        for r in results["matches"]  # pyright: ignore[reportIndexIssue]
    ]
```

upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/pinecone_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    vector_id = vector_id or str_to_hash(str(vector))
```

(continues on next page)

(continued from previous page)

```
    params: dict[str, Any] = {"namespace": namespace} | kwargs

    self.index.upsert(vectors=[(vector_id, vector, meta)], **params)  # pyright:␣
↪ignore[reportArgumentType]

    return vector_id
```

qdrant

- `__all__ = ['QdrantVectorStoreDriver']` module-attribute

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="qdrant.mdx#griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.location" title="location = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.location)"><code class="language-python">	*Optional[str]*	An optional location for the Qdrant client. If set to ':memory:', an in-memory client is used.
<a href="qdrant.mdx#griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.url" title="url = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.url)"><code class="language-python">	*Optional[str]*	An optional Qdrant API URL.

continues on next page

Table 286 – continued from previous page

Name	Type	Description
<a href="qdrant.mdx#griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.host" title="host = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.host)"><code class="language-python">	*Optional[str]*	An optional Qdrant host.
<a href="qdrant.mdx#griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.path" title="path = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.path)"><code class="language-python">	*Optional[str]*	Persistence path for QdrantLocal. Default: None
<a href="qdrant.mdx#griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.port" title="port = field(default=6333, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.port)"><code class="language-python">	*int*	The port number for the Qdrant client. Defaults: 6333.

continues on next page

Table 286 – continued from previous page

Name	Type	Description
<a href="qdrant.mdx#griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.grpc_port" title="grpc_port = field(default=6334, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.grpc_port)"><code class="language-python">	*int*	The gRPC port number for the Qdrant client. Defaults: 6334.
<a href="qdrant.mdx#griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.prefer_grpc" title="prefer_grpc = field(default=False, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.prefer_grpc)"><code class="language-python">	*bool*	A boolean indicating whether to prefer gRPC over HTTP. Defaults: False.

continues on next page

Table 286 – continued from previous page

Name	Type	Description
<a href="qdrant.mdx#griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.force_disable_check_same_thread' title="force_disable_check_same_thread = field(default=False, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.force_disable_check_same_thread) class="language-python">	*Optional[bool]*	For QdrantLocal, force disable check_same_thread. Default: False Only use this if you can guarantee that you can resolve the thread safety outside QdrantClient.
<a href="qdrant.mdx#griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.timeout" title="timeout = field(default=5, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.timeout)"><code class="language-python">	*Optional[int]*	Timeout for REST and gRPC API requests. Default: 5 seconds for REST and unlimited for gRPC

continues on next page

Table 286 – continued from previous page

Name	Type	Description
<a href="qdrant.mdx#griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.api_key" title="api_key = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.api_key)"><c class="language-python">	*Optional[str]*	API key for authentication in Qdrant Cloud. Defaults: False
<a href="qdrant.mdx#griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.https" title="https = field(default=True, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.https)"><cod class="language-python">	*bool*	If true - use HTTPS(SSL) protocol. Default: True
<a href="qdrant.mdx#griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.prefix" title="prefix = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.prefix)"><code class="language-python">	*Optional[str]*	Add prefix to the REST URL path. Example: service/v1 will result in Example: service/v1 will result in http://localhost:6333/service/v1/{qdrant-endpoint} for REST API. Defaults: None

continues on next page

Table 286 – continued from previous page

Name	Type	Description
<a href="qdrant.mdx#griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.distance" title="distance = field(default=DEFAULT_DISTANCE, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.distance)"><code class="language-python">	*str*	The distance metric to be used for the vectors. Defaults: 'COSINE'.
<a href="qdrant.mdx#griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.collection_name" title="collection_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.collection_name)"><code class="language-python">	*str*	The name of the Qdrant collection.

continues on next page

Table 286 – continued from previous page

Name	Type	Description
<a href="qdrant.mdx#griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.vector_name" title="vector_name = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.vector_name)"><code class="language-python">	*Optional[str]*	An optional name for the vectors.
<a href="qdrant.mdx#griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.content_payload_key" title="content_payload_key = field(default=CONTENT_PAYLOAD_KEY, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant.QdrantVectorStoreDriver.content_payload_key)"><code class="language-python">	*str*	The key for the content payload in the metadata. Defaults: 'data'.

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
@define
class QdrantVectorStoreDriver(BaseVectorStoreDriver):
    """Vector Store Driver for Qdrant.

    Attributes:
        location: An optional location for the Qdrant client. If set to :memory:, an␣
↪in-memory client is used.
        url: An optional Qdrant API URL.
        host: An optional Qdrant host.
```

(continues on next page)

(continued from previous page)

```
        path: Persistence path for QdrantLocal. Default: None
        port: The port number for the Qdrant client. Defaults: 6333.
        grpc_port: The gRPC port number for the Qdrant client. Defaults: 6334.
        prefer_grpc: A boolean indicating whether to prefer gRPC over HTTP. Defaults:␣
↪False.
        force_disable_check_same_thread: For QdrantLocal, force disable check_same_
↪thread. Default: False Only use this if you can guarantee that you can resolve the␣
↪thread safety outside QdrantClient.
        timeout: Timeout for REST and gRPC API requests. Default: 5 seconds for REST␣
↪and unlimited for gRPC
        api_key: API key for authentication in Qdrant Cloud. Defaults: False
        https: If true - use HTTPS(SSL) protocol. Default: True
        prefix: Add prefix to the REST URL path. Example: service/v1 will result in␣
↪Example: service/v1 will result in http://localhost:6333/service/v1/{qdrant-
↪endpoint} for REST API. Defaults: None
        distance: The distance metric to be used for the vectors. Defaults: COSINE.
        collection_name: The name of the Qdrant collection.
        vector_name: An optional name for the vectors.
        content_payload_key: The key for the content payload in the metadata.␣
↪Defaults: data.
    """

    location: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    url: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    host: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    path: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    port: int = field(default=6333, kw_only=True, metadata={"serializable": True})
    grpc_port: int = field(default=6334, kw_only=True, metadata={"serializable": True}
↪)
    prefer_grpc: bool = field(default=False, kw_only=True, metadata={"serializable":␣
↪True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    https: bool = field(default=True, kw_only=True, metadata={"serializable": True})
    prefix: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    force_disable_check_same_thread: Optional[bool] = field(
        default=False,
        kw_only=True,
        metadata={"serializable": True},
    )
    timeout: Optional[int] = field(default=5, kw_only=True, metadata={"serializable":␣
↪True})
    distance: str = field(default=DEFAULT_DISTANCE, kw_only=True, metadata={
↪"serializable": True})
    collection_name: str = field(kw_only=True, metadata={"serializable": True})
    vector_name: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
```

(continues on next page)

(continued from previous page)

```
    content_payload_key: str = field(default=CONTENT_PAYLOAD_KEY, kw_only=True,␣
↪metadata={"serializable": True})
    _client: Optional[QdrantClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> QdrantClient:
        return import_optional_dependency("qdrant_client").QdrantClient(
            location=self.location,
            url=self.url,
            host=self.host,
            path=self.path,
            port=self.port,
            prefer_grpc=self.prefer_grpc,
            grpc_port=self.grpc_port,
            api_key=self.api_key,
            https=self.https,
            prefix=self.prefix,
            force_disable_check_same_thread=self.force_disable_check_same_thread,
            timeout=self.timeout,
        )

    def delete_vector(self, vector_id: str) -> None:
        """Delete a vector from the Qdrant collection based on its ID.

        Parameters:
            vector_id (str | id): ID of the vector to delete.
        """
        deletion_response = self.client.delete(
            collection_name=self.collection_name,
            points_selector=import_optional_dependency("qdrant_client.http.models").
↪PointIdsList(points=[vector_id]),
        )
        if deletion_response.status == import_optional_dependency("qdrant_client.http.
↪models").UpdateStatus.COMPLETED:
            logging.info("ID %s is successfully deleted", vector_id)

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Query the Qdrant collection based on a query vector.

        Parameters:
            vector (list[float]): Query vector.
            count (Optional[int]): Optional number of results to return.
```

(continues on next page)

(continued from previous page)

```
            namespace (Optional[str]): Optional namespace of the vectors.
            include_vectors (bool): Whether to include vectors in the results.

        Returns:
            list[BaseVectorStoreDriver.Entry]: List of Entry objects.
        """
        # Create a search request
        request = {"collection_name": self.collection_name, "query_vector": vector,
↪"limit": count}
        request = {k: v for k, v in request.items() if v is not None}
        results = self.client.search(**request)

        # Convert results to QueryResult objects
        return [
            BaseVectorStoreDriver.Entry(
                id=str(result.id),
                vector=result.vector if include_vectors else [],  # pyright:␣
↪ignore[reportArgumentType]
                score=result.score,
                meta={k: v for k, v in result.payload.items() if k not in ["_score",
↪"_tensor_facets"]},
            )
            for result in results
            if result.payload is not None
        ]

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        content: Optional[str] = None,
        **kwargs,
    ) -> str:
        """Upsert vectors into the Qdrant collection.

        Parameters:
            vector (list[float]): The vector to be upserted.
            vector_id (Optional[str]): Optional vector ID.
            namespace (Optional[str]): Optional namespace for the vector.
            meta (Optional[dict]): Optional dictionary containing metadata.
            content (Optional[str]): The text content to be included in the payload.

        Returns:
            str: The ID of the upserted vector.
        """
        if vector_id is None:
            vector_id = str(uuid.uuid5(uuid.NAMESPACE_DNS, str(vector)))

        if meta is None:
```

(continues on next page)

(continued from previous page)

```
            meta = {}

        if content:
            meta[self.content_payload_key] = content

        points = import_optional_dependency("qdrant_client.http.models").Batch(
            ids=[vector_id],
            vectors=[vector],
            payloads=[meta] if meta else None,
        )

        self.client.upsert(collection_name=self.collection_name, points=points)
        return vector_id

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        """Load a vector entry from the Qdrant collection based on its ID.

        Parameters:
            vector_id (str): ID of the vector to load.
            namespace (str, optional): Optional namespace of the vector.

        Returns:
            Optional[BaseVectorStoreDriver.Entry]: Vector entry if found, else None.
        """
        results = self.client.retrieve(collection_name=self.collection_name,␣
↪ids=[vector_id])
        if results:
            entry = results[0]
            if entry.payload is None:
                entry.payload = {}
            return BaseVectorStoreDriver.Entry(
                id=str(entry.id),
                vector=entry.vector if entry.vector is not None else [],  # pyright:␣
↪ignore[reportArgumentType]
                meta={k: v for k, v in entry.payload.items() if k not in ["_score", "_
↪tensor_facets"]},
            )
        return None

    def load_entries(self, *, namespace: Optional[str] = None, **kwargs) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Load vector entries from the Qdrant collection.

        Parameters:
            namespace: Optional namespace of the vectors.

        Returns:
            List of points.
        """
        results = self.client.retrieve(
            collection_name=self.collection_name,
```

(continues on next page)

(continued from previous page)

```
            ids=kwargs.get("ids", []),
            with_payload=kwargs.get("with_payload", True),
            with_vectors=kwargs.get("with_vectors", True),
        )

        return [
            BaseVectorStoreDriver.Entry(
                id=str(entry.id),
                vector=entry.vector if kwargs.get("with_vectors", True) else [],  #␣
↪pyright: ignore[reportArgumentType]
                meta={k: v for k, v in entry.payload.items() if k not in ["_score", "_
↪tensor_facets"]},
            )
            for entry in results
            if entry.payload is not None
        ]
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `collection_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `content_payload_key = field(default=CONTENT_PAYLOAD_KEY, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `distance = field(default=DEFAULT_DISTANCE, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `force_disable_check_same_thread = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `grpc_port = field(default=6334, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `host = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `https = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `location = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `path = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `port = field(default=6333, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `prefer_grpc = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `prefix = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

- `timeout = field(default=5, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute
- `url = field(default=None, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute
- `vector_name = field(default=None, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute

client()

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
@lazy_property()
def client(self) -> QdrantClient:
    return import_optional_dependency("qdrant_client").QdrantClient(
        location=self.location,
        url=self.url,
        host=self.host,
        path=self.path,
        port=self.port,
        prefer_grpc=self.prefer_grpc,
        grpc_port=self.grpc_port,
        api_key=self.api_key,
        https=self.https,
        prefix=self.prefix,
        force_disable_check_same_thread=self.force_disable_check_same_thread,
        timeout=self.timeout,
    )
```

delete_vector(vector_id)

Delete a vector from the Qdrant collection based on its ID.

Parameters

Name	Type	Description	Default
vector_id	*str I id*	ID of the vector to delete. 	*required*

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> None:
    """Delete a vector from the Qdrant collection based on its ID.

    Parameters:
        vector_id (str | id): ID of the vector to delete.
    """
    deletion_response = self.client.delete(
        collection_name=self.collection_name,
        points_selector=import_optional_dependency("qdrant_client.http.models").
↪PointIdsList(points=[vector_id]),
```

(continues on next page)

(continued from previous page)

```
    )
    if deletion_response.status == import_optional_dependency("qdrant_client.http.
↪models").UpdateStatus.COMPLETED:
        logging.info("ID %s is successfully deleted", vector_id)
```

load_entries(*, namespace=None, **kwargs)

Load vector entries from the Qdrant collection.

Parameters

Name	Type	Description	Default
namespace	*Optional[str]*	Optional namespace of the vectors. 	*None*

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	List of points.

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None, **kwargs) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Load vector entries from the Qdrant collection.

    Parameters:
        namespace: Optional namespace of the vectors.

    Returns:
        List of points.
    """
    results = self.client.retrieve(
        collection_name=self.collection_name,
        ids=kwargs.get("ids", []),
        with_payload=kwargs.get("with_payload", True),
        with_vectors=kwargs.get("with_vectors", True),
    )

    return [
        BaseVectorStoreDriver.Entry(
            id=str(entry.id),
            vector=entry.vector if kwargs.get("with_vectors", True) else [],  #␣
↪pyright: ignore[reportArgumentType]
            meta={k: v for k, v in entry.payload.items() if k not in ["_score", "_
```

(continues on next page)

(continued from previous page)

```
↪tensor_facets"]},
        )
        for entry in results
        if entry.payload is not None
    ]
```

load_entry(vector_id, *, namespace=None)

Load a vector entry from the Qdrant collection based on its ID.

Parameters

Name	Type	Description	Default
vector_id	*str*	ID of the vector to load. 	*required*
namespace	*str*	Optional namespace of the vector. 	*None*

Returns

Type	Description
<a href="./#griptape.drivers.vector.BaseVectorStoreDriver.Entry" title="Entry (griptape.drivers.vector.BaseVectorStoreDriver.Entry)"><code class="language-python">C	Optional[BaseVectorStoreDriver.Entry]: Vector entry if found, else None.

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    """Load a vector entry from the Qdrant collection based on its ID.

    Parameters:
        vector_id (str): ID of the vector to load.
        namespace (str, optional): Optional namespace of the vector.

    Returns:
        Optional[BaseVectorStoreDriver.Entry]: Vector entry if found, else None.
    """
    results = self.client.retrieve(collection_name=self.collection_name, ids=[vector_
↪id])
    if results:
        entry = results[0]
        if entry.payload is None:
```

(continues on next page)

(continued from previous page)

```
            entry.payload = {}
        return BaseVectorStoreDriver.Entry(
            id=str(entry.id),
            vector=entry.vector if entry.vector is not None else [],  # pyright:␣
↪ignore[reportArgumentType]
            meta={k: v for k, v in entry.payload.items() if k not in ["_score", "_
↪tensor_facets"]},
        )
    return None
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs)

Query the Qdrant collection based on a query vector.

Parameters

Name	Type	Description	Default
vector	<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	Query vector. 	*required*
count	*Optional[int]*	Optional number of results to return. 	*None*
namespace	*Optional[str]*	Optional namespace of the vectors. 	*None*
include_vectors	*bool*	Whether to include vectors in the results. 	*False*

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	list[BaseVectorStoreDriver.Entry]: List of Entry objects.

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
```

(continues on next page)

(continued from previous page)

```
    include_vectors: bool = False,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    """Query the Qdrant collection based on a query vector.

    Parameters:
        vector (list[float]): Query vector.
        count (Optional[int]): Optional number of results to return.
        namespace (Optional[str]): Optional namespace of the vectors.
        include_vectors (bool): Whether to include vectors in the results.

    Returns:
        list[BaseVectorStoreDriver.Entry]: List of Entry objects.
    """
    # Create a search request
    request = {"collection_name": self.collection_name, "query_vector": vector, "limit
↪": count}
    request = {k: v for k, v in request.items() if v is not None}
    results = self.client.search(**request)

    # Convert results to QueryResult objects
    return [
        BaseVectorStoreDriver.Entry(
            id=str(result.id),
            vector=result.vector if include_vectors else [],  # pyright:␣
↪ignore[reportArgumentType]
            score=result.score,
            meta={k: v for k, v in result.payload.items() if k not in ["_score", "_
↪tensor_facets"]},
        )
        for result in results
        if result.payload is not None
    ]
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, content=None, **kwargs)

Upsert vectors into the Qdrant collection.

Parameters

Name	Type	Description	Default
vector	<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	The vector to be upserted. 	*required*
vector_id	*Optional[str]*	Optional vector ID. 	*None*

continues on next page

Table 294 – continued from previous page

Name	Type	Description	Default
namespace	*Optional[str]*	Optional namespace for the vector. 	*None*
meta	*Optional[dict]*	Optional dictionary containing metadata. 	*None*
content	*Optional[str]*	The text content to be included in the payload. 	*None*

Returns

Name	Type	Description
str	*str*	The ID of the upserted vector.

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    content: Optional[str] = None,
    **kwargs,
) -> str:
    """Upsert vectors into the Qdrant collection.

    Parameters:
        vector (list[float]): The vector to be upserted.
        vector_id (Optional[str]): Optional vector ID.
        namespace (Optional[str]): Optional namespace for the vector.
        meta (Optional[dict]): Optional dictionary containing metadata.
        content (Optional[str]): The text content to be included in the payload.

    Returns:
        str: The ID of the upserted vector.
    """
    if vector_id is None:
        vector_id = str(uuid.uuid5(uuid.NAMESPACE_DNS, str(vector)))

    if meta is None:
        meta = {}

    if content:
        meta[self.content_payload_key] = content
```

(continues on next page)

(continued from previous page)

```
    points = import_optional_dependency("qdrant_client.http.models").Batch(
        ids=[vector_id],
        vectors=[vector],
        payloads=[meta] if meta else None,
    )

    self.client.upsert(collection_name=self.collection_name, points=points)
    return vector_id
```

qdrant_vector_store_driver

- `CONTENT_PAYLOAD_KEY = 'data'` module-attribute
- `DEFAULT_DISTANCE = 'Cosine'` module-attribute

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="qdrant_vector_store_driver.mdx#griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.location" title="location = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.location)"><code class="language-python">	*Optional[str]*	An optional location for the Qdrant client. If set to ':memory:', an in-memory client is used.

continues on next page

Table 296 – continued from previous page

Name	Type	Description
<a href="qdrant_vector_store_driver.mdx#griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.url" title="url = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.url)"><code class="language-python">	*Optional[str]*	An optional Qdrant API URL.
<a href="qdrant_vector_store_driver.mdx#griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.host" title="host = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.host)"><code class="language-python">	*Optional[str]*	An optional Qdrant host.

continues on next page

Table 296 – continued from previous page

Name	Type	Description
<a href="qdrant_vector_store_driver.mdx#griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.path" title="path = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant_vector_store_driver.QdrantVector-StoreDriver.path)"><code class="language-python">	*Optional[str]*	Persistence path for QdrantLocal. Default: None
<a href="qdrant_vector_store_driver.mdx#griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.port" title="port = field(default=6333, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant_vector_store_driver.QdrantVector-StoreDriver.port)"><code class="language-python">	*int*	The port number for the Qdrant client. Defaults: 6333.

continues on next page

Table 296 – continued from previous page

Name	Type	Description
<a href="qdrant_vector_store_driver.mdx#griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.grpc_port" title="grpc_port = field(default=6334, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.grpc_port)"><code class="language-python">	*int*	The gRPC port number for the Qdrant client. Defaults: 6334.
<a href="qdrant_vector_store_driver.mdx#griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.prefer_grpc" title="prefer_grpc = field(default=False, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.prefer_grpc)"><code class="language-python">	*bool*	A boolean indicating whether to prefer gRPC over HTTP. Defaults: False.

continues on next page

Table 296 – continued from previous page

Name	Type	Description
<a href="qdrant_vector_store_driver.mdx#griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.force_disable_check_same_thread" title="force_disable_check_same_thread = field(default=False, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.force_disable_check_same_thread)" class="language-python">	*Optional[bool]*	For QdrantLocal, force disable check_same_thread. Default: False Only use this if you can guarantee that you can resolve the thread safety outside QdrantClient.
<a href="qdrant_vector_store_driver.mdx#griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.timeout" title="timeout = field(default=5, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.timeout)"><code class="language-python">	*Optional[int]*	Timeout for REST and gRPC API requests. Default: 5 seconds for REST and unlimited for gRPC

continues on next page

Table 296 – continued from previous page

Name	Type	Description
<a href="qdrant_vector_store_driver.mdx#griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.api_key" title="api_key = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.api_key)"><c class="language-python">	*Optional[str]*	API key for authentication in Qdrant Cloud. Defaults: False
<a href="qdrant_vector_store_driver.mdx#griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.https" title="https = field(default=True, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.https)"><cod class="language-python">	*bool*	If true - use HTTPS(SSL) protocol. Default: True

continues on next page

Table 296 – continued from previous page

Name	Type	Description
<a href="qdrant_vector_store_driver.mdx#griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.prefix" title="prefix = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.prefix)"><code class="language-python">	*Optional[str]*	Add prefix to the REST URL path. Example: service/v1 will result in Example: service/v1 will result in http://localhost:6333/service/v1/{qdrant-endpoint} for REST API. Defaults: None
<a href="qdrant_vector_store_driver.mdx#griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.distance" title="distance = field(default=DEFAULT_DISTANCE, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.distance)"><code class="language-python">	*str*	The distance metric to be used for the vectors. Defaults: 'COSINE'.

continues on next page

Table 296 – continued from previous page

Name	Type	Description
<a href="qdrant_vector_store_driver.mdx#griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.collection_name" title="collection_name = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.collection_name)"><code class="language-python">	*str*	The name of the Qdrant collection.
<a href="qdrant_vector_store_driver.mdx#griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.vector_name" title="vector_name = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.vector_name)"><code class="language-python">	*Optional[str]*	An optional name for the vectors.

continues on next page

Table 296 – continued from previous page

Name	Type	Description
<a href="qdrant_vector_store_driver.mdx#griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.content_payload_key" title="content_payload_key = field(default=CONTENT_PAYLOAD_KEY, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.qdrant_vector_store_driver.QdrantVectorStoreDriver.content_payload_key)"><code class="language-python">	*str*	The key for the content payload in the metadata. Defaults: 'data'.

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
@define
class QdrantVectorStoreDriver(BaseVectorStoreDriver):
    """Vector Store Driver for Qdrant.

    Attributes:
        location: An optional location for the Qdrant client. If set to :memory:, an␣
↪in-memory client is used.
        url: An optional Qdrant API URL.
        host: An optional Qdrant host.
        path: Persistence path for QdrantLocal. Default: None
        port: The port number for the Qdrant client. Defaults: 6333.
        grpc_port: The gRPC port number for the Qdrant client. Defaults: 6334.
        prefer_grpc: A boolean indicating whether to prefer gRPC over HTTP. Defaults:␣
↪False.
        force_disable_check_same_thread: For QdrantLocal, force disable check_same_
↪thread. Default: False Only use this if you can guarantee that you can resolve the␣
↪thread safety outside QdrantClient.
        timeout: Timeout for REST and gRPC API requests. Default: 5 seconds for REST␣
↪and unlimited for gRPC
        api_key: API key for authentication in Qdrant Cloud. Defaults: False
        https: If true - use HTTPS(SSL) protocol. Default: True
        prefix: Add prefix to the REST URL path. Example: service/v1 will result in␣
↪Example: service/v1 will result in http://localhost:6333/service/v1/{qdrant-
↪endpoint} for REST API. Defaults: None
```

(continues on next page)

(continued from previous page)

```
        distance: The distance metric to be used for the vectors. Defaults: COSINE.
        collection_name: The name of the Qdrant collection.
        vector_name: An optional name for the vectors.
        content_payload_key: The key for the content payload in the metadata.␣
↪Defaults: data.
    """

    location: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    url: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    host: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    path: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    port: int = field(default=6333, kw_only=True, metadata={"serializable": True})
    grpc_port: int = field(default=6334, kw_only=True, metadata={"serializable": True}
↪)
    prefer_grpc: bool = field(default=False, kw_only=True, metadata={"serializable":␣
↪True})
    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    https: bool = field(default=True, kw_only=True, metadata={"serializable": True})
    prefix: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    force_disable_check_same_thread: Optional[bool] = field(
        default=False,
        kw_only=True,
        metadata={"serializable": True},
    )
    timeout: Optional[int] = field(default=5, kw_only=True, metadata={"serializable":␣
↪True})
    distance: str = field(default=DEFAULT_DISTANCE, kw_only=True, metadata={
↪"serializable": True})
    collection_name: str = field(kw_only=True, metadata={"serializable": True})
    vector_name: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    content_payload_key: str = field(default=CONTENT_PAYLOAD_KEY, kw_only=True,␣
↪metadata={"serializable": True})
    _client: Optional[QdrantClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> QdrantClient:
        return import_optional_dependency("qdrant_client").QdrantClient(
            location=self.location,
            url=self.url,
            host=self.host,
            path=self.path,
            port=self.port,
            prefer_grpc=self.prefer_grpc,
```

(continues on next page)

(continued from previous page)

```
            grpc_port=self.grpc_port,
            api_key=self.api_key,
            https=self.https,
            prefix=self.prefix,
            force_disable_check_same_thread=self.force_disable_check_same_thread,
            timeout=self.timeout,
        )

    def delete_vector(self, vector_id: str) -> None:
        """Delete a vector from the Qdrant collection based on its ID.

        Parameters:
            vector_id (str | id): ID of the vector to delete.
        """
        deletion_response = self.client.delete(
            collection_name=self.collection_name,
            points_selector=import_optional_dependency("qdrant_client.http.models").
↪PointIdsList(points=[vector_id]),
        )
        if deletion_response.status == import_optional_dependency("qdrant_client.http.
↪models").UpdateStatus.COMPLETED:
            logging.info("ID %s is successfully deleted", vector_id)

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Query the Qdrant collection based on a query vector.

        Parameters:
            vector (list[float]): Query vector.
            count (Optional[int]): Optional number of results to return.
            namespace (Optional[str]): Optional namespace of the vectors.
            include_vectors (bool): Whether to include vectors in the results.

        Returns:
            list[BaseVectorStoreDriver.Entry]: List of Entry objects.
        """
        # Create a search request
        request = {"collection_name": self.collection_name, "query_vector": vector,
↪"limit": count}
        request = {k: v for k, v in request.items() if v is not None}
        results = self.client.search(**request)

        # Convert results to QueryResult objects
        return [
            BaseVectorStoreDriver.Entry(
```

(continues on next page)

(continued from previous page)

```
                id=str(result.id),
                vector=result.vector if include_vectors else [],  # pyright:␣
↪ignore[reportArgumentType]
                score=result.score,
                meta={k: v for k, v in result.payload.items() if k not in ["_score",
↪"_tensor_facets"]},
            )
            for result in results
            if result.payload is not None
        ]

    def upsert_vector(
        self,
        vector: list[float],
        *,
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        content: Optional[str] = None,
        **kwargs,
    ) -> str:
        """Upsert vectors into the Qdrant collection.

        Parameters:
            vector (list[float]): The vector to be upserted.
            vector_id (Optional[str]): Optional vector ID.
            namespace (Optional[str]): Optional namespace for the vector.
            meta (Optional[dict]): Optional dictionary containing metadata.
            content (Optional[str]): The text content to be included in the payload.

        Returns:
            str: The ID of the upserted vector.
        """
        if vector_id is None:
            vector_id = str(uuid.uuid5(uuid.NAMESPACE_DNS, str(vector)))

        if meta is None:
            meta = {}

        if content:
            meta[self.content_payload_key] = content

        points = import_optional_dependency("qdrant_client.http.models").Batch(
            ids=[vector_id],
            vectors=[vector],
            payloads=[meta] if meta else None,
        )

        self.client.upsert(collection_name=self.collection_name, points=points)
        return vector_id

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
```

(continues on next page)

(continued from previous page)

```
↪Optional[BaseVectorStoreDriver.Entry]:
        """Load a vector entry from the Qdrant collection based on its ID.

        Parameters:
            vector_id (str): ID of the vector to load.
            namespace (str, optional): Optional namespace of the vector.

        Returns:
            Optional[BaseVectorStoreDriver.Entry]: Vector entry if found, else None.
        """
        results = self.client.retrieve(collection_name=self.collection_name,␣
↪ids=[vector_id])
        if results:
            entry = results[0]
            if entry.payload is None:
                entry.payload = {}
            return BaseVectorStoreDriver.Entry(
                id=str(entry.id),
                vector=entry.vector if entry.vector is not None else [],  # pyright:␣
↪ignore[reportArgumentType]
                meta={k: v for k, v in entry.payload.items() if k not in ["_score", "_
↪tensor_facets"]},
            )
        return None

    def load_entries(self, *, namespace: Optional[str] = None, **kwargs) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Load vector entries from the Qdrant collection.

        Parameters:
            namespace: Optional namespace of the vectors.

        Returns:
            List of points.
        """
        results = self.client.retrieve(
            collection_name=self.collection_name,
            ids=kwargs.get("ids", []),
            with_payload=kwargs.get("with_payload", True),
            with_vectors=kwargs.get("with_vectors", True),
        )

        return [
            BaseVectorStoreDriver.Entry(
                id=str(entry.id),
                vector=entry.vector if kwargs.get("with_vectors", True) else [],  #␣
↪pyright: ignore[reportArgumentType]
                meta={k: v for k, v in entry.payload.items() if k not in ["_score", "_
↪tensor_facets"]},
            )
            for entry in results
            if entry.payload is not None
```

(continues on next page)

(continued from previous page)

```
        ]
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `collection_name = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `content_payload_key = field(default=CONTENT_PAYLOAD_KEY, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `distance = field(default=DEFAULT_DISTANCE, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `force_disable_check_same_thread = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `grpc_port = field(default=6334, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `host = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `https = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `location = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `path = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `port = field(default=6333, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `prefer_grpc = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `prefix = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `timeout = field(default=5, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `url = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `vector_name = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
@lazy_property()
def client(self) -> QdrantClient:
    return import_optional_dependency("qdrant_client").QdrantClient(
```

(continues on next page)

(continued from previous page)

```
        location=self.location,
        url=self.url,
        host=self.host,
        path=self.path,
        port=self.port,
        prefer_grpc=self.prefer_grpc,
        grpc_port=self.grpc_port,
        api_key=self.api_key,
        https=self.https,
        prefix=self.prefix,
        force_disable_check_same_thread=self.force_disable_check_same_thread,
        timeout=self.timeout,
    )
```

delete_vector(vector_id)

Delete a vector from the Qdrant collection based on its ID.

Parameters

Name	Type	Description	Default
vector_id	*str I id*	ID of the vector to delete. 	*required*

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> None:
    """Delete a vector from the Qdrant collection based on its ID.

    Parameters:
        vector_id (str | id): ID of the vector to delete.
    """
    deletion_response = self.client.delete(
        collection_name=self.collection_name,
        points_selector=import_optional_dependency("qdrant_client.http.models").
↪PointIdsList(points=[vector_id]),
    )
    if deletion_response.status == import_optional_dependency("qdrant_client.http.
↪models").UpdateStatus.COMPLETED:
        logging.info("ID %s is successfully deleted", vector_id)
```

load_entries(*, namespace=None, **kwargs)

Load vector entries from the Qdrant collection.

Parameters

Name	Type	Description	Default
namespace	*Optional[str]*	Optional namespace of the vectors. 	*None*

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	List of points.

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None, **kwargs) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Load vector entries from the Qdrant collection.

    Parameters:
        namespace: Optional namespace of the vectors.

    Returns:
        List of points.
    """
    results = self.client.retrieve(
        collection_name=self.collection_name,
        ids=kwargs.get("ids", []),
        with_payload=kwargs.get("with_payload", True),
        with_vectors=kwargs.get("with_vectors", True),
    )

    return [
        BaseVectorStoreDriver.Entry(
            id=str(entry.id),
            vector=entry.vector if kwargs.get("with_vectors", True) else [],  #␣
↪pyright: ignore[reportArgumentType]
            meta={k: v for k, v in entry.payload.items() if k not in ["_score", "_
↪tensor_facets"]},
        )
        for entry in results
        if entry.payload is not None
    ]
```

load_entry(vector_id, *, namespace=None)

Load a vector entry from the Qdrant collection based on its ID.

Parameters

Name	Type	Description	Default
vector_id	*str*	ID of the vector to load. 	*required*
namespace	*str*	Optional namespace of the vector. 	*None*

Returns

Type	Description
<a href="./#griptape.drivers.vector.BaseVectorStoreDriver.Entry" title="Entry (griptape.drivers.vector.BaseVectorStoreDriver.Entry)"><code class="language-python">C	Optional[BaseVectorStoreDriver.Entry]: Vector entry if found, else None.

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    """Load a vector entry from the Qdrant collection based on its ID.

    Parameters:
        vector_id (str): ID of the vector to load.
        namespace (str, optional): Optional namespace of the vector.

    Returns:
        Optional[BaseVectorStoreDriver.Entry]: Vector entry if found, else None.
    """
    results = self.client.retrieve(collection_name=self.collection_name, ids=[vector_
↪id])
    if results:
        entry = results[0]
        if entry.payload is None:
            entry.payload = {}
        return BaseVectorStoreDriver.Entry(
            id=str(entry.id),
            vector=entry.vector if entry.vector is not None else [],  # pyright:␣
↪ignore[reportArgumentType]
            meta={k: v for k, v in entry.payload.items() if k not in ["_score", "_
↪tensor_facets"]},
```

(continues on next page)

(continued from previous page)

```
        )
    return None
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs)

Query the Qdrant collection based on a query vector.

Parameters

Name	Type	Description	Default
vector	<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	Query vector. 	*required*
count	*Optional[int]*	Optional number of results to return. 	*None*
namespace	*Optional[str]*	Optional namespace of the vectors. 	*None*
include_vectors	*bool*	Whether to include vectors in the results. 	*False*

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	list[BaseVectorStoreDriver.Entry]: List of Entry objects.

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    """Query the Qdrant collection based on a query vector.

    Parameters:
        vector (list[float]): Query vector.
```

(continues on next page)

(continued from previous page)

```
        count (Optional[int]): Optional number of results to return.
        namespace (Optional[str]): Optional namespace of the vectors.
        include_vectors (bool): Whether to include vectors in the results.

    Returns:
        list[BaseVectorStoreDriver.Entry]: List of Entry objects.
    """
    # Create a search request
    request = {"collection_name": self.collection_name, "query_vector": vector, "limit
↪": count}
    request = {k: v for k, v in request.items() if v is not None}
    results = self.client.search(**request)

    # Convert results to QueryResult objects
    return [
        BaseVectorStoreDriver.Entry(
            id=str(result.id),
            vector=result.vector if include_vectors else [],  # pyright:␣
↪ignore[reportArgumentType]
            score=result.score,
            meta={k: v for k, v in result.payload.items() if k not in ["_score", "_
↪tensor_facets"]},
        )
        for result in results
        if result.payload is not None
    ]
```

upsert_vector(vector, *, vector_id=None, namespace=None, meta=None, content=None, **kwargs)

Upsert vectors into the Qdrant collection.

Parameters

Name	Type	Description	Default
vector	<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	The vector to be upserted. 	*required*
vector_id	*Optional[str]*	Optional vector ID. 	*None*
namespace	*Optional[str]*	Optional namespace for the vector. 	*None*
meta	*Optional[dict]*	Optional dictionary containing metadata. 	*None*

continues on next page

Table 304 – continued from previous page

Name	Type	Description	Default
content	*Optional[str]*	The text content to be included in the payload. 	*None*

Returns

Name	Type	Description
str	*str*	The ID of the upserted vector.

Source Code in griptape/drivers/vector/qdrant_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    *,
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    content: Optional[str] = None,
    **kwargs,
) -> str:
    """Upsert vectors into the Qdrant collection.

    Parameters:
        vector (list[float]): The vector to be upserted.
        vector_id (Optional[str]): Optional vector ID.
        namespace (Optional[str]): Optional namespace for the vector.
        meta (Optional[dict]): Optional dictionary containing metadata.
        content (Optional[str]): The text content to be included in the payload.

    Returns:
        str: The ID of the upserted vector.
    """
    if vector_id is None:
        vector_id = str(uuid.uuid5(uuid.NAMESPACE_DNS, str(vector)))

    if meta is None:
        meta = {}

    if content:
        meta[self.content_payload_key] = content

    points = import_optional_dependency("qdrant_client.http.models").Batch(
        ids=[vector_id],
        vectors=[vector],
        payloads=[meta] if meta else None,
    )
```

(continues on next page)

(continued from previous page)

```
    self.client.upsert(collection_name=self.collection_name, points=points)
    return vector_id
```

redis

- `__all__ = ['RedisVectorStoreDriver']` module-attribute

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="redis.mdx#griptape.drivers.vector.redis.RedisVectorStoreDriver.host" title="host = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.redis.RedisVectorStoreDriver.host)"><code class="language-python">	*str*	The host of the Redis instance.
<a href="redis.mdx#griptape.drivers.vector.redis.RedisVectorStoreDriver.port" title="port = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.redis.RedisVectorStoreDriver.port)"><code class="language-python">	*int*	The port of the Redis instance.

continues on next page

Table 306 – continued from previous page

Name	Type	Description
<a href="redis.mdx#griptape.drivers.vector.redis.RedisVectorStoreDriver.db" title="db = field(kw_only=True, default=0, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.redis.RedisVectorStoreDriver.db)"><code class="language-python">	*int*	The database of the Redis instance.
<a href="redis.mdx#griptape.drivers.vector.redis.RedisVectorStoreDriver.username" title="username = field(kw_only=True, default='default', metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.vector.redis.RedisVectorStoreDriver.username)"><code class="language-python">	*str*	The username of the Redis instance.
<a href="redis.mdx#griptape.drivers.vector.redis.RedisVectorStoreDriver.password" title="password = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.vector.redis.RedisVectorStoreDriver.password)"><code class="language-python">	*Optional[str]*	The password of the Redis instance.

continues on next page

Table 306 – continued from previous page

Name	Type	Description
<a href="redis.mdx#griptape.drivers.vector.redis.RedisVectorStoreDriver.index" title="index = field(kw_only=True, metadata={'serializable': True})">class-attribute instance-attribute (griptape.drivers.vector.redis.RedisVectorStoreDriver.index)"><code class="language-python">	*str*	The name of the index to use.

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
@define
class RedisVectorStoreDriver(BaseVectorStoreDriver):
    """A Vector Store Driver for Redis.

    This driver interfaces with a Redis instance and utilizes the Redis hashes and␣
↪RediSearch module to store, retrieve, and query vectors in a structured manner.
    Proper setup of the Redis instance and RediSearch is necessary for the driver to␣
↪function correctly.

    Attributes:
        host: The host of the Redis instance.
        port: The port of the Redis instance.
        db: The database of the Redis instance.
        username: The username of the Redis instance.
        password: The password of the Redis instance.
        index: The name of the index to use.
    """

    host: str = field(kw_only=True, metadata={"serializable": True})
    username: str = field(kw_only=True, default="default", metadata={"serializable":␣
↪False})
    port: int = field(kw_only=True, metadata={"serializable": True})
    db: int = field(kw_only=True, default=0, metadata={"serializable": True})
    password: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": False})
    index: str = field(kw_only=True, metadata={"serializable": True})
    _client: Optional[Redis] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> Redis:
```

(continues on next page)

(continued from previous page)

```
        return import_optional_dependency("redis").Redis(
            host=self.host,
            port=self.port,
            db=self.db,
            username=self.username,
            password=self.password,
            decode_responses=False,
        )

    def upsert_vector(
        self,
        vector: list[float],
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        """Inserts or updates a vector in Redis.

        If a vector with the given vector ID already exists, it is updated; otherwise,
↪ a new vector is inserted.
        Metadata associated with the vector can also be provided.
        """
        vector_id = vector_id or str_to_hash(str(vector))
        key = self._generate_key(vector_id, namespace)
        bytes_vector = json.dumps(vector).encode("utf-8")

        mapping = {}
        mapping["vector"] = np.array(vector, dtype=np.float32).tobytes()
        mapping["vec_string"] = bytes_vector

        if namespace:
            mapping["namespace"] = namespace

        if meta:
            mapping["metadata"] = json.dumps(meta)

        self.client.hset(key, mapping=mapping)

        return vector_id

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        """Retrieves a specific vector entry from Redis based on its identifier and␣
↪optional namespace.

        Returns:
            If the entry is found, it returns an instance of BaseVectorStoreDriver.
↪Entry; otherwise, None is returned.
        """
        key = self._generate_key(vector_id, namespace)
        result = self.client.hgetall(key)
```

(continues on next page)

(continued from previous page)

```
        vector = np.frombuffer(result[b"vector"], dtype=np.float32).tolist()  #␣
↪pyright: ignore[reportIndexIssue] https://github.com/redis/redis-py/issues/2399
        meta = json.loads(result[b"metadata"]) if b"metadata" in result else None  #␣
↪pyright: ignore[reportIndexIssue, reportOperatorIssue]

        return BaseVectorStoreDriver.Entry(id=vector_id, meta=meta, vector=vector,␣
↪namespace=namespace)

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Retrieves all vector entries from Redis that match the optional namespace.

        Returns:
            A list of `BaseVectorStoreDriver.Entry` objects.
        """
        pattern = f"{namespace}:*" if namespace else "*"
        keys = self.client.keys(pattern)

        entries = []
        for key in keys:  # pyright: ignore[reportGeneralTypeIssues] https://github.
↪com/redis/redis-py/issues/2399
            entry = self.load_entry(key.decode("utf-8"), namespace=namespace)
            if entry:
                entries.append(entry)

        return entries

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Performs a nearest neighbor search on Redis to find vectors similar to the␣
↪provided input vector.

        Results can be limited using the count parameter and optionally filtered by a␣
↪namespace.

        Returns:
            A list of BaseVectorStoreDriver.Entry objects, each encapsulating the␣
↪retrieved vector, its similarity score, metadata, and namespace.
        """
        search_query = import_optional_dependency("redis.commands.search.query")

        filter_expression = f"(@namespace:{{{namespace}}})" if namespace else "*"
        query_expression = (
            search_query.Query(f"{filter_expression}=>[KNN {count or 10} @vector
↪$vector as score]")
```

(continues on next page)

(continued from previous page)

```
            .sort_by("score")
            .return_fields("id", "score", "metadata", "vec_string")
            .paging(0, count or 10)
            .dialect(2)
        )

        query_params = {"vector": np.array(vector, dtype=np.float32).tobytes()}

        results = self.client.ft(self.index).search(query_expression, query_params).
↪docs  # pyright: ignore[reportArgumentType, reportAttributeAccessIssue]

        query_results = []
        for document in results:
            metadata = json.loads(document.metadata) if hasattr(document, "metadata")␣
↪else None
            namespace = document.id.split(":")[0] if ":" in document.id else None
            vector_id = document.id.split(":")[1] if ":" in document.id else document.
↪id
            vector_float_list = json.loads(document.vec_string) if include_vectors␣
↪else None
            query_results.append(
                BaseVectorStoreDriver.Entry(
                    id=vector_id,
                    vector=vector_float_list,
                    score=float(document.score),
                    meta=metadata,
                    namespace=namespace,
                ),
            )
        return query_results

    def _generate_key(self, vector_id: str, namespace: Optional[str] = None) -> str:
        """Generates a Redis key using the provided vector ID and optionally a␣
↪namespace."""
        return f"{namespace}:{vector_id}" if namespace else vector_id

    def _get_doc_prefix(self, namespace: Optional[str] = None) -> str:
        """Get the document prefix based on the provided namespace."""
        return f"{namespace}:" if namespace else ""

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `db = field(kw_only=True, default=0, metadata={'serializable': True})` class-attribute instance-attribute
- `host = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `index = field(kw_only=True, metadata={'serializable': True})` class-attribute

instance-attribute

- `password = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `port = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `username = field(kw_only=True, default='default', metadata={'serializable': False})` class-attribute instance-attribute

_generate_key(vector_id, namespace=None)

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def _generate_key(self, vector_id: str, namespace: Optional[str] = None) -> str:
    """Generates a Redis key using the provided vector ID and optionally a namespace."
↪"""
    return f"{namespace}:{vector_id}" if namespace else vector_id
```

_get_doc_prefix(namespace=None)

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def _get_doc_prefix(self, namespace: Optional[str] = None) -> str:
    """Get the document prefix based on the provided namespace."""
    return f"{namespace}:" if namespace else ""
```

client()

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
@lazy_property()
def client(self) -> Redis:
    return import_optional_dependency("redis").Redis(
        host=self.host,
        port=self.port,
        db=self.db,
        username=self.username,
        password=self.password,
        decode_responses=False,
    )
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

load_entries(*, namespace=None)

Retrieves all vector entries from Redis that match the optional namespace.

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	A list of *BaseVectorStoreDriver.Entry* objects.

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Retrieves all vector entries from Redis that match the optional namespace.

    Returns:
        A list of `BaseVectorStoreDriver.Entry` objects.
    """
    pattern = f"{namespace}:*" if namespace else "*"
    keys = self.client.keys(pattern)

    entries = []
    for key in keys:  # pyright: ignore[reportGeneralTypeIssues] https://github.com/
↪redis/redis-py/issues/2399
        entry = self.load_entry(key.decode("utf-8"), namespace=namespace)
        if entry:
            entries.append(entry)

    return entries
```

load_entry(vector_id, *, namespace=None)

Retrieves a specific vector entry from Redis based on its identifier and optional namespace.

Returns

Type	Description
<a href="./#griptape.drivers.vector.BaseVectorStoreDriver.Entry" title="Entry (griptape.drivers.vector.BaseVectorStoreDriver.Entry)"><code class="language-python">C	If the entry is found, it returns an instance of BaseVectorStoreDriver.Entry; otherwise, None is returned.

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    """Retrieves a specific vector entry from Redis based on its identifier and␣
↪optional namespace.

    Returns:
        If the entry is found, it returns an instance of BaseVectorStoreDriver.Entry;␣
↪otherwise, None is returned.
    """
    key = self._generate_key(vector_id, namespace)
    result = self.client.hgetall(key)
    vector = np.frombuffer(result[b"vector"], dtype=np.float32).tolist()  # pyright:␣
↪ignore[reportIndexIssue] https://github.com/redis/redis-py/issues/2399
    meta = json.loads(result[b"metadata"]) if b"metadata" in result else None  #␣
↪pyright: ignore[reportIndexIssue, reportOperatorIssue]

    return BaseVectorStoreDriver.Entry(id=vector_id, meta=meta, vector=vector,␣
↪namespace=namespace)
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs)

Performs a nearest neighbor search on Redis to find vectors similar to the provided input vector.

Results can be limited using the count parameter and optionally filtered by a namespace.

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	A list of BaseVectorStoreDriver.Entry objects, each encapsulating the retrieved vector, its similarity score, metadata, and namespace.

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    """Performs a nearest neighbor search on Redis to find vectors similar to the␣
↪provided input vector.

    Results can be limited using the count parameter and optionally filtered by a␣
↪namespace.

    Returns:
```

(continues on next page)

(continued from previous page)

```
        A list of BaseVectorStoreDriver.Entry objects, each encapsulating the␣
↪retrieved vector, its similarity score, metadata, and namespace.
    """
    search_query = import_optional_dependency("redis.commands.search.query")

    filter_expression = f"(@namespace:{{{namespace}}})" if namespace else "*"
    query_expression = (
        search_query.Query(f"{filter_expression}=>[KNN {count or 10} @vector $vector␣
↪as score]")
        .sort_by("score")
        .return_fields("id", "score", "metadata", "vec_string")
        .paging(0, count or 10)
        .dialect(2)
    )

    query_params = {"vector": np.array(vector, dtype=np.float32).tobytes()}

    results = self.client.ft(self.index).search(query_expression, query_params).docs
↪# pyright: ignore[reportArgumentType, reportAttributeAccessIssue]

    query_results = []
    for document in results:
        metadata = json.loads(document.metadata) if hasattr(document, "metadata")␣
↪else None
        namespace = document.id.split(":")[0] if ":" in document.id else None
        vector_id = document.id.split(":")[1] if ":" in document.id else document.id
        vector_float_list = json.loads(document.vec_string) if include_vectors else␣
↪None
        query_results.append(
            BaseVectorStoreDriver.Entry(
                id=vector_id,
                vector=vector_float_list,
                score=float(document.score),
                meta=metadata,
                namespace=namespace,
            ),
        )
    return query_results
```

upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    """Inserts or updates a vector in Redis.
```

(continues on next page)

(continued from previous page)

```
    If a vector with the given vector ID already exists, it is updated; otherwise, a␣
↪new vector is inserted.
    Metadata associated with the vector can also be provided.
    """
    vector_id = vector_id or str_to_hash(str(vector))
    key = self._generate_key(vector_id, namespace)
    bytes_vector = json.dumps(vector).encode("utf-8")

    mapping = {}
    mapping["vector"] = np.array(vector, dtype=np.float32).tobytes()
    mapping["vec_string"] = bytes_vector

    if namespace:
        mapping["namespace"] = namespace

    if meta:
        mapping["metadata"] = json.dumps(meta)

    self.client.hset(key, mapping=mapping)

    return vector_id
```

redis_vector_store_driver

Bases: *BaseVectorStoreDriver* “)

Attributes

Name	Type	Description
<a href="redis_vector_store_driver.mdx#griptape.drivers.vector.redis_vector_store_driver.RedisVectorStoreDriver.host" title="host = field(kw_only=True, metadata={'serializable': True})">class-attribute instance-attribute (griptape.drivers.vector.redis_vector_store_driver.RedisVectorStoreDriver.host)"><code class="language-python">	*str*	The host of the Redis instance.

continues on next page

Table 310 – continued from previous page

Name	Type	Description
<a href="redis_vector_store_driver.mdx#griptape.drivers.vector.redis_vector_store_driver.RedisVectorStoreDriver.port" title="port = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.redis_vector_store_driver.RedisVectorStoreDriver.port)"><code class="language-python">	*int*	The port of the Redis instance.
<a href="redis_vector_store_driver.mdx#griptape.drivers.vector.redis_vector_store_driver.RedisVectorStoreDriver.db" title="db = field(kw_only=True, default=0, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.redis_vector_store_driver.RedisVectorStoreDriver.db)"><code class="language-python">	*int*	The database of the Redis instance.

continues on next page

Table 310 – continued from previous page

Name	Type	Description
<a href="redis_vector_store_driver.mdx#griptape.drivers.vector.redis_vector_store_driver.RedisVectorStoreDriver.username" title="username = field(kw_only=True, default='default', metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.vector.redis_vector_store_driver.RedisVectorStoreDriver.username)"><code class="language-python">	*str*	The username of the Redis instance.
<a href="redis_vector_store_driver.mdx#griptape.drivers.vector.redis_vector_store_driver.RedisVectorStoreDriver.password" title="password = field(default=None, kw_only=True, metadata={'serializable': False}) class-attribute instance-attribute (griptape.drivers.vector.redis_vector_store_driver.RedisVectorStoreDriver.password)"><code class="language-python">	*Optional[str]*	The password of the Redis instance.

continues on next page

Table 310 – continued from previous page

Name	Type	Description
<a href="redis_vector_store_driver.mdx#griptape.drivers.vector.redis_vector_store_driver.RedisVectorStoreDriver.index" title="index = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.drivers.vector.redis_vector_store_driver.RedisVectorStoreDriver.index)"><code class="language-python">	*str*	The name of the index to use.

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
@define
class RedisVectorStoreDriver(BaseVectorStoreDriver):
    """A Vector Store Driver for Redis.

    This driver interfaces with a Redis instance and utilizes the Redis hashes and␣
↪RediSearch module to store, retrieve, and query vectors in a structured manner.
    Proper setup of the Redis instance and RediSearch is necessary for the driver to␣
↪function correctly.

    Attributes:
        host: The host of the Redis instance.
        port: The port of the Redis instance.
        db: The database of the Redis instance.
        username: The username of the Redis instance.
        password: The password of the Redis instance.
        index: The name of the index to use.
    """

    host: str = field(kw_only=True, metadata={"serializable": True})
    username: str = field(kw_only=True, default="default", metadata={"serializable":␣
↪False})
    port: int = field(kw_only=True, metadata={"serializable": True})
    db: int = field(kw_only=True, default=0, metadata={"serializable": True})
    password: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": False})
    index: str = field(kw_only=True, metadata={"serializable": True})
    _client: Optional[Redis] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})
```

(continues on next page)

(continued from previous page)

```
    @lazy_property()
    def client(self) -> Redis:
        return import_optional_dependency("redis").Redis(
            host=self.host,
            port=self.port,
            db=self.db,
            username=self.username,
            password=self.password,
            decode_responses=False,
        )

    def upsert_vector(
        self,
        vector: list[float],
        vector_id: Optional[str] = None,
        namespace: Optional[str] = None,
        meta: Optional[dict] = None,
        **kwargs,
    ) -> str:
        """Inserts or updates a vector in Redis.

        If a vector with the given vector ID already exists, it is updated; otherwise,
↪ a new vector is inserted.
        Metadata associated with the vector can also be provided.
        """
        vector_id = vector_id or str_to_hash(str(vector))
        key = self._generate_key(vector_id, namespace)
        bytes_vector = json.dumps(vector).encode("utf-8")

        mapping = {}
        mapping["vector"] = np.array(vector, dtype=np.float32).tobytes()
        mapping["vec_string"] = bytes_vector

        if namespace:
            mapping["namespace"] = namespace

        if meta:
            mapping["metadata"] = json.dumps(meta)

        self.client.hset(key, mapping=mapping)

        return vector_id

    def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
        """Retrieves a specific vector entry from Redis based on its identifier and␣
↪optional namespace.

        Returns:
            If the entry is found, it returns an instance of BaseVectorStoreDriver.
↪Entry; otherwise, None is returned.
        """
```

(continues on next page)

(continued from previous page)

```
        key = self._generate_key(vector_id, namespace)
        result = self.client.hgetall(key)
        vector = np.frombuffer(result[b"vector"], dtype=np.float32).tolist()  #␣
↪pyright: ignore[reportIndexIssue] https://github.com/redis/redis-py/issues/2399
        meta = json.loads(result[b"metadata"]) if b"metadata" in result else None  #␣
↪pyright: ignore[reportIndexIssue, reportOperatorIssue]

        return BaseVectorStoreDriver.Entry(id=vector_id, meta=meta, vector=vector,␣
↪namespace=namespace)

    def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
        """Retrieves all vector entries from Redis that match the optional namespace.

        Returns:
            A list of `BaseVectorStoreDriver.Entry` objects.
        """
        pattern = f"{namespace}:*" if namespace else "*"
        keys = self.client.keys(pattern)

        entries = []
        for key in keys:  # pyright: ignore[reportGeneralTypeIssues] https://github.
↪com/redis/redis-py/issues/2399
            entry = self.load_entry(key.decode("utf-8"), namespace=namespace)
            if entry:
                entries.append(entry)

        return entries

    def query_vector(
        self,
        vector: list[float],
        *,
        count: Optional[int] = None,
        namespace: Optional[str] = None,
        include_vectors: bool = False,
        **kwargs,
    ) -> list[BaseVectorStoreDriver.Entry]:
        """Performs a nearest neighbor search on Redis to find vectors similar to the␣
↪provided input vector.

        Results can be limited using the count parameter and optionally filtered by a␣
↪namespace.

        Returns:
            A list of BaseVectorStoreDriver.Entry objects, each encapsulating the␣
↪retrieved vector, its similarity score, metadata, and namespace.
        """
        search_query = import_optional_dependency("redis.commands.search.query")

        filter_expression = f"(@namespace:{{{namespace}}})" if namespace else "*"
        query_expression = (
```

(continues on next page)

(continued from previous page)

```
            search_query.Query(f"{filter_expression}=>[KNN {count or 10} @vector
↪$vector as score]")
            .sort_by("score")
            .return_fields("id", "score", "metadata", "vec_string")
            .paging(0, count or 10)
            .dialect(2)
        )

        query_params = {"vector": np.array(vector, dtype=np.float32).tobytes()}

        results = self.client.ft(self.index).search(query_expression, query_params).
↪docs  # pyright: ignore[reportArgumentType, reportAttributeAccessIssue]

        query_results = []
        for document in results:
            metadata = json.loads(document.metadata) if hasattr(document, "metadata")␣
↪else None
            namespace = document.id.split(":")[0] if ":" in document.id else None
            vector_id = document.id.split(":")[1] if ":" in document.id else document.
↪id
            vector_float_list = json.loads(document.vec_string) if include_vectors␣
↪else None
            query_results.append(
                BaseVectorStoreDriver.Entry(
                    id=vector_id,
                    vector=vector_float_list,
                    score=float(document.score),
                    meta=metadata,
                    namespace=namespace,
                ),
            )
        return query_results

    def _generate_key(self, vector_id: str, namespace: Optional[str] = None) -> str:
        """Generates a Redis key using the provided vector ID and optionally a␣
↪namespace."""
        return f"{namespace}:{vector_id}" if namespace else vector_id

    def _get_doc_prefix(self, namespace: Optional[str] = None) -> str:
        """Get the document prefix based on the provided namespace."""
        return f"{namespace}:" if namespace else ""

    def delete_vector(self, vector_id: str) -> NoReturn:
        raise NotImplementedError(f"{self.__class__.__name__} does not support␣
↪deletion.")
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `db = field(kw_only=True, default=0, metadata={'serializable': True})` class-attribute instance-attribute
- `host = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

- `index = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `password = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `port = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `username = field(kw_only=True, default='default', metadata={'serializable': False})` class-attribute instance-attribute

_generate_key(vector_id, namespace=None)

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def _generate_key(self, vector_id: str, namespace: Optional[str] = None) -> str:
    """Generates a Redis key using the provided vector ID and optionally a namespace."
↪"""
    return f"{namespace}:{vector_id}" if namespace else vector_id
```

_get_doc_prefix(namespace=None)

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def _get_doc_prefix(self, namespace: Optional[str] = None) -> str:
    """Get the document prefix based on the provided namespace."""
    return f"{namespace}:" if namespace else ""
```

client()

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
@lazy_property()
def client(self) -> Redis:
    return import_optional_dependency("redis").Redis(
        host=self.host,
        port=self.port,
        db=self.db,
        username=self.username,
        password=self.password,
        decode_responses=False,
    )
```

delete_vector(vector_id)

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def delete_vector(self, vector_id: str) -> NoReturn:
    raise NotImplementedError(f"{self.__class__.__name__} does not support deletion.")
```

load_entries(*, namespace=None)

Retrieves all vector entries from Redis that match the optional namespace.

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	A list of *BaseVectorStoreDriver.Entry* objects.

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def load_entries(self, *, namespace: Optional[str] = None) ->␣
↪list[BaseVectorStoreDriver.Entry]:
    """Retrieves all vector entries from Redis that match the optional namespace.

    Returns:
        A list of `BaseVectorStoreDriver.Entry` objects.
    """
    pattern = f"{namespace}:*" if namespace else "*"
    keys = self.client.keys(pattern)

    entries = []
    for key in keys:  # pyright: ignore[reportGeneralTypeIssues] https://github.com/
↪redis/redis-py/issues/2399
        entry = self.load_entry(key.decode("utf-8"), namespace=namespace)
        if entry:
            entries.append(entry)

    return entries
```

load_entry(vector_id, *, namespace=None)

Retrieves a specific vector entry from Redis based on its identifier and optional namespace.

Returns

Type	Description
<a href="./#griptape.drivers.vector.BaseVectorStoreDriver.Entry" title="Entry (griptape.drivers.vector.BaseVectorStoreDriver.Entry)"><code class="language-python">C	If the entry is found, it returns an instance of BaseVectorStoreDriver.Entry; otherwise, None is returned.

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def load_entry(self, vector_id: str, *, namespace: Optional[str] = None) ->␣
↪Optional[BaseVectorStoreDriver.Entry]:
    """Retrieves a specific vector entry from Redis based on its identifier and␣
↪optional namespace.

    Returns:
        If the entry is found, it returns an instance of BaseVectorStoreDriver.Entry;␣
↪otherwise, None is returned.
    """
    key = self._generate_key(vector_id, namespace)
    result = self.client.hgetall(key)
    vector = np.frombuffer(result[b"vector"], dtype=np.float32).tolist()  # pyright:␣
↪ignore[reportIndexIssue] https://github.com/redis/redis-py/issues/2399
    meta = json.loads(result[b"metadata"]) if b"metadata" in result else None  #␣
↪pyright: ignore[reportIndexIssue, reportOperatorIssue]

    return BaseVectorStoreDriver.Entry(id=vector_id, meta=meta, vector=vector,␣
↪namespace=namespace)
```

query_vector(vector, *, count=None, namespace=None, include_vectors=False, **kwargs)

Performs a nearest neighbor search on Redis to find vectors similar to the provided input vector.

Results can be limited using the count parameter and optionally filtered by a namespace.

Returns

Type	Description
<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">li	A list of BaseVectorStoreDriver.Entry objects, each encapsulating the retrieved vector, its similarity score, metadata, and namespace.

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def query_vector(
    self,
    vector: list[float],
    *,
    count: Optional[int] = None,
    namespace: Optional[str] = None,
    include_vectors: bool = False,
    **kwargs,
) -> list[BaseVectorStoreDriver.Entry]:
    """Performs a nearest neighbor search on Redis to find vectors similar to the␣
↪provided input vector.

    Results can be limited using the count parameter and optionally filtered by a␣
↪namespace.

    Returns:
```

(continues on next page)

(continued from previous page)

```
        A list of BaseVectorStoreDriver.Entry objects, each encapsulating the␣
↪retrieved vector, its similarity score, metadata, and namespace.
    """
    search_query = import_optional_dependency("redis.commands.search.query")

    filter_expression = f"(@namespace:{{{namespace}}})" if namespace else "*"
    query_expression = (
        search_query.Query(f"{filter_expression}=>[KNN {count or 10} @vector $vector␣
↪as score]")
        .sort_by("score")
        .return_fields("id", "score", "metadata", "vec_string")
        .paging(0, count or 10)
        .dialect(2)
    )

    query_params = {"vector": np.array(vector, dtype=np.float32).tobytes()}

    results = self.client.ft(self.index).search(query_expression, query_params).docs
↪# pyright: ignore[reportArgumentType, reportAttributeAccessIssue]

    query_results = []
    for document in results:
        metadata = json.loads(document.metadata) if hasattr(document, "metadata")␣
↪else None
        namespace = document.id.split(":")[0] if ":" in document.id else None
        vector_id = document.id.split(":")[1] if ":" in document.id else document.id
        vector_float_list = json.loads(document.vec_string) if include_vectors else␣
↪None
        query_results.append(
            BaseVectorStoreDriver.Entry(
                id=vector_id,
                vector=vector_float_list,
                score=float(document.score),
                meta=metadata,
                namespace=namespace,
            ),
        )
    return query_results
```

upsert_vector(vector, vector_id=None, namespace=None, meta=None, **kwargs)

Source Code in griptape/drivers/vector/redis_vector_store_driver.py

```
def upsert_vector(
    self,
    vector: list[float],
    vector_id: Optional[str] = None,
    namespace: Optional[str] = None,
    meta: Optional[dict] = None,
    **kwargs,
) -> str:
    """Inserts or updates a vector in Redis.
```

(continues on next page)

(continued from previous page)

```
    If a vector with the given vector ID already exists, it is updated; otherwise, a␣
↪new vector is inserted.
    Metadata associated with the vector can also be provided.
    """
    vector_id = vector_id or str_to_hash(str(vector))
    key = self._generate_key(vector_id, namespace)
    bytes_vector = json.dumps(vector).encode("utf-8")

    mapping = {}
    mapping["vector"] = np.array(vector, dtype=np.float32).tobytes()
    mapping["vec_string"] = bytes_vector

    if namespace:
        mapping["namespace"] = namespace

    if meta:
        mapping["metadata"] = json.dumps(meta)

    self.client.hset(key, mapping=mapping)

    return vector_id
```

web_scraper

- `__all__ = ['BaseWebScraperDriver']` module-attribute

Bases:

ABC

Source Code in griptape/drivers/web_scraper/base_web_scraper_driver.py

```
class BaseWebScraperDriver(ABC):
    def scrape_url(self, url: str) -> TextArtifact:
        source = self.fetch_url(url)

        return self.extract_page(source)

    @abstractmethod
    def fetch_url(self, url: str) -> str: ...

    @abstractmethod
    def extract_page(self, page: str) -> TextArtifact: ...
```

extract_page(page)abstractmethod

Source Code in griptape/drivers/web_scraper/base_web_scraper_driver.py

```
@abstractmethod
def extract_page(self, page: str) -> TextArtifact: ...
```

fetch_url(url)abstractmethod

Source Code in griptape/drivers/web_scraper/base_web_scraper_driver.py

```
@abstractmethod
def fetch_url(self, url: str) -> str: ...
```

scrape_url(url)

Source Code in griptape/drivers/web_scraper/base_web_scraper_driver.py

```
def scrape_url(self, url: str) -> TextArtifact:
    source = self.fetch_url(url)

    return self.extract_page(source)
```

base_web_scraper_driver

Bases:

ABC

Source Code in griptape/drivers/web_scraper/base_web_scraper_driver.py

```
class BaseWebScraperDriver(ABC):
    def scrape_url(self, url: str) -> TextArtifact:
        source = self.fetch_url(url)

        return self.extract_page(source)

    @abstractmethod
    def fetch_url(self, url: str) -> str: ...

    @abstractmethod
    def extract_page(self, page: str) -> TextArtifact: ...
```

extract_page(page)abstractmethod

Source Code in griptape/drivers/web_scraper/base_web_scraper_driver.py

```
@abstractmethod
def extract_page(self, page: str) -> TextArtifact: ...
```

fetch_url(url)abstractmethod

Source Code in griptape/drivers/web_scraper/base_web_scraper_driver.py

```
@abstractmethod
def fetch_url(self, url: str) -> str: ...
```

scrape_url(url)

Source Code in griptape/drivers/web_scraper/base_web_scraper_driver.py

```
def scrape_url(self, url: str) -> TextArtifact:
    source = self.fetch_url(url)

    return self.extract_page(source)
```

markdownify

- `__all__ = ['MarkdownifyWebScraperDriver']` module-attribute

Bases: *BaseWebScraperDriver* “)

Attributes

Name	Type	Description
<code class="language-python">	*bool*	If *True*, the driver will include link urls in the markdown output.

continues on next page

Table 314 – continued from previous page

Name	Type	Description
<code class="language-python">	<code class="language-python">list[str]</code>	Optionally provide custom tags to exclude from the scraped content.
<code class="language-python">	<code class="language-python">list[str]</code>	Optionally provide custom classes to exclude from the scraped content.

continues on next page

Table 314 – continued from previous page

Name	Type	Description
<code class="language-python">	<code class="language-python">list[str]</code>	Optionally provide custom ids to exclude from the scraped content.
<a href="markdownify.mdx#griptape.drivers.web_scraper.downify.MarkdownifyWebScraperDriver.timeout" title="timeout = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.drivers.web_scraper.downify.MarkdownifyWebScraperDriver.timeout)"><code class="language-python">	*Optional[int]*	Optionally provide a timeout in milliseconds for the page to continue loading after the browser has emitted the "load" event.

Source Code in griptape/drivers/web_scraper/markdownify_web_scraper_driver.py

```
@define
class MarkdownifyWebScraperDriver(BaseWebScraperDriver):
    """Driver to scrape a webpage and return the content in markdown format.

    As a prerequisite to using MarkdownifyWebScraperDriver, you need to install the␣
↪browsers used by
    playwright. You can do this by running: `poetry run playwright install`.
    For more details about playwright, see https://playwright.dev/python/docs/library.

    Attributes:
        include_links: If `True`, the driver will include link urls in the markdown␣
↪output.
        exclude_tags: Optionally provide custom tags to exclude from the scraped␣
↪content.
        exclude_classes: Optionally provide custom classes to exclude from the␣
```

(continues on next page)

(continued from previous page)

```
↪scraped content.
        exclude_ids: Optionally provide custom ids to exclude from the scraped␣
↪content.
        timeout: Optionally provide a timeout in milliseconds for the page to␣
↪continue loading after
            the browser has emitted the "load" event.
    """

    DEFAULT_EXCLUDE_TAGS = ["script", "style", "head", "audio", "img", "picture",
↪"source", "video"]

    include_links: bool = field(default=True, kw_only=True)
    exclude_tags: list[str] = field(
        default=Factory(lambda self: self.DEFAULT_EXCLUDE_TAGS, takes_self=True),
        kw_only=True,
    )
    exclude_classes: list[str] = field(default=Factory(list), kw_only=True)
    exclude_ids: list[str] = field(default=Factory(list), kw_only=True)
    timeout: Optional[int] = field(default=None, kw_only=True)

    def fetch_url(self, url: str) -> str:
        sync_playwright = import_optional_dependency("playwright.sync_api").sync_
↪playwright

        with sync_playwright() as p, p.chromium.launch(headless=True) as browser:
            page = browser.new_page()

            def skip_loading_images(route: Any) -> Any:
                if route.request.resource_type == "image":
                    return route.abort()
                route.continue_()
                return None

            page.route("**/*", skip_loading_images)

            page.goto(url)

            # Some websites require a delay before the content is fully loaded
            # even after the browser has emitted "load" event.
            if self.timeout:
                page.wait_for_timeout(self.timeout)

            content = page.content()

            if not content:
                raise Exception("cant access URL")

            return content

    def extract_page(self, page: str) -> TextArtifact:
        bs4 = import_optional_dependency("bs4")
        markdownify = import_optional_dependency("markdownify")
```

(continues on next page)

(continued from previous page)

```
        include_links = self.include_links

        # Custom MarkdownConverter to optionally linked urls. If include_links is␣
↪False only
        # the text of the link is returned.
        class OptionalLinksMarkdownConverter(markdownify.MarkdownConverter):
            def convert_a(self, el: Any, text: str, parent_tags: Any) -> str:
                if include_links:
                    return super().convert_a(el, text, parent_tags)
                return text

        soup = bs4.BeautifulSoup(page, "html.parser")

        # Remove unwanted elements
        exclude_selector = ",".join(
            self.exclude_tags + [f".{c}" for c in self.exclude_classes] + [f"#{i}"␣
↪for i in self.exclude_ids],
        )
        if exclude_selector:
            for s in soup.select(exclude_selector):
                s.extract()

        text = OptionalLinksMarkdownConverter().convert_soup(soup)

        # Remove leading and trailing whitespace from the entire text
        text = text.strip()

        # Remove trailing whitespace from each line
        text = re.sub(r"[ \t]+$", "", text, flags=re.MULTILINE)

        # Indent using 2 spaces instead of tabs
        text = re.sub(r"(\n?\s*?)\t", r"  ", text)

        # Remove triple+ newlines (keep double newlines for paragraphs)
        text = re.sub(r"\n\n+", "\n\n", text)

        return TextArtifact(text)
```

- `DEFAULT_EXCLUDE_TAGS = ['script', 'style', 'head', 'audio', 'img', 'picture', 'source', 'video']` class-attribute instance-attribute
- `exclude_classes = field(default=Factory(list), kw_only=True)` class-attribute instance-attribute
- `exclude_ids = field(default=Factory(list), kw_only=True)` class-attribute instance-attribute
- `exclude_tags = field(default=Factory(lambda self: self.DEFAULT_EXCLUDE_TAGS, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `include_links = field(default=True, kw_only=True)` class-attribute instance-attribute
- `timeout = field(default=None, kw_only=True)` class-attribute instance-attribute

extract_page(page)

Source Code in griptape/drivers/web_scraper/markdownify_web_scraper_driver.py

```
def extract_page(self, page: str) -> TextArtifact:
    bs4 = import_optional_dependency("bs4")
    markdownify = import_optional_dependency("markdownify")
    include_links = self.include_links

    # Custom MarkdownConverter to optionally linked urls. If include_links is False␣
↪only
    # the text of the link is returned.
    class OptionalLinksMarkdownConverter(markdownify.MarkdownConverter):
        def convert_a(self, el: Any, text: str, parent_tags: Any) -> str:
            if include_links:
                return super().convert_a(el, text, parent_tags)
            return text

    soup = bs4.BeautifulSoup(page, "html.parser")

    # Remove unwanted elements
    exclude_selector = ",".join(
        self.exclude_tags + [f".{c}" for c in self.exclude_classes] + [f"#{i}" for i␣
↪in self.exclude_ids],
    )
    if exclude_selector:
        for s in soup.select(exclude_selector):
            s.extract()

    text = OptionalLinksMarkdownConverter().convert_soup(soup)

    # Remove leading and trailing whitespace from the entire text
    text = text.strip()

    # Remove trailing whitespace from each line
    text = re.sub(r"[ \t]+$", "", text, flags=re.MULTILINE)

    # Indent using 2 spaces instead of tabs
    text = re.sub(r"(\n?\s*?)\t", r"  ", text)

    # Remove triple+ newlines (keep double newlines for paragraphs)
    text = re.sub(r"\n\n+", "\n\n", text)

    return TextArtifact(text)
```

fetch_url(url)

Source Code in griptape/drivers/web_scraper/markdownify_web_scraper_driver.py

```
def fetch_url(self, url: str) -> str:
    sync_playwright = import_optional_dependency("playwright.sync_api").sync_
↪playwright
```

(continues on next page)

(continued from previous page)

```
    with sync_playwright() as p, p.chromium.launch(headless=True) as browser:
        page = browser.new_page()

        def skip_loading_images(route: Any) -> Any:
            if route.request.resource_type == "image":
                return route.abort()
            route.continue_()
            return None

        page.route("**/*", skip_loading_images)

        page.goto(url)

        # Some websites require a delay before the content is fully loaded
        # even after the browser has emitted "load" event.
        if self.timeout:
            page.wait_for_timeout(self.timeout)

        content = page.content()

        if not content:
            raise Exception("cant access URL")

        return content
```

markdownify_web_scraper_driver

Bases: *BaseWebScraperDriver* “)

Attributes

Name	Type	Description
<code class="language-python">	*bool*	If *True*, the driver will include link urls in the markdown output.
<code class="language-python">	<code class="language-python">list[str]</code>	Optionally provide custom tags to exclude from the scraped content.

continues on next page

Table 315 – continued from previous page

Name	Type	Description
<code class="language-python">	<code class="language-python">list[str]</code>	Optionally provide custom classes to exclude from the scraped content.
<code class="language-python">	<code class="language-python">list[str]</code>	Optionally provide custom ids to exclude from the scraped content.

continues on next page

Table 315 – continued from previous page

Name	Type	Description
<a href="markdownify_web_scraper_driver.griptape.drivers.web_scraper.markdownify_web_scraper_driver.MarkdownifyWebScraperDriver.timeout" title="timeout = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.drivers.web_scraper.markdownify_web_scraper_driver.MarkdownifyWebScraperDriver.timeout)"><code class="language-python">	*Optional[int]*	Optionally provide a timeout in milliseconds for the page to continue loading after the browser has emitted the "load" event.

Source Code in griptape/drivers/web_scraper/markdownify_web_scraper_driver.py

```
@define
class MarkdownifyWebScraperDriver(BaseWebScraperDriver):
    """Driver to scrape a webpage and return the content in markdown format.

    As a prerequisite to using MarkdownifyWebScraperDriver, you need to install the␣
↪browsers used by
    playwright. You can do this by running: `poetry run playwright install`.
    For more details about playwright, see https://playwright.dev/python/docs/library.

    Attributes:
        include_links: If `True`, the driver will include link urls in the markdown␣
↪output.
        exclude_tags: Optionally provide custom tags to exclude from the scraped␣
↪content.
        exclude_classes: Optionally provide custom classes to exclude from the␣
↪scraped content.
        exclude_ids: Optionally provide custom ids to exclude from the scraped␣
↪content.
        timeout: Optionally provide a timeout in milliseconds for the page to␣
↪continue loading after
            the browser has emitted the "load" event.
    """

    DEFAULT_EXCLUDE_TAGS = ["script", "style", "head", "audio", "img", "picture",
↪"source", "video"]

    include_links: bool = field(default=True, kw_only=True)
    exclude_tags: list[str] = field(
        default=Factory(lambda self: self.DEFAULT_EXCLUDE_TAGS, takes_self=True),
```

(continues on next page)

(continued from previous page)

```
        kw_only=True,
    )
    exclude_classes: list[str] = field(default=Factory(list), kw_only=True)
    exclude_ids: list[str] = field(default=Factory(list), kw_only=True)
    timeout: Optional[int] = field(default=None, kw_only=True)

    def fetch_url(self, url: str) -> str:
        sync_playwright = import_optional_dependency("playwright.sync_api").sync_
↪playwright

        with sync_playwright() as p, p.chromium.launch(headless=True) as browser:
            page = browser.new_page()

            def skip_loading_images(route: Any) -> Any:
                if route.request.resource_type == "image":
                    return route.abort()
                route.continue_()
                return None

            page.route("**/*", skip_loading_images)

            page.goto(url)

            # Some websites require a delay before the content is fully loaded
            # even after the browser has emitted "load" event.
            if self.timeout:
                page.wait_for_timeout(self.timeout)

            content = page.content()

            if not content:
                raise Exception("cant access URL")

            return content

    def extract_page(self, page: str) -> TextArtifact:
        bs4 = import_optional_dependency("bs4")
        markdownify = import_optional_dependency("markdownify")
        include_links = self.include_links

        # Custom MarkdownConverter to optionally linked urls. If include_links is␣
↪False only
        # the text of the link is returned.
        class OptionalLinksMarkdownConverter(markdownify.MarkdownConverter):
            def convert_a(self, el: Any, text: str, parent_tags: Any) -> str:
                if include_links:
                    return super().convert_a(el, text, parent_tags)
                return text

        soup = bs4.BeautifulSoup(page, "html.parser")

        # Remove unwanted elements
```

(continues on next page)

(continued from previous page)

```
        exclude_selector = ",".join(
            self.exclude_tags + [f".{c}" for c in self.exclude_classes] + [f"#{i}"␣
↪for i in self.exclude_ids],
        )
        if exclude_selector:
            for s in soup.select(exclude_selector):
                s.extract()

        text = OptionalLinksMarkdownConverter().convert_soup(soup)

        # Remove leading and trailing whitespace from the entire text
        text = text.strip()

        # Remove trailing whitespace from each line
        text = re.sub(r"[ \t]+$", "", text, flags=re.MULTILINE)

        # Indent using 2 spaces instead of tabs
        text = re.sub(r"(\n?\s*?)\t", r"  ", text)

        # Remove triple+ newlines (keep double newlines for paragraphs)
        text = re.sub(r"\n\n+", "\n\n", text)

        return TextArtifact(text)
```

- `DEFAULT_EXCLUDE_TAGS = ['script', 'style', 'head', 'audio', 'img', 'picture', 'source', 'video']` class-attribute instance-attribute
- `exclude_classes = field(default=Factory(list), kw_only=True)` class-attribute instance-attribute
- `exclude_ids = field(default=Factory(list), kw_only=True)` class-attribute instance-attribute
- `exclude_tags = field(default=Factory(lambda self: self.DEFAULT_EXCLUDE_TAGS, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `include_links = field(default=True, kw_only=True)` class-attribute instance-attribute
- `timeout = field(default=None, kw_only=True)` class-attribute instance-attribute

extract_page(page)

Source Code in griptape/drivers/web_scraper/markdownify_web_scraper_driver.py

```
def extract_page(self, page: str) -> TextArtifact:
    bs4 = import_optional_dependency("bs4")
    markdownify = import_optional_dependency("markdownify")
    include_links = self.include_links

    # Custom MarkdownConverter to optionally linked urls. If include_links is False␣
↪only
    # the text of the link is returned.
    class OptionalLinksMarkdownConverter(markdownify.MarkdownConverter):
        def convert_a(self, el: Any, text: str, parent_tags: Any) -> str:
            if include_links:
                return super().convert_a(el, text, parent_tags)
```

(continues on next page)

(continued from previous page)

```
            return text

    soup = bs4.BeautifulSoup(page, "html.parser")

    # Remove unwanted elements
    exclude_selector = ",".join(
        self.exclude_tags + [f".{c}" for c in self.exclude_classes] + [f"#{i}" for i␣
↪in self.exclude_ids],
    )
    if exclude_selector:
        for s in soup.select(exclude_selector):
            s.extract()

    text = OptionalLinksMarkdownConverter().convert_soup(soup)

    # Remove leading and trailing whitespace from the entire text
    text = text.strip()

    # Remove trailing whitespace from each line
    text = re.sub(r"[ \t]+$", "", text, flags=re.MULTILINE)

    # Indent using 2 spaces instead of tabs
    text = re.sub(r"(\n?\s*?)\t", r"  ", text)

    # Remove triple+ newlines (keep double newlines for paragraphs)
    text = re.sub(r"\n\n+", "\n\n", text)

    return TextArtifact(text)
```

fetch_url(url)

Source Code in griptape/drivers/web_scraper/markdownify_web_scraper_driver.py

```
def fetch_url(self, url: str) -> str:
    sync_playwright = import_optional_dependency("playwright.sync_api").sync_
↪playwright

    with sync_playwright() as p, p.chromium.launch(headless=True) as browser:
        page = browser.new_page()

        def skip_loading_images(route: Any) -> Any:
            if route.request.resource_type == "image":
                return route.abort()
            route.continue_()
            return None

        page.route("**/*", skip_loading_images)

        page.goto(url)

        # Some websites require a delay before the content is fully loaded
        # even after the browser has emitted "load" event.
```

(continues on next page)

(continued from previous page)

```
        if self.timeout:
            page.wait_for_timeout(self.timeout)

        content = page.content()

        if not content:
            raise Exception("cant access URL")

        return content
```

proxy

- `__all__ = ['ProxyWebScraperDriver']` module-attribute

Bases: *BaseWebScraperDriver* “)

Source Code in griptape/drivers/web_scraper/proxy_web_scraper_driver.py

```
@define
class ProxyWebScraperDriver(BaseWebScraperDriver):
    proxies: dict = field(kw_only=True, metadata={"serializable": False})
    params: dict = field(default=Factory(dict), kw_only=True, metadata={"serializable
↪": True})

    def fetch_url(self, url: str) -> str:
        response = requests.get(url, proxies=self.proxies, **self.params)

        return response.text

    def extract_page(self, page: str) -> TextArtifact:
        return TextArtifact(page)
```

- `params = field(default=Factory(dict), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `proxies = field(kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute

extract_page(page)

Source Code in griptape/drivers/web_scraper/proxy_web_scraper_driver.py

```
def extract_page(self, page: str) -> TextArtifact:
    return TextArtifact(page)
```

fetch_url(url)

Source Code in griptape/drivers/web_scraper/proxy_web_scraper_driver.py

```
def fetch_url(self, url: str) -> str:
    response = requests.get(url, proxies=self.proxies, **self.params)

    return response.text
```

proxy_web_scraper_driver

Bases: *BaseWebScraperDriver* “)

Source Code in griptape/drivers/web_scraper/proxy_web_scraper_driver.py

```
@define
class ProxyWebScraperDriver(BaseWebScraperDriver):
    proxies: dict = field(kw_only=True, metadata={"serializable": False})
    params: dict = field(default=Factory(dict), kw_only=True, metadata={"serializable
↪": True})

    def fetch_url(self, url: str) -> str:
        response = requests.get(url, proxies=self.proxies, **self.params)

        return response.text

    def extract_page(self, page: str) -> TextArtifact:
        return TextArtifact(page)
```

- `params = field(default=Factory(dict), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `proxies = field(kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute

extract_page(page)

Source Code in griptape/drivers/web_scraper/proxy_web_scraper_driver.py

```
def extract_page(self, page: str) -> TextArtifact:
    return TextArtifact(page)
```

fetch_url(url)

Source Code in griptape/drivers/web_scraper/proxy_web_scraper_driver.py

```
def fetch_url(self, url: str) -> str:
    response = requests.get(url, proxies=self.proxies, **self.params)

    return response.text
```

trafilatura

- `__all__ = ['TrafilaturaWebScraperDriver']` module-attribute

Bases: *BaseWebScraperDriver* “)

Source Code in griptape/drivers/web_scraper/trafilatura_web_scraper_driver.py

```
@define
class TrafilaturaWebScraperDriver(BaseWebScraperDriver):
    include_links: bool = field(default=True, kw_only=True)
    no_ssl: bool = field(default=False, kw_only=True)
```

(continues on next page)

(continued from previous page)

```
    def fetch_url(self, url: str) -> str:
        trafilatura = import_optional_dependency("trafilatura")
        use_config = trafilatura.settings.use_config

        config = use_config()
        page = trafilatura.fetch_url(url, no_ssl=self.no_ssl)

        # This disables signal, so that trafilatura can work on any thread:
        # More info: https://trafilatura.readthedocs.io/usage-python.html#disabling-
↪signal
        config.set("DEFAULT", "EXTRACTION_TIMEOUT", "0")

        # Disable error logging in trafilatura as it sometimes logs errors from lxml,␣
↪even though
        # the end result of page parsing is successful.
        logging.getLogger("trafilatura").setLevel(logging.FATAL)

        if page is None:
            raise Exception("cant access URL")

        return page

    def extract_page(self, page: str) -> TextArtifact:
        trafilatura = import_optional_dependency("trafilatura")
        use_config = trafilatura.settings.use_config

        config = use_config()

        extracted_page = trafilatura.extract(
            page,
            include_links=self.include_links,
            output_format="json",
            config=config,
        )

        if not extracted_page:
            raise Exception("cant extract page")

        text = json.loads(extracted_page).get("text")

        return TextArtifact(text)
```

- `include_links = field(default=True, kw_only=True)` class-attribute instance-attribute
- `no_ssl = field(default=False, kw_only=True)` class-attribute instance-attribute

extract_page(page)

Source Code in griptape/drivers/web_scraper/trafilatura_web_scraper_driver.py

```
def extract_page(self, page: str) -> TextArtifact:
    trafilatura = import_optional_dependency("trafilatura")
    use_config = trafilatura.settings.use_config
```

(continues on next page)

(continued from previous page)

```
    config = use_config()

    extracted_page = trafilatura.extract(
        page,
        include_links=self.include_links,
        output_format="json",
        config=config,
    )

    if not extracted_page:
        raise Exception("cant extract page")

    text = json.loads(extracted_page).get("text")

    return TextArtifact(text)
```

fetch_url(url)

Source Code in griptape/drivers/web_scraper/trafilatura_web_scraper_driver.py

```
def fetch_url(self, url: str) -> str:
    trafilatura = import_optional_dependency("trafilatura")
    use_config = trafilatura.settings.use_config

    config = use_config()
    page = trafilatura.fetch_url(url, no_ssl=self.no_ssl)

    # This disables signal, so that trafilatura can work on any thread:
    # More info: https://trafilatura.readthedocs.io/usage-python.html#disabling-signal
    config.set("DEFAULT", "EXTRACTION_TIMEOUT", "0")

    # Disable error logging in trafilatura as it sometimes logs errors from lxml,␣
↪even though
    # the end result of page parsing is successful.
    logging.getLogger("trafilatura").setLevel(logging.FATAL)

    if page is None:
        raise Exception("cant access URL")

    return page
```

trafilatura_web_scraper_driver

Bases: *BaseWebScraperDriver* “)

Source Code in griptape/drivers/web_scraper/trafilatura_web_scraper_driver.py

```
@define
class TrafilaturaWebScraperDriver(BaseWebScraperDriver):
    include_links: bool = field(default=True, kw_only=True)
    no_ssl: bool = field(default=False, kw_only=True)
```

(continues on next page)

(continued from previous page)

```
    def fetch_url(self, url: str) -> str:
        trafilatura = import_optional_dependency("trafilatura")
        use_config = trafilatura.settings.use_config

        config = use_config()
        page = trafilatura.fetch_url(url, no_ssl=self.no_ssl)

        # This disables signal, so that trafilatura can work on any thread:
        # More info: https://trafilatura.readthedocs.io/usage-python.html#disabling-
↪signal
        config.set("DEFAULT", "EXTRACTION_TIMEOUT", "0")

        # Disable error logging in trafilatura as it sometimes logs errors from lxml,␣
↪even though
        # the end result of page parsing is successful.
        logging.getLogger("trafilatura").setLevel(logging.FATAL)

        if page is None:
            raise Exception("cant access URL")

        return page

    def extract_page(self, page: str) -> TextArtifact:
        trafilatura = import_optional_dependency("trafilatura")
        use_config = trafilatura.settings.use_config

        config = use_config()

        extracted_page = trafilatura.extract(
            page,
            include_links=self.include_links,
            output_format="json",
            config=config,
        )

        if not extracted_page:
            raise Exception("cant extract page")

        text = json.loads(extracted_page).get("text")

        return TextArtifact(text)
```

- `include_links = field(default=True, kw_only=True)` class-attribute instance-attribute
- `no_ssl = field(default=False, kw_only=True)` class-attribute instance-attribute

extract_page(page)

Source Code in griptape/drivers/web_scraper/trafilatura_web_scraper_driver.py

```
def extract_page(self, page: str) -> TextArtifact:
    trafilatura = import_optional_dependency("trafilatura")
```

(continues on next page)

(continued from previous page)

```
    use_config = trafilatura.settings.use_config

    config = use_config()

    extracted_page = trafilatura.extract(
        page,
        include_links=self.include_links,
        output_format="json",
        config=config,
    )

    if not extracted_page:
        raise Exception("cant extract page")

    text = json.loads(extracted_page).get("text")

    return TextArtifact(text)
```

fetch_url(url)

Source Code in griptape/drivers/web_scraper/trafilatura_web_scraper_driver.py

```
def fetch_url(self, url: str) -> str:
    trafilatura = import_optional_dependency("trafilatura")
    use_config = trafilatura.settings.use_config

    config = use_config()
    page = trafilatura.fetch_url(url, no_ssl=self.no_ssl)

    # This disables signal, so that trafilatura can work on any thread:
    # More info: https://trafilatura.readthedocs.io/usage-python.html#disabling-signal
    config.set("DEFAULT", "EXTRACTION_TIMEOUT", "0")

    # Disable error logging in trafilatura as it sometimes logs errors from lxml,␣
↪even though
    # the end result of page parsing is successful.
    logging.getLogger("trafilatura").setLevel(logging.FATAL)

    if page is None:
        raise Exception("cant access URL")

    return page
```

web_search

- `__all__ = ['BaseWebSearchDriver']` module-attribute

Bases:

ABC

Source Code in griptape/drivers/web_search/base_web_search_driver.py

```
@define
class BaseWebSearchDriver(ABC):
    results_count: int = field(default=5, kw_only=True)

    @abstractmethod
    def search(self, query: str, **kwargs) -> ListArtifact: ...
```

- `results_count = field(default=5, kw_only=True)` class-attribute instance-attribute

search(query, **kwargs)abstractmethod

Source Code in griptape/drivers/web_search/base_web_search_driver.py

```
@abstractmethod
def search(self, query: str, **kwargs) -> ListArtifact: ...
```

base_web_search_driver

Bases:

ABC

Source Code in griptape/drivers/web_search/base_web_search_driver.py

```
@define
class BaseWebSearchDriver(ABC):
    results_count: int = field(default=5, kw_only=True)

    @abstractmethod
    def search(self, query: str, **kwargs) -> ListArtifact: ...
```

- `results_count = field(default=5, kw_only=True)` class-attribute instance-attribute

search(query, **kwargs)abstractmethod

Source Code in griptape/drivers/web_search/base_web_search_driver.py

```
@abstractmethod
def search(self, query: str, **kwargs) -> ListArtifact: ...
```

duck_duck_go

- `__all__ = ['DuckDuckGoWebSearchDriver']` module-attribute

Bases: *BaseWebSearchDriver* “)

Source Code in griptape/drivers/web_search/duck_duck_go_web_search_driver.py

```
@define
class DuckDuckGoWebSearchDriver(BaseWebSearchDriver):
    language: str = field(default="en", kw_only=True)
    country: str = field(default="us", kw_only=True)
    _client: Optional[DDGS] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})
```

(continues on next page)

(continued from previous page)

```
    @lazy_property()
    def client(self) -> DDGS:
        return import_optional_dependency("duckduckgo_search").DDGS()

    def search(self, query: str, **kwargs) -> ListArtifact:
        try:
            results = self.client.text(
                query, region=f"{self.language}-{self.country}", max_results=self.
↪results_count, **kwargs
            )
            return ListArtifact(
                [
                    TextArtifact(
                        json.dumps({"title": result["title"], "url": result["href"],
↪"description": result["body"]}),
                    )
                    for result in results
                ],
            )
        except Exception as e:
            raise Exception(f"Error searching {query} with DuckDuckGo: {e}") from e
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `country = field(default='us', kw_only=True)` class-attribute instance-attribute
- `language = field(default='en', kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/web_search/duck_duck_go_web_search_driver.py

```
@lazy_property()
def client(self) -> DDGS:
    return import_optional_dependency("duckduckgo_search").DDGS()
```

search(query, **kwargs)

Source Code in griptape/drivers/web_search/duck_duck_go_web_search_driver.py

```
def search(self, query: str, **kwargs) -> ListArtifact:
    try:
        results = self.client.text(
            query, region=f"{self.language}-{self.country}", max_results=self.results_
↪count, **kwargs
        )
        return ListArtifact(
            [
                TextArtifact(
                    json.dumps({"title": result["title"], "url": result["href"],
↪"description": result["body"]}),
```

(continues on next page)

(continued from previous page)

```
                )
                for result in results
            ],
        )
    except Exception as e:
        raise Exception(f"Error searching {query} with DuckDuckGo: {e}") from e
```

duck_duck_go_web_search_driver

Bases: *BaseWebSearchDriver* “)

Source Code in griptape/drivers/web_search/duck_duck_go_web_search_driver.py

```
@define
class DuckDuckGoWebSearchDriver(BaseWebSearchDriver):
    language: str = field(default="en", kw_only=True)
    country: str = field(default="us", kw_only=True)
    _client: Optional[DDGS] = field(default=None, kw_only=True, alias="client",␣
↪metadata={"serializable": False})

    @lazy_property()
    def client(self) -> DDGS:
        return import_optional_dependency("duckduckgo_search").DDGS()

    def search(self, query: str, **kwargs) -> ListArtifact:
        try:
            results = self.client.text(
                query, region=f"{self.language}-{self.country}", max_results=self.
↪results_count, **kwargs
            )
            return ListArtifact(
                [
                    TextArtifact(
                        json.dumps({"title": result["title"], "url": result["href"],
↪"description": result["body"]}),
                    )
                    for result in results
                ],
            )
        except Exception as e:
            raise Exception(f"Error searching {query} with DuckDuckGo: {e}") from e
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `country = field(default='us', kw_only=True)` class-attribute instance-attribute
- `language = field(default='en', kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/web_search/duck_duck_go_web_search_driver.py

```
@lazy_property()
def client(self) -> DDGS:
    return import_optional_dependency("duckduckgo_search").DDGS()
```

search(query, **kwargs)

Source Code in griptape/drivers/web_search/duck_duck_go_web_search_driver.py

```
def search(self, query: str, **kwargs) -> ListArtifact:
    try:
        results = self.client.text(
            query, region=f"{self.language}-{self.country}", max_results=self.results_
↪count, **kwargs
        )
        return ListArtifact(
            [
                TextArtifact(
                    json.dumps({"title": result["title"], "url": result["href"],
↪"description": result["body"]}),
                )
                for result in results
            ],
        )
    except Exception as e:
        raise Exception(f"Error searching {query} with DuckDuckGo: {e}") from e
```

exa

- `__all__ = ['ExaWebSearchDriver']` module-attribute

Bases: *BaseWebSearchDriver* “)

Source Code in griptape/drivers/web_search/exa_web_search_driver.py

```
@define
class ExaWebSearchDriver(BaseWebSearchDriver):
    api_key: str = field(kw_only=True, default=None)
    highlights: bool = field(default=False, kw_only=True)
    use_autoprompt: bool = field(default=False, kw_only=True)
    params: dict[str, Any] = field(factory=dict, kw_only=True, metadata={"serializable
↪": True})
    _client: Optional[Exa] = field(default=None, kw_only=True, alias="client")

    @lazy_property()
    def client(self) -> Exa:
        return import_optional_dependency("exa_py").Exa(api_key=self.api_key)

    def search(self, query: str, **kwargs) -> ListArtifact[JsonArtifact]:
        response = self.client.search_and_contents(  # pyright:␣
↪ignore[reportCallIssue]
```

(continues on next page)

(continued from previous page)

```
            highlights=self.highlights,  # pyright: ignore[reportArgumentType]
            use_autoprompt=self.use_autoprompt,
            query=query,
            num_results=self.results_count,
            text=True,
            **self.params,
            **kwargs,
        )
        results = [
            {"title": result.title, "url": result.url, "highlights": result.
↪highlights, "text": result.text}
            for result in response.results
        ]
        return ListArtifact([JsonArtifact(result) for result in results])
```

- `_client = field(default=None, kw_only=True, alias='client')` class-attribute instance-attribute
- `api_key = field(kw_only=True, default=None)` class-attribute instance-attribute
- `highlights = field(default=False, kw_only=True)` class-attribute instance-attribute
- `params = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `use_autoprompt = field(default=False, kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/web_search/exa_web_search_driver.py

```
@lazy_property()
def client(self) -> Exa:
    return import_optional_dependency("exa_py").Exa(api_key=self.api_key)
```

search(query, **kwargs)

Source Code in griptape/drivers/web_search/exa_web_search_driver.py

```
def search(self, query: str, **kwargs) -> ListArtifact[JsonArtifact]:
    response = self.client.search_and_contents(  # pyright: ignore[reportCallIssue]
        highlights=self.highlights,  # pyright: ignore[reportArgumentType]
        use_autoprompt=self.use_autoprompt,
        query=query,
        num_results=self.results_count,
        text=True,
        **self.params,
        **kwargs,
    )
    results = [
        {"title": result.title, "url": result.url, "highlights": result.highlights,
↪"text": result.text}
        for result in response.results
```

(continues on next page)

(continued from previous page)

```
    ]
    return ListArtifact([JsonArtifact(result) for result in results])
```

exa_web_search_driver

Bases: *BaseWebSearchDriver* “)

Source Code in griptape/drivers/web_search/exa_web_search_driver.py

```
@define
class ExaWebSearchDriver(BaseWebSearchDriver):
    api_key: str = field(kw_only=True, default=None)
    highlights: bool = field(default=False, kw_only=True)
    use_autoprompt: bool = field(default=False, kw_only=True)
    params: dict[str, Any] = field(factory=dict, kw_only=True, metadata={"serializable
↪": True})
    _client: Optional[Exa] = field(default=None, kw_only=True, alias="client")

    @lazy_property()
    def client(self) -> Exa:
        return import_optional_dependency("exa_py").Exa(api_key=self.api_key)

    def search(self, query: str, **kwargs) -> ListArtifact[JsonArtifact]:
        response = self.client.search_and_contents(  # pyright:␣
↪ignore[reportCallIssue]
            highlights=self.highlights,  # pyright: ignore[reportArgumentType]
            use_autoprompt=self.use_autoprompt,
            query=query,
            num_results=self.results_count,
            text=True,
            **self.params,
            **kwargs,
        )
        results = [
            {"title": result.title, "url": result.url, "highlights": result.
↪highlights, "text": result.text}
            for result in response.results
        ]
        return ListArtifact([JsonArtifact(result) for result in results])
```

- `_client = field(default=None, kw_only=True, alias='client')` class-attribute instance-attribute
- `api_key = field(kw_only=True, default=None)` class-attribute instance-attribute
- `highlights = field(default=False, kw_only=True)` class-attribute instance-attribute
- `params = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `use_autoprompt = field(default=False, kw_only=True)` class-attribute instance-attribute

client()

Source Code in griptape/drivers/web_search/exa_web_search_driver.py

```
@lazy_property()
def client(self) -> Exa:
    return import_optional_dependency("exa_py").Exa(api_key=self.api_key)
```

search(query, **kwargs)

Source Code in griptape/drivers/web_search/exa_web_search_driver.py

```
def search(self, query: str, **kwargs) -> ListArtifact[JsonArtifact]:
    response = self.client.search_and_contents(  # pyright: ignore[reportCallIssue]
        highlights=self.highlights,  # pyright: ignore[reportArgumentType]
        use_autoprompt=self.use_autoprompt,
        query=query,
        num_results=self.results_count,
        text=True,
        **self.params,
        **kwargs,
    )
    results = [
        {"title": result.title, "url": result.url, "highlights": result.highlights,
↪"text": result.text}
        for result in response.results
    ]
    return ListArtifact([JsonArtifact(result) for result in results])
```

google

- `__all__ = ['GoogleWebSearchDriver']` module-attribute

Bases: *BaseWebSearchDriver* “)

Source Code in griptape/drivers/web_search/google_web_search_driver.py

```
@define
class GoogleWebSearchDriver(BaseWebSearchDriver):
    api_key: str = field(kw_only=True)
    search_id: str = field(kw_only=True)
    language: str = field(default="en", kw_only=True)
    country: str = field(default="us", kw_only=True)

    def search(self, query: str, **kwargs) -> ListArtifact:
        return ListArtifact([TextArtifact(json.dumps(result)) for result in self._
↪search_google(query, **kwargs)])

    def _search_google(self, query: str, **kwargs) -> list[dict]:
        query_params = {
            "key": self.api_key,
            "cx": self.search_id,
            "q": query,
            "start": 0,
```

(continues on next page)

(continued from previous page)

```
            "lr": f"lang_{self.language}",
            "num": self.results_count,
            "gl": self.country,
            **kwargs,
        }
        response = requests.get("https://www.googleapis.com/customsearch/v1",␣
↪params=query_params)

        if response.status_code == 200:
            data = response.json()

            return [{"url": r["link"], "title": r["title"], "description": r["snippet
↪"]} for r in data["items"]]

        raise Exception(
            f"Google Search API returned an error with status code "
            f"{response.status_code} and reason {response.reason}",
        )
```

- `api_key = field(kw_only=True)` class-attribute instance-attribute
- `country = field(default='us', kw_only=True)` class-attribute instance-attribute
- `language = field(default='en', kw_only=True)` class-attribute instance-attribute
- `search_id = field(kw_only=True)` class-attribute instance-attribute

_search_google(query, **kwargs)

Source Code in griptape/drivers/web_search/google_web_search_driver.py

```
def _search_google(self, query: str, **kwargs) -> list[dict]:
    query_params = {
        "key": self.api_key,
        "cx": self.search_id,
        "q": query,
        "start": 0,
        "lr": f"lang_{self.language}",
        "num": self.results_count,
        "gl": self.country,
        **kwargs,
    }
    response = requests.get("https://www.googleapis.com/customsearch/v1",␣
↪params=query_params)

    if response.status_code == 200:
        data = response.json()

        return [{"url": r["link"], "title": r["title"], "description": r["snippet"]}␣
↪for r in data["items"]]

    raise Exception(
        f"Google Search API returned an error with status code "
```

(continues on next page)

(continued from previous page)

```
        f"{response.status_code} and reason {response.reason}",
    )
```

search(query, **kwargs)

Source Code in griptape/drivers/web_search/google_web_search_driver.py

```
def search(self, query: str, **kwargs) -> ListArtifact:
    return ListArtifact([TextArtifact(json.dumps(result)) for result in self._search_
↪google(query, **kwargs)])
```

google_web_search_driver

Bases: *BaseWebSearchDriver* “)

Source Code in griptape/drivers/web_search/google_web_search_driver.py

```
@define
class GoogleWebSearchDriver(BaseWebSearchDriver):
    api_key: str = field(kw_only=True)
    search_id: str = field(kw_only=True)
    language: str = field(default="en", kw_only=True)
    country: str = field(default="us", kw_only=True)

    def search(self, query: str, **kwargs) -> ListArtifact:
        return ListArtifact([TextArtifact(json.dumps(result)) for result in self._
↪search_google(query, **kwargs)])

    def _search_google(self, query: str, **kwargs) -> list[dict]:
        query_params = {
            "key": self.api_key,
            "cx": self.search_id,
            "q": query,
            "start": 0,
            "lr": f"lang_{self.language}",
            "num": self.results_count,
            "gl": self.country,
            **kwargs,
        }
        response = requests.get("https://www.googleapis.com/customsearch/v1",␣
↪params=query_params)

        if response.status_code == 200:
            data = response.json()

            return [{"url": r["link"], "title": r["title"], "description": r["snippet
↪"]} for r in data["items"]]

        raise Exception(
            f"Google Search API returned an error with status code "
            f"{response.status_code} and reason {response.reason}",
        )
```

- `api_key = field(kw_only=True)` class-attribute instance-attribute
- `country = field(default='us', kw_only=True)` class-attribute instance-attribute
- `language = field(default='en', kw_only=True)` class-attribute instance-attribute
- `search_id = field(kw_only=True)` class-attribute instance-attribute

_search_google(query, **kwargs)

Source Code in griptape/drivers/web_search/google_web_search_driver.py

```
def _search_google(self, query: str, **kwargs) -> list[dict]:
    query_params = {
        "key": self.api_key,
        "cx": self.search_id,
        "q": query,
        "start": 0,
        "lr": f"lang_{self.language}",
        "num": self.results_count,
        "gl": self.country,
        **kwargs,
    }
    response = requests.get("https://www.googleapis.com/customsearch/v1",␣
↪params=query_params)

    if response.status_code == 200:
        data = response.json()

        return [{"url": r["link"], "title": r["title"], "description": r["snippet"]}␣
↪for r in data["items"]]

    raise Exception(
        f"Google Search API returned an error with status code "
        f"{response.status_code} and reason {response.reason}",
    )
```

search(query, **kwargs)

Source Code in griptape/drivers/web_search/google_web_search_driver.py

```
def search(self, query: str, **kwargs) -> ListArtifact:
    return ListArtifact([TextArtifact(json.dumps(result)) for result in self._search_
↪google(query, **kwargs)])
```

perplexity

- `__all__ = ['PerplexityWebSearchDriver']` module-attribute

Bases: *BaseWebSearchDriver* “)

Source Code in griptape/drivers/web_search/perplexity_web_search_driver.py

```
@define
class PerplexityWebSearchDriver(BaseWebSearchDriver):
```

(continues on next page)

(continued from previous page)

```
    model: str = field(default="sonar-pro", kw_only=True, metadata={"serializable":␣
↪True})
    api_key: Optional[str] = field(kw_only=True, default=None)
    _prompt_driver: Optional[PerplexityPromptDriver] = field(default=None, alias=
↪"prompt_driver")

    @lazy_property()
    def prompt_driver(self) -> PerplexityPromptDriver:
        if self.api_key is None:
            raise ValueError("api_key is required")
        return PerplexityPromptDriver(model=self.model, api_key=self.api_key)

    def search(self, query: str, **kwargs) -> ListArtifact:
        message = self.prompt_driver.run(PromptStack.from_
↪artifact(TextArtifact(query)))

        return ListArtifact([message.to_artifact()])
```

- `_prompt_driver = field(default=None, alias='prompt_driver')` class-attribute instance-attribute
- `api_key = field(kw_only=True, default=None)` class-attribute instance-attribute
- `model = field(default='sonar-pro', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

prompt_driver()

Source Code in griptape/drivers/web_search/perplexity_web_search_driver.py

```
@lazy_property()
def prompt_driver(self) -> PerplexityPromptDriver:
    if self.api_key is None:
        raise ValueError("api_key is required")
    return PerplexityPromptDriver(model=self.model, api_key=self.api_key)
```

search(query, **kwargs)

Source Code in griptape/drivers/web_search/perplexity_web_search_driver.py

```
def search(self, query: str, **kwargs) -> ListArtifact:
    message = self.prompt_driver.run(PromptStack.from_artifact(TextArtifact(query)))

    return ListArtifact([message.to_artifact()])
```

perplexity_web_search_driver

Bases: *BaseWebSearchDriver* “)

Source Code in griptape/drivers/web_search/perplexity_web_search_driver.py

```
@define
class PerplexityWebSearchDriver(BaseWebSearchDriver):
```

(continues on next page)

(continued from previous page)

```
    model: str = field(default="sonar-pro", kw_only=True, metadata={"serializable":␣
↪True})
    api_key: Optional[str] = field(kw_only=True, default=None)
    _prompt_driver: Optional[PerplexityPromptDriver] = field(default=None, alias=
↪"prompt_driver")

    @lazy_property()
    def prompt_driver(self) -> PerplexityPromptDriver:
        if self.api_key is None:
            raise ValueError("api_key is required")
        return PerplexityPromptDriver(model=self.model, api_key=self.api_key)

    def search(self, query: str, **kwargs) -> ListArtifact:
        message = self.prompt_driver.run(PromptStack.from_
↪artifact(TextArtifact(query)))

        return ListArtifact([message.to_artifact()])
```

- `_prompt_driver = field(default=None, alias='prompt_driver')` class-attribute instance-attribute
- `api_key = field(kw_only=True, default=None)` class-attribute instance-attribute
- `model = field(default='sonar-pro', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

prompt_driver()

Source Code in griptape/drivers/web_search/perplexity_web_search_driver.py

```
@lazy_property()
def prompt_driver(self) -> PerplexityPromptDriver:
    if self.api_key is None:
        raise ValueError("api_key is required")
    return PerplexityPromptDriver(model=self.model, api_key=self.api_key)
```

search(query, **kwargs)

Source Code in griptape/drivers/web_search/perplexity_web_search_driver.py

```
def search(self, query: str, **kwargs) -> ListArtifact:
    message = self.prompt_driver.run(PromptStack.from_artifact(TextArtifact(query)))

    return ListArtifact([message.to_artifact()])
```

tavily

- `__all__ = ['TavilyWebSearchDriver']` module-attribute

Bases: *BaseWebSearchDriver* “)

Source Code in griptape/drivers/web_search/tavily_web_search_driver.py

```
@define
class TavilyWebSearchDriver(BaseWebSearchDriver):
    api_key: str = field(kw_only=True)
    params: dict[str, Any] = field(factory=dict, kw_only=True, metadata={"serializable
↪": True})
    _client: Optional[TavilyClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> TavilyClient:
        return import_optional_dependency("tavily").TavilyClient(self.api_key)

    def search(self, query: str, **kwargs) -> ListArtifact:
        response = self.client.search(query, max_results=self.results_count, **self.
↪params, **kwargs)
        results = response.get("results", [])
        return ListArtifact([(JsonArtifact(result)) for result in results])
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(kw_only=True)` class-attribute instance-attribute
- `params = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/web_search/tavily_web_search_driver.py

```
@lazy_property()
def client(self) -> TavilyClient:
    return import_optional_dependency("tavily").TavilyClient(self.api_key)
```

search(query, **kwargs)

Source Code in griptape/drivers/web_search/tavily_web_search_driver.py

```
def search(self, query: str, **kwargs) -> ListArtifact:
    response = self.client.search(query, max_results=self.results_count, **self.
↪params, **kwargs)
    results = response.get("results", [])
    return ListArtifact([(JsonArtifact(result)) for result in results])
```

tavily_web_search_driver

Bases: *BaseWebSearchDriver* “)

Source Code in griptape/drivers/web_search/tavily_web_search_driver.py

```
@define
class TavilyWebSearchDriver(BaseWebSearchDriver):
    api_key: str = field(kw_only=True)
```

(continues on next page)

(continued from previous page)

```
    params: dict[str, Any] = field(factory=dict, kw_only=True, metadata={"serializable
↪": True})
    _client: Optional[TavilyClient] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> TavilyClient:
        return import_optional_dependency("tavily").TavilyClient(self.api_key)

    def search(self, query: str, **kwargs) -> ListArtifact:
        response = self.client.search(query, max_results=self.results_count, **self.
↪params, **kwargs)
        results = response.get("results", [])
        return ListArtifact([(JsonArtifact(result)) for result in results])
```

- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(kw_only=True)` class-attribute instance-attribute
- `params = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/drivers/web_search/tavily_web_search_driver.py

```
@lazy_property()
def client(self) -> TavilyClient:
    return import_optional_dependency("tavily").TavilyClient(self.api_key)
```

search(query, **kwargs)

Source Code in griptape/drivers/web_search/tavily_web_search_driver.py

```
def search(self, query: str, **kwargs) -> ListArtifact:
    response = self.client.search(query, max_results=self.results_count, **self.
↪params, **kwargs)
    results = response.get("results", [])
    return ListArtifact([(JsonArtifact(result)) for result in results])
```

11.5.8 engines

- `__all__ = ['BaseEvalEngine', 'BaseExtractionEngine', 'BaseSummaryEngine', 'CsvExtractionEngine', 'EvalEngine', 'JsonExtractionEngine', 'PromptSummaryEngine', 'RagEngine']` module-attribute

Bases:

ABC

Source Code in griptape/engines/eval/base_eval_engine.py

```
@define
class BaseEvalEngine(ABC): ...
```

BaseExtractionEngine

Bases:

ABC

Source Code in griptape/engines/extraction/base_extraction_engine.py

```
@define
class BaseExtractionEngine(ABC):
    max_token_multiplier: float = field(default=0.5, kw_only=True)
    chunk_joiner: str = field(default="\n\n", kw_only=True)
    prompt_driver: BasePromptDriver = field(
        default=Factory(lambda: Defaults.drivers_config.prompt_driver), kw_only=True
    )
    chunker: BaseChunker = field(
        default=Factory(
            lambda self: TextChunker(tokenizer=self.prompt_driver.tokenizer, max_
↪tokens=self.max_chunker_tokens),
            takes_self=True,
        ),
        kw_only=True,
    )

    @max_token_multiplier.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_max_token_multiplier(self, _: Attribute, max_token_multiplier: int) -
↪> None:
        if max_token_multiplier > 1:
            raise ValueError("has to be less than or equal to 1")
        if max_token_multiplier <= 0:
            raise ValueError("has to be greater than 0")

    @property
    def max_chunker_tokens(self) -> int:
        return round(self.prompt_driver.tokenizer.max_input_tokens * self.max_token_
↪multiplier)

    @property
    def min_response_tokens(self) -> int:
        return round(
            self.prompt_driver.tokenizer.max_input_tokens
            - self.prompt_driver.tokenizer.max_input_tokens * self.max_token_
↪multiplier,
        )

    def extract_text(
        self,
        text: str,
        *,
        rulesets: Optional[list[Ruleset]] = None,
        **kwargs,
```

(continues on next page)

(continued from previous page)

```
    ) -> ListArtifact:
        return self.extract_artifacts(ListArtifact([TextArtifact(text)]),␣
↪rulesets=rulesets, **kwargs)

    @abstractmethod
    def extract_artifacts(
        self,
        artifacts: ListArtifact[TextArtifact],
        *,
        rulesets: Optional[list[Ruleset]] = None,
        **kwargs,
    ) -> ListArtifact: ...
```

- `chunk_joiner = field(default='\n\n', kw_only=True)` class-attribute instance-attribute
- `chunker = field(default=Factory(lambda self: TextChunker(tokenizer=self.prompt_driver.tokenizer, max_tokens=self.max_chunker_tokens), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `max_chunker_tokens` property
- `max_token_multiplier = field(default=0.5, kw_only=True)` class-attribute instance-attribute
- `min_response_tokens` property
- `prompt_driver = field(default=Factory(lambda: Defaults.drivers_config.prompt_driver), kw_only=True)` class-attribute instance-attribute

extract_artifacts(artifacts, *, rulesets=None, **kwargs)abstractmethod

Source Code in griptape/engines/extraction/base_extraction_engine.py

```
@abstractmethod
def extract_artifacts(
    self,
    artifacts: ListArtifact[TextArtifact],
    *,
    rulesets: Optional[list[Ruleset]] = None,
    **kwargs,
) -> ListArtifact: ...
```

extract_text(text, *, rulesets=None, **kwargs)

Source Code in griptape/engines/extraction/base_extraction_engine.py

```
def extract_text(
    self,
    text: str,
    *,
    rulesets: Optional[list[Ruleset]] = None,
    **kwargs,
) -> ListArtifact:
    return self.extract_artifacts(ListArtifact([TextArtifact(text)]),␣
↪rulesets=rulesets, **kwargs)
```

validate*max_token_multiplier(*, max_token_multiplier)

Source Code in griptape/engines/extraction/base_extraction_engine.py

```
@max_token_multiplier.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_max_token_multiplier(self, _: Attribute, max_token_multiplier: int) ->␣
↪None:
    if max_token_multiplier > 1:
        raise ValueError("has to be less than or equal to 1")
    if max_token_multiplier <= 0:
        raise ValueError("has to be greater than 0")
```

BaseSummaryEngine

Bases:

ABC

Source Code in griptape/engines/summary/base_summary_engine.py

```
@define
class BaseSummaryEngine(ABC):
    def summarize_text(self, text: str, *, rulesets: Optional[list[Ruleset]] = None) -
↪> str:
        return self.summarize_artifacts(ListArtifact([TextArtifact(text)]),␣
↪rulesets=rulesets).value

    @abstractmethod
    def summarize_artifacts(
        self,
        artifacts: ListArtifact,
        *,
        rulesets: Optional[list[Ruleset]] = None,
    ) -> TextArtifact: ...
```

summarize_artifacts(artifacts, *, rulesets=None)abstractmethod

Source Code in griptape/engines/summary/base_summary_engine.py

```
@abstractmethod
def summarize_artifacts(
    self,
    artifacts: ListArtifact,
    *,
    rulesets: Optional[list[Ruleset]] = None,
) -> TextArtifact: ...
```

summarize_text(text, *, rulesets=None)

Source Code in griptape/engines/summary/base_summary_engine.py

```
def summarize_text(self, text: str, *, rulesets: Optional[list[Ruleset]] = None) ->␣
↪str:
```

(continues on next page)

(continued from previous page)

```
    return self.summarize_artifacts(ListArtifact([TextArtifact(text)]),␣
↪rulesets=rulesets).value
```

CsvExtractionEngine

Bases: *BaseExtractionEngine* “)

Source Code in griptape/engines/extraction/csv_extraction_engine.py

```
@define
class CsvExtractionEngine(BaseExtractionEngine):
    column_names: list[str] = field(kw_only=True)
    generate_system_template: J2 = field(default=Factory(lambda: J2("engines/
↪extraction/csv/system.j2")), kw_only=True)
    generate_user_template: J2 = field(default=Factory(lambda: J2("engines/extraction/
↪csv/user.j2")), kw_only=True)
    format_header: Callable[[list[str]], str] = field(
        default=Factory(lambda: lambda value: ",".join(value)), kw_only=True
    )
    format_row: Callable[[dict], str] = field(
        default=Factory(lambda: lambda value: ",".join([value or "" for value in␣
↪value.values()])), kw_only=True
    )

    def extract_artifacts(
        self,
        artifacts: ListArtifact[TextArtifact],
        *,
        rulesets: Optional[list[Ruleset]] = None,
        **kwargs,
    ) -> ListArtifact[TextArtifact]:
        return ListArtifact(
            self._extract_rec(
                cast("list[TextArtifact]", artifacts.value),
                [TextArtifact(self.format_header(self.column_names))],
                rulesets=rulesets,
            ),
            item_separator="\n",
        )

    def text_to_csv_rows(self, text: str) -> list[TextArtifact]:
        rows = []

        with io.StringIO(text) as f:
            for row in csv.DictReader(f):
                rows.append(TextArtifact(self.format_row(row)))

        return rows

    def _extract_rec(
        self,
        artifacts: list[TextArtifact],
```

(continues on next page)

(continued from previous page)

```
        rows: list[TextArtifact],
        *,
        rulesets: Optional[list[Ruleset]] = None,
    ) -> list[TextArtifact]:
        artifacts_text = self.chunk_joiner.join([a.value for a in artifacts])
        system_prompt = self.generate_system_template.render(
            column_names=self.column_names,
            rulesets=J2("rulesets/rulesets.j2").render(rulesets=rulesets),
        )
        user_prompt = self.generate_user_template.render(
            text=artifacts_text,
        )

        if (
            self.prompt_driver.tokenizer.count_input_tokens_left(system_prompt + user_
↪prompt)
            >= self.min_response_tokens
        ):
            rows.extend(
                self.text_to_csv_rows(
                    self.prompt_driver.run(
                        PromptStack(
                            messages=[
                                Message(system_prompt, role=Message.SYSTEM_ROLE),
                                Message(user_prompt, role=Message.USER_ROLE),
                            ]
                        )
                    ).value,
                ),
            )

            return rows
        chunks = self.chunker.chunk(artifacts_text)
        partial_text = self.generate_user_template.render(
            text=chunks[0].value,
        )

        rows.extend(
            self.text_to_csv_rows(
                self.prompt_driver.run(
                    PromptStack(
                        messages=[
                            Message(system_prompt, role=Message.SYSTEM_ROLE),
                            Message(partial_text, role=Message.USER_ROLE),
                        ]
                    )
                ).value,
            ),
        )

        return self._extract_rec(chunks[1:], rows, rulesets=rulesets)
```

- `column_names = field(kw_only=True)` class-attribute instance-attribute

- `format_header = field(default=Factory(lambda: lambda value: ','.join(value)), kw_only=True)` class-attribute instance-attribute
- `format_row = field(default=Factory(lambda: lambda value: ','.join([value or '' for value in value.values()])), kw_only=True)` class-attribute instance-attribute
- `generate_system_template = field(default=Factory(lambda: J2('engines/extraction/csv/system.j2')), kw_only=True)` class-attribute instance-attribute
- `generate_user_template = field(default=Factory(lambda: J2('engines/extraction/csv/user.j2')), kw_only=True)` class-attribute instance-attribute

_extract_rec(artifacts, rows, *, rulesets=None)

Source Code in griptape/engines/extraction/csv_extraction_engine.py

```
def _extract_rec(
    self,
    artifacts: list[TextArtifact],
    rows: list[TextArtifact],
    *,
    rulesets: Optional[list[Ruleset]] = None,
) -> list[TextArtifact]:
    artifacts_text = self.chunk_joiner.join([a.value for a in artifacts])
    system_prompt = self.generate_system_template.render(
        column_names=self.column_names,
        rulesets=J2("rulesets/rulesets.j2").render(rulesets=rulesets),
    )
    user_prompt = self.generate_user_template.render(
        text=artifacts_text,
    )

    if (
        self.prompt_driver.tokenizer.count_input_tokens_left(system_prompt + user_
↪prompt)
        >= self.min_response_tokens
    ):
        rows.extend(
            self.text_to_csv_rows(
                self.prompt_driver.run(
                    PromptStack(
                        messages=[
                            Message(system_prompt, role=Message.SYSTEM_ROLE),
                            Message(user_prompt, role=Message.USER_ROLE),
                        ]
                    )
                ).value,
            ),
        )

        return rows
    chunks = self.chunker.chunk(artifacts_text)
    partial_text = self.generate_user_template.render(
        text=chunks[0].value,
    )
```

(continues on next page)

(continued from previous page)

```
    rows.extend(
        self.text_to_csv_rows(
            self.prompt_driver.run(
                PromptStack(
                    messages=[
                        Message(system_prompt, role=Message.SYSTEM_ROLE),
                        Message(partial_text, role=Message.USER_ROLE),
                    ]
                )
            ).value,
        ),
    )

    return self._extract_rec(chunks[1:], rows, rulesets=rulesets)
```

extract_artifacts(artifacts, *, rulesets=None, **kwargs)

Source Code in griptape/engines/extraction/csv_extraction_engine.py

```
def extract_artifacts(
    self,
    artifacts: ListArtifact[TextArtifact],
    *,
    rulesets: Optional[list[Ruleset]] = None,
    **kwargs,
) -> ListArtifact[TextArtifact]:
    return ListArtifact(
        self._extract_rec(
            cast("list[TextArtifact]", artifacts.value),
            [TextArtifact(self.format_header(self.column_names))],
            rulesets=rulesets,
        ),
        item_separator="\n",
    )
```

text_to_csv_rows(text)

Source Code in griptape/engines/extraction/csv_extraction_engine.py

```
def text_to_csv_rows(self, text: str) -> list[TextArtifact]:
    rows = []

    with io.StringIO(text) as f:
        for row in csv.DictReader(f):
            rows.append(TextArtifact(self.format_row(row)))

    return rows
```

EvalEngine

Bases: BaseEvalEngine “) , *SerializableMixin* ”)

Source Code in griptape/engines/eval/eval_engine.py

```
@define(kw_only=True)
class EvalEngine(BaseEvalEngine, SerializableMixin):
    id: str = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata=
↪{"serializable": True})
    name: str = field(
        default=Factory(lambda self: self.id, takes_self=True),
        metadata={"serializable": True},
    )
    criteria: Optional[str] = field(default=None, metadata={"serializable": True})
    evaluation_steps: Optional[list[str]] = field(default=None, metadata={
↪"serializable": True})
    prompt_driver: BasePromptDriver = field(default=Factory(lambda: Defaults.drivers_
↪config.prompt_driver))
    generate_steps_system_template: J2 = field(default=Factory(lambda: J2("engines/
↪eval/steps/system.j2")))
    generate_steps_user_template: J2 = field(default=Factory(lambda: J2("engines/eval/
↪steps/user.j2")))
    generate_results_system_template: J2 = field(default=Factory(lambda: J2("engines/
↪eval/results/system.j2")))
    generate_results_user_template: J2 = field(default=Factory(lambda: J2("engines/
↪eval/results/user.j2")))

    @criteria.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_criteria(self, _: Attribute, value: Optional[str]) -> None:
        if value is None:
            if self.evaluation_steps is None:
                raise ValueError("either criteria or evaluation_steps must be␣
↪specified")
            return

        if self.evaluation_steps is not None:
            raise ValueError("cant have both criteria and evaluation_steps specified")

        if not value:
            raise ValueError("criteria must not be empty")

    @evaluation_steps.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_evaluation_steps(self, _: Attribute, value: Optional[list[str]]) ->␣
↪None:
        if value is None:
            if self.criteria is None:
                raise ValueError("either evaluation_steps or criteria must be␣
↪specified")
            return

        if self.criteria is not None:
```

(continues on next page)

(continued from previous page)

```
            raise ValueError("cant have both evaluation_steps and criteria specified")

        if not value:
            raise ValueError("evaluation_steps must not be empty")

    def evaluate(self, input: str, actual_output: str, **kwargs) -> tuple[float,␣
↪str]:  # noqa: A002
        evaluation_params = {
            key.replace("_", " ").title(): value
            for key, value in {"input": input, "actual_output": actual_output,␣
↪**kwargs}.items()
        }

        if self.evaluation_steps is None:
            # Need to disable validators to allow for both `criteria` and `evaluation_
↪steps` to be set
            with validators.disabled():
                self.evaluation_steps = self._generate_steps(evaluation_params)

        return self._generate_results(evaluation_params)

    def _generate_steps(self, evaluation_params: dict[str, str]) -> list[str]:
        system_prompt = self.generate_steps_system_template.render(
            evaluation_params=", ".join(param for param in evaluation_params),
            criteria=self.criteria,
        )
        user_prompt = self.generate_steps_user_template.render()

        result = self.prompt_driver.run(
            PromptStack(
                messages=[
                    Message(system_prompt, role=Message.SYSTEM_ROLE),
                    Message(user_prompt, role=Message.USER_ROLE),
                ],
                output_schema=STEPS_SCHEMA,
            ),
        ).to_artifact()

        parsed_result = json.loads(result.value)

        return parsed_result["steps"]

    def _generate_results(self, evaluation_params: dict[str, str]) -> tuple[float,␣
↪str]:
        system_prompt = self.generate_results_system_template.render(
            evaluation_params=", ".join(param for param in evaluation_params),
            evaluation_steps=self.evaluation_steps,
            evaluation_text="\n\n".join(f"{key}: {value}" for key, value in␣
↪evaluation_params.items()),
        )
        user_prompt = self.generate_results_user_template.render()
```

(continues on next page)

(continued from previous page)

```
        result = self.prompt_driver.run(
            PromptStack(
                messages=[
                    Message(system_prompt, role=Message.SYSTEM_ROLE),
                    Message(user_prompt, role=Message.USER_ROLE),
                ],
                output_schema=RESULTS_SCHEMA,
            ),
        ).to_text()

        parsed_result = json.loads(result)

        # Better to have the LLM deal strictly with integers to avoid ambiguities␣
↪with floating point precision.
        # We want the user to receive a float, however.
        score = float(parsed_result["score"]) / 10
        reason = parsed_result["reason"]

        return score, reason
```

- `criteria = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `evaluation_steps = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `generate_results_system_template = field(default=Factory(lambda: J2('engines/eval/results/system.j2')))` class-attribute instance-attribute
- `generate_results_user_template = field(default=Factory(lambda: J2('engines/eval/results/user.j2')))` class-attribute instance-attribute
- `generate_steps_system_template = field(default=Factory(lambda: J2('engines/eval/steps/system.j2')))` class-attribute instance-attribute
- `generate_steps_user_template = field(default=Factory(lambda: J2('engines/eval/steps/user.j2')))` class-attribute instance-attribute
- `id = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `name = field(default=Factory(lambda self: self.id, takes_self=True), metadata={'serializable': True})` class-attribute instance-attribute
- `prompt_driver = field(default=Factory(lambda: Defaults.drivers_config.prompt_driver))` class-attribute instance-attribute

_generate_results(evaluation_params)

Source Code in griptape/engines/eval/eval_engine.py

```
def _generate_results(self, evaluation_params: dict[str, str]) -> tuple[float, str]:
    system_prompt = self.generate_results_system_template.render(
        evaluation_params=", ".join(param for param in evaluation_params),
        evaluation_steps=self.evaluation_steps,
        evaluation_text="\n\n".join(f"{key}: {value}" for key, value in evaluation_
```

(continues on next page)

(continued from previous page)

```
↪params.items()),
    )
    user_prompt = self.generate_results_user_template.render()

    result = self.prompt_driver.run(
        PromptStack(
            messages=[
                Message(system_prompt, role=Message.SYSTEM_ROLE),
                Message(user_prompt, role=Message.USER_ROLE),
            ],
            output_schema=RESULTS_SCHEMA,
        ),
    ).to_text()

    parsed_result = json.loads(result)

    # Better to have the LLM deal strictly with integers to avoid ambiguities with␣
↪floating point precision.
    # We want the user to receive a float, however.
    score = float(parsed_result["score"]) / 10
    reason = parsed_result["reason"]

    return score, reason
```

_generate_steps(evaluation_params)

Source Code in griptape/engines/eval/eval_engine.py

```
def _generate_steps(self, evaluation_params: dict[str, str]) -> list[str]:
    system_prompt = self.generate_steps_system_template.render(
        evaluation_params=", ".join(param for param in evaluation_params),
        criteria=self.criteria,
    )
    user_prompt = self.generate_steps_user_template.render()

    result = self.prompt_driver.run(
        PromptStack(
            messages=[
                Message(system_prompt, role=Message.SYSTEM_ROLE),
                Message(user_prompt, role=Message.USER_ROLE),
            ],
            output_schema=STEPS_SCHEMA,
        ),
    ).to_artifact()

    parsed_result = json.loads(result.value)

    return parsed_result["steps"]
```

evaluate(input, actual_output, **kwargs)

Source Code in griptape/engines/eval/eval_engine.py

```
def evaluate(self, input: str, actual_output: str, **kwargs) -> tuple[float, str]:  #␣
↪noqa: A002
    evaluation_params = {
        key.replace("_", " ").title(): value
        for key, value in {"input": input, "actual_output": actual_output, **kwargs}.
↪items()
    }

    if self.evaluation_steps is None:
        # Need to disable validators to allow for both `criteria` and `evaluation_
↪steps` to be set
        with validators.disabled():
            self.evaluation_steps = self._generate_steps(evaluation_params)

    return self._generate_results(evaluation_params)
```

validate*criteria(*, value)

Source Code in griptape/engines/eval/eval_engine.py

```
@criteria.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_criteria(self, _: Attribute, value: Optional[str]) -> None:
    if value is None:
        if self.evaluation_steps is None:
            raise ValueError("either criteria or evaluation_steps must be specified")
        return

    if self.evaluation_steps is not None:
        raise ValueError("cant have both criteria and evaluation_steps specified")

    if not value:
        raise ValueError("criteria must not be empty")
```

validate*evaluation_steps(*, value)

Source Code in griptape/engines/eval/eval_engine.py

```
@evaluation_steps.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_evaluation_steps(self, _: Attribute, value: Optional[list[str]]) -> None:
    if value is None:
        if self.criteria is None:
            raise ValueError("either evaluation_steps or criteria must be specified")
        return

    if self.criteria is not None:
        raise ValueError("cant have both evaluation_steps and criteria specified")
```

(continues on next page)

(continued from previous page)

```
    if not value:
        raise ValueError("evaluation_steps must not be empty")
```

JsonExtractionEngine

Bases: *BaseExtractionEngine* “)

Source Code in griptape/engines/extraction/json_extraction_engine.py

```
@define
class JsonExtractionEngine(BaseExtractionEngine):
    JSON_PATTERN = r"(?s)[^\[]*(\[.*\])"

    template_schema: dict = field(kw_only=True)
    generate_system_template: J2 = field(default=Factory(lambda: J2("engines/
↪extraction/json/system.j2")), kw_only=True)
    generate_user_template: J2 = field(default=Factory(lambda: J2("engines/extraction/
↪json/user.j2")), kw_only=True)

    def extract_artifacts(
        self,
        artifacts: ListArtifact[TextArtifact],
        *,
        rulesets: Optional[list[Ruleset]] = None,
        **kwargs,
    ) -> ListArtifact[JsonArtifact]:
        return ListArtifact(
            self._extract_rec(cast("list[TextArtifact]", artifacts.value), [],␣
↪rulesets=rulesets),
            item_separator="\n",
        )

    def json_to_text_artifacts(self, json_input: str) -> list[JsonArtifact]:
        json_matches = re.findall(self.JSON_PATTERN, json_input, re.DOTALL)

        if json_matches:
            return [JsonArtifact(e) for e in json.loads(json_matches[-1])]
        return []

    def _extract_rec(
        self,
        artifacts: list[TextArtifact],
        extractions: list[JsonArtifact],
        *,
        rulesets: Optional[list[Ruleset]] = None,
    ) -> list[JsonArtifact]:
        artifacts_text = self.chunk_joiner.join([a.value for a in artifacts])
        system_prompt = self.generate_system_template.render(
            json_template_schema=json.dumps(self.template_schema),
            rulesets=J2("rulesets/rulesets.j2").render(rulesets=rulesets),
        )
        user_prompt = self.generate_user_template.render(
```

(continues on next page)

(continued from previous page)

```
            text=artifacts_text,
        )

        if (
            self.prompt_driver.tokenizer.count_input_tokens_left(user_prompt + system_
↪prompt)
            >= self.min_response_tokens
        ):
            extractions.extend(
                self.json_to_text_artifacts(
                    self.prompt_driver.run(
                        PromptStack(
                            messages=[
                                Message(system_prompt, role=Message.SYSTEM_ROLE),
                                Message(user_prompt, role=Message.USER_ROLE),
                            ]
                        )
                    ).value
                ),
            )

            return extractions
        chunks = self.chunker.chunk(artifacts_text)
        partial_text = self.generate_user_template.render(
            text=chunks[0].value,
        )

        extractions.extend(
            self.json_to_text_artifacts(
                self.prompt_driver.run(
                    PromptStack(
                        messages=[
                            Message(system_prompt, role=Message.SYSTEM_ROLE),
                            Message(partial_text, role=Message.USER_ROLE),
                        ]
                    )
                ).value,
            ),
        )

        return self._extract_rec(chunks[1:], extractions, rulesets=rulesets)
```

- `JSON_PATTERN = '(?s)[^\\[]*(\\[.*\\])'` class-attribute instance-attribute
- `generate_system_template = field(default=Factory(lambda: J2('engines/extraction/json/system.j2')), kw_only=True)` class-attribute instance-attribute
- `generate_user_template = field(default=Factory(lambda: J2('engines/extraction/json/user.j2')), kw_only=True)` class-attribute instance-attribute
- `template_schema = field(kw_only=True)` class-attribute instance-attribute

_extract_rec(artifacts, extractions, *, rulesets=None)

Source Code in griptape/engines/extraction/json_extraction_engine.py

```
def _extract_rec(
    self,
    artifacts: list[TextArtifact],
    extractions: list[JsonArtifact],
    *,
    rulesets: Optional[list[Ruleset]] = None,
) -> list[JsonArtifact]:
    artifacts_text = self.chunk_joiner.join([a.value for a in artifacts])
    system_prompt = self.generate_system_template.render(
        json_template_schema=json.dumps(self.template_schema),
        rulesets=J2("rulesets/rulesets.j2").render(rulesets=rulesets),
    )
    user_prompt = self.generate_user_template.render(
        text=artifacts_text,
    )

    if (
        self.prompt_driver.tokenizer.count_input_tokens_left(user_prompt + system_
↪prompt)
        >= self.min_response_tokens
    ):
        extractions.extend(
            self.json_to_text_artifacts(
                self.prompt_driver.run(
                    PromptStack(
                        messages=[
                            Message(system_prompt, role=Message.SYSTEM_ROLE),
                            Message(user_prompt, role=Message.USER_ROLE),
                        ]
                    )
                ).value
            ),
        )

        return extractions
    chunks = self.chunker.chunk(artifacts_text)
    partial_text = self.generate_user_template.render(
        text=chunks[0].value,
    )

    extractions.extend(
        self.json_to_text_artifacts(
            self.prompt_driver.run(
                PromptStack(
                    messages=[
                        Message(system_prompt, role=Message.SYSTEM_ROLE),
                        Message(partial_text, role=Message.USER_ROLE),
                    ]
                )
            ).value,
```

(continues on next page)

(continued from previous page)

```
        ),
    )

    return self._extract_rec(chunks[1:], extractions, rulesets=rulesets)
```

extract_artifacts(artifacts, *, rulesets=None, **kwargs)

Source Code in griptape/engines/extraction/json_extraction_engine.py

```
def extract_artifacts(
    self,
    artifacts: ListArtifact[TextArtifact],
    *,
    rulesets: Optional[list[Ruleset]] = None,
    **kwargs,
) -> ListArtifact[JsonArtifact]:
    return ListArtifact(
        self._extract_rec(cast("list[TextArtifact]", artifacts.value), [],␣
↪rulesets=rulesets),
        item_separator="\n",
    )
```

json_to_text_artifacts(json_input)

Source Code in griptape/engines/extraction/json_extraction_engine.py

```
def json_to_text_artifacts(self, json_input: str) -> list[JsonArtifact]:
    json_matches = re.findall(self.JSON_PATTERN, json_input, re.DOTALL)

    if json_matches:
        return [JsonArtifact(e) for e in json.loads(json_matches[-1])]
    return []
```

PromptSummaryEngine

Bases: *BaseSummaryEngine* “)

Source Code in griptape/engines/summary/prompt_summary_engine.py

```
@define
class PromptSummaryEngine(BaseSummaryEngine):
    chunk_joiner: str = field(default="\n\n", kw_only=True)
    max_token_multiplier: float = field(default=0.5, kw_only=True)
    generate_system_template: J2 = field(default=Factory(lambda: J2("engines/summary/
↪system.j2")), kw_only=True)
    generate_user_template: J2 = field(default=Factory(lambda: J2("engines/summary/
↪user.j2")), kw_only=True)
    prompt_driver: BasePromptDriver = field(
        default=Factory(lambda: Defaults.drivers_config.prompt_driver), kw_only=True
    )
    chunker: BaseChunker = field(
        default=Factory(
```

(continues on next page)

(continued from previous page)

```
            lambda self: TextChunker(tokenizer=self.prompt_driver.tokenizer, max_
↪tokens=self.max_chunker_tokens),
            takes_self=True,
        ),
        kw_only=True,
    )

    @max_token_multiplier.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_allowlist(self, _: Attribute, max_token_multiplier: int) -> None:
        if max_token_multiplier > 1:
            raise ValueError("has to be less than or equal to 1")
        if max_token_multiplier <= 0:
            raise ValueError("has to be greater than 0")

    @property
    def max_chunker_tokens(self) -> int:
        return round(self.prompt_driver.tokenizer.max_input_tokens * self.max_token_
↪multiplier)

    @property
    def min_response_tokens(self) -> int:
        return round(
            self.prompt_driver.tokenizer.max_input_tokens
            - self.prompt_driver.tokenizer.max_input_tokens * self.max_token_
↪multiplier,
        )

    def summarize_artifacts(self, artifacts: ListArtifact, *, rulesets:␣
↪Optional[list[Ruleset]] = None) -> TextArtifact:
        return self.summarize_artifacts_rec(cast("list[TextArtifact]", artifacts.
↪value), None, rulesets=rulesets)

    def summarize_artifacts_rec(
        self,
        artifacts: list[TextArtifact],
        summary: Optional[str] = None,
        rulesets: Optional[list[Ruleset]] = None,
    ) -> TextArtifact:
        if not artifacts:
            if summary is None:
                raise ValueError("No artifacts to summarize")
            return TextArtifact(summary)

        artifacts_text = self.chunk_joiner.join([a.to_text() for a in artifacts])

        system_prompt = self.generate_system_template.render(
            summary=summary,
            rulesets=J2("rulesets/rulesets.j2").render(rulesets=rulesets),
        )

        user_prompt = self.generate_user_template.render(text=artifacts_text)
```

(continues on next page)

(continued from previous page)

```
        if (
            self.prompt_driver.tokenizer.count_input_tokens_left(user_prompt + system_
↪prompt)
            >= self.min_response_tokens
        ):
            result = self.prompt_driver.run(
                PromptStack(
                    messages=[
                        Message(system_prompt, role=Message.SYSTEM_ROLE),
                        Message(user_prompt, role=Message.USER_ROLE),
                    ],
                ),
            ).to_artifact()

            if isinstance(result, TextArtifact):
                return result
            raise ValueError("Prompt driver did not return a TextArtifact")
        chunks = self.chunker.chunk(artifacts_text)

        partial_text = self.generate_user_template.render(text=chunks[0].value)

        return self.summarize_artifacts_rec(
            chunks[1:],
            self.prompt_driver.run(
                PromptStack(
                    messages=[
                        Message(system_prompt, role=Message.SYSTEM_ROLE),
                        Message(partial_text, role=Message.USER_ROLE),
                    ],
                ),
            ).value,
            rulesets=rulesets,
        )
```

- `chunk_joiner = field(default='\n\n', kw_only=True)` class-attribute instance-attribute
- `chunker = field(default=Factory(lambda self: TextChunker(tokenizer=self.prompt_driver.tokenizer, max_tokens=self.max_chunker_tokens), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `generate_system_template = field(default=Factory(lambda: J2('engines/summary/system.j2')), kw_only=True)` class-attribute instance-attribute
- `generate_user_template = field(default=Factory(lambda: J2('engines/summary/user.j2')), kw_only=True)` class-attribute instance-attribute
- `max_chunker_tokens` property
- `max_token_multiplier = field(default=0.5, kw_only=True)` class-attribute instance-attribute
- `min_response_tokens` property
- `prompt_driver = field(default=Factory(lambda: Defaults.drivers_config.prompt_driver), kw_only=True)` class-attribute instance-attribute

summarize_artifacts(artifacts, *, rulesets=None)

Source Code in griptape/engines/summary/prompt_summary_engine.py

```
def summarize_artifacts(self, artifacts: ListArtifact, *, rulesets:␣
↪Optional[list[Ruleset]] = None) -> TextArtifact:
    return self.summarize_artifacts_rec(cast("list[TextArtifact]", artifacts.value),␣
↪None, rulesets=rulesets)
```

summarize_artifacts_rec(artifacts, summary=None, rulesets=None)

Source Code in griptape/engines/summary/prompt_summary_engine.py

```
def summarize_artifacts_rec(
    self,
    artifacts: list[TextArtifact],
    summary: Optional[str] = None,
    rulesets: Optional[list[Ruleset]] = None,
) -> TextArtifact:
    if not artifacts:
        if summary is None:
            raise ValueError("No artifacts to summarize")
        return TextArtifact(summary)

    artifacts_text = self.chunk_joiner.join([a.to_text() for a in artifacts])

    system_prompt = self.generate_system_template.render(
        summary=summary,
        rulesets=J2("rulesets/rulesets.j2").render(rulesets=rulesets),
    )

    user_prompt = self.generate_user_template.render(text=artifacts_text)

    if (
        self.prompt_driver.tokenizer.count_input_tokens_left(user_prompt + system_
↪prompt)
        >= self.min_response_tokens
    ):
        result = self.prompt_driver.run(
            PromptStack(
                messages=[
                    Message(system_prompt, role=Message.SYSTEM_ROLE),
                    Message(user_prompt, role=Message.USER_ROLE),
                ],
            ),
        ).to_artifact()

        if isinstance(result, TextArtifact):
            return result
        raise ValueError("Prompt driver did not return a TextArtifact")
    chunks = self.chunker.chunk(artifacts_text)

    partial_text = self.generate_user_template.render(text=chunks[0].value)
```

(continues on next page)

(continued from previous page)

```
    return self.summarize_artifacts_rec(
        chunks[1:],
        self.prompt_driver.run(
            PromptStack(
                messages=[
                    Message(system_prompt, role=Message.SYSTEM_ROLE),
                    Message(partial_text, role=Message.USER_ROLE),
                ],
            ),
        ).value,
        rulesets=rulesets,
    )
```

validate*allowlist(*, max_token_multiplier)

Source Code in griptape/engines/summary/prompt_summary_engine.py

```
@max_token_multiplier.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_allowlist(self, _: Attribute, max_token_multiplier: int) -> None:
    if max_token_multiplier > 1:
        raise ValueError("has to be less than or equal to 1")
    if max_token_multiplier <= 0:
        raise ValueError("has to be greater than 0")
```

RagEngine

Source Code in griptape/engines/rag/rag_engine.py

```
@define(kw_only=True)
class RagEngine:
    query_stage: Optional[QueryRagStage] = field(default=None)
    retrieval_stage: Optional[RetrievalRagStage] = field(default=None)
    response_stage: Optional[ResponseRagStage] = field(default=None)

    def __attrs_post_init__(self) -> None:
        modules = []

        if self.query_stage is not None:
            modules.extend(self.query_stage.modules)

        if self.retrieval_stage is not None:
            modules.extend(self.retrieval_stage.modules)

        if self.response_stage is not None:
            modules.extend(self.response_stage.modules)

        module_names = [m.name for m in modules]

        if len(module_names) > len(set(module_names)):
            raise ValueError("module names have to be unique")
```

(continues on next page)

(continued from previous page)

```
    def process_query(self, query: str) -> RagContext:
        return self.process(RagContext(query=query))

    def process(self, context: RagContext) -> RagContext:
        if self.query_stage:
            context = self.query_stage.run(context)

        if self.retrieval_stage:
            context = self.retrieval_stage.run(context)

        if self.response_stage:
            context = self.response_stage.run(context)

        return context
```

- `query_stage = field(default=None)` class-attribute instance-attribute
- `response_stage = field(default=None)` class-attribute instance-attribute
- `retrieval_stage = field(default=None)` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/engines/rag/rag_engine.py

```
def __attrs_post_init__(self) -> None:
    modules = []

    if self.query_stage is not None:
        modules.extend(self.query_stage.modules)

    if self.retrieval_stage is not None:
        modules.extend(self.retrieval_stage.modules)

    if self.response_stage is not None:
        modules.extend(self.response_stage.modules)

    module_names = [m.name for m in modules]

    if len(module_names) > len(set(module_names)):
        raise ValueError("module names have to be unique")
```

process(context)

Source Code in griptape/engines/rag/rag_engine.py

```
def process(self, context: RagContext) -> RagContext:
    if self.query_stage:
        context = self.query_stage.run(context)

    if self.retrieval_stage:
        context = self.retrieval_stage.run(context)
```

(continues on next page)

(continued from previous page)

```
    if self.response_stage:
        context = self.response_stage.run(context)

    return context
```

process_query(query)

Source Code in griptape/engines/rag/rag_engine.py

```
def process_query(self, query: str) -> RagContext:
    return self.process(RagContext(query=query))
```

eval

base_eval_engine

Bases:

ABC

Source Code in griptape/engines/eval/base_eval_engine.py

```
@define
class BaseEvalEngine(ABC): ...
```

eval_engine

- `RESULTS_SCHEMA = schema.Schema({'score': float, 'reason': str})` module-attribute
- `STEPS_SCHEMA = schema.Schema({'steps': [str]})` module-attribute

Bases: BaseEvalEngine “) , *SerializableMixin* ”)

Source Code in griptape/engines/eval/eval_engine.py

```
@define(kw_only=True)
class EvalEngine(BaseEvalEngine, SerializableMixin):
    id: str = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata=
↪{"serializable": True})
    name: str = field(
        default=Factory(lambda self: self.id, takes_self=True),
        metadata={"serializable": True},
    )
    criteria: Optional[str] = field(default=None, metadata={"serializable": True})
    evaluation_steps: Optional[list[str]] = field(default=None, metadata={
↪"serializable": True})
    prompt_driver: BasePromptDriver = field(default=Factory(lambda: Defaults.drivers_
↪config.prompt_driver))
    generate_steps_system_template: J2 = field(default=Factory(lambda: J2("engines/
↪eval/steps/system.j2")))
    generate_steps_user_template: J2 = field(default=Factory(lambda: J2("engines/eval/
↪steps/user.j2")))
    generate_results_system_template: J2 = field(default=Factory(lambda: J2("engines/
↪eval/results/system.j2")))
```

(continues on next page)

(continued from previous page)

```
    generate_results_user_template: J2 = field(default=Factory(lambda: J2("engines/
↪eval/results/user.j2")))

    @criteria.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_criteria(self, _: Attribute, value: Optional[str]) -> None:
        if value is None:
            if self.evaluation_steps is None:
                raise ValueError("either criteria or evaluation_steps must be␣
↪specified")
            return

        if self.evaluation_steps is not None:
            raise ValueError("cant have both criteria and evaluation_steps specified")

        if not value:
            raise ValueError("criteria must not be empty")

    @evaluation_steps.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_evaluation_steps(self, _: Attribute, value: Optional[list[str]]) ->␣
↪None:
        if value is None:
            if self.criteria is None:
                raise ValueError("either evaluation_steps or criteria must be␣
↪specified")
            return

        if self.criteria is not None:
            raise ValueError("cant have both evaluation_steps and criteria specified")

        if not value:
            raise ValueError("evaluation_steps must not be empty")

    def evaluate(self, input: str, actual_output: str, **kwargs) -> tuple[float,␣
↪str]:  # noqa: A002
        evaluation_params = {
            key.replace("_", " ").title(): value
            for key, value in {"input": input, "actual_output": actual_output,␣
↪**kwargs}.items()
        }

        if self.evaluation_steps is None:
            # Need to disable validators to allow for both `criteria` and `evaluation_
↪steps` to be set
            with validators.disabled():
                self.evaluation_steps = self._generate_steps(evaluation_params)

        return self._generate_results(evaluation_params)

    def _generate_steps(self, evaluation_params: dict[str, str]) -> list[str]:
        system_prompt = self.generate_steps_system_template.render(
```

(continues on next page)

(continued from previous page)

```
            evaluation_params=", ".join(param for param in evaluation_params),
            criteria=self.criteria,
        )
        user_prompt = self.generate_steps_user_template.render()

        result = self.prompt_driver.run(
            PromptStack(
                messages=[
                    Message(system_prompt, role=Message.SYSTEM_ROLE),
                    Message(user_prompt, role=Message.USER_ROLE),
                ],
                output_schema=STEPS_SCHEMA,
            ),
        ).to_artifact()

        parsed_result = json.loads(result.value)

        return parsed_result["steps"]

    def _generate_results(self, evaluation_params: dict[str, str]) -> tuple[float,␣
↪str]:
        system_prompt = self.generate_results_system_template.render(
            evaluation_params=", ".join(param for param in evaluation_params),
            evaluation_steps=self.evaluation_steps,
            evaluation_text="\n\n".join(f"{key}: {value}" for key, value in␣
↪evaluation_params.items()),
        )
        user_prompt = self.generate_results_user_template.render()

        result = self.prompt_driver.run(
            PromptStack(
                messages=[
                    Message(system_prompt, role=Message.SYSTEM_ROLE),
                    Message(user_prompt, role=Message.USER_ROLE),
                ],
                output_schema=RESULTS_SCHEMA,
            ),
        ).to_text()

        parsed_result = json.loads(result)

        # Better to have the LLM deal strictly with integers to avoid ambiguities␣
↪with floating point precision.
        # We want the user to receive a float, however.
        score = float(parsed_result["score"]) / 10
        reason = parsed_result["reason"]

        return score, reason
```

- `criteria = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `evaluation_steps = field(default=None, metadata={'serializable': True})` class-attribute

instance-attribute

- `generate_results_system_template = field(default=Factory(lambda: J2('engines/eval/results/system.j2')))` class-attribute instance-attribute
- `generate_results_user_template = field(default=Factory(lambda: J2('engines/eval/results/user.j2')))` class-attribute instance-attribute
- `generate_steps_system_template = field(default=Factory(lambda: J2('engines/eval/steps/system.j2')))` class-attribute instance-attribute
- `generate_steps_user_template = field(default=Factory(lambda: J2('engines/eval/steps/user.j2')))` class-attribute instance-attribute
- `id = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `name = field(default=Factory(lambda self: self.id, takes_self=True), metadata={'serializable': True})` class-attribute instance-attribute
- `prompt_driver = field(default=Factory(lambda: Defaults.drivers_config.prompt_driver))` class-attribute instance-attribute

_generate_results(evaluation_params)

Source Code in griptape/engines/eval/eval_engine.py

```
def _generate_results(self, evaluation_params: dict[str, str]) -> tuple[float, str]:
    system_prompt = self.generate_results_system_template.render(
        evaluation_params=", ".join(param for param in evaluation_params),
        evaluation_steps=self.evaluation_steps,
        evaluation_text="\n\n".join(f"{key}: {value}" for key, value in evaluation_
↪params.items()),
    )
    user_prompt = self.generate_results_user_template.render()

    result = self.prompt_driver.run(
        PromptStack(
            messages=[
                Message(system_prompt, role=Message.SYSTEM_ROLE),
                Message(user_prompt, role=Message.USER_ROLE),
            ],
            output_schema=RESULTS_SCHEMA,
        ),
    ).to_text()

    parsed_result = json.loads(result)

    # Better to have the LLM deal strictly with integers to avoid ambiguities with␣
↪floating point precision.
    # We want the user to receive a float, however.
    score = float(parsed_result["score"]) / 10
    reason = parsed_result["reason"]

    return score, reason
```

_generate_steps(evaluation_params)

Source Code in griptape/engines/eval/eval_engine.py

```
def _generate_steps(self, evaluation_params: dict[str, str]) -> list[str]:
    system_prompt = self.generate_steps_system_template.render(
        evaluation_params=", ".join(param for param in evaluation_params),
        criteria=self.criteria,
    )
    user_prompt = self.generate_steps_user_template.render()

    result = self.prompt_driver.run(
        PromptStack(
            messages=[
                Message(system_prompt, role=Message.SYSTEM_ROLE),
                Message(user_prompt, role=Message.USER_ROLE),
            ],
            output_schema=STEPS_SCHEMA,
        ),
    ).to_artifact()

    parsed_result = json.loads(result.value)

    return parsed_result["steps"]
```

evaluate(input, actual_output, **kwargs)

Source Code in griptape/engines/eval/eval_engine.py

```
def evaluate(self, input: str, actual_output: str, **kwargs) -> tuple[float, str]:  #␣
↪noqa: A002
    evaluation_params = {
        key.replace("_", " ").title(): value
        for key, value in {"input": input, "actual_output": actual_output, **kwargs}.
↪items()
    }

    if self.evaluation_steps is None:
        # Need to disable validators to allow for both `criteria` and `evaluation_
↪steps` to be set
        with validators.disabled():
            self.evaluation_steps = self._generate_steps(evaluation_params)

    return self._generate_results(evaluation_params)
```

validate*criteria(*, value)

Source Code in griptape/engines/eval/eval_engine.py

```
@criteria.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_criteria(self, _: Attribute, value: Optional[str]) -> None:
    if value is None:
```

(continues on next page)

(continued from previous page)

```
        if self.evaluation_steps is None:
            raise ValueError("either criteria or evaluation_steps must be specified")
        return

    if self.evaluation_steps is not None:
        raise ValueError("cant have both criteria and evaluation_steps specified")

    if not value:
        raise ValueError("criteria must not be empty")
```

validate*evaluation_steps(*, value)

Source Code in griptape/engines/eval/eval_engine.py

```
@evaluation_steps.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_evaluation_steps(self, _: Attribute, value: Optional[list[str]]) -> None:
    if value is None:
        if self.criteria is None:
            raise ValueError("either evaluation_steps or criteria must be specified")
        return

    if self.criteria is not None:
        raise ValueError("cant have both evaluation_steps and criteria specified")

    if not value:
        raise ValueError("evaluation_steps must not be empty")
```

extraction

base_extraction_engine

Bases:

ABC

Source Code in griptape/engines/extraction/base_extraction_engine.py

```
@define
class BaseExtractionEngine(ABC):
    max_token_multiplier: float = field(default=0.5, kw_only=True)
    chunk_joiner: str = field(default="\n\n", kw_only=True)
    prompt_driver: BasePromptDriver = field(
        default=Factory(lambda: Defaults.drivers_config.prompt_driver), kw_only=True
    )
    chunker: BaseChunker = field(
        default=Factory(
            lambda self: TextChunker(tokenizer=self.prompt_driver.tokenizer, max_
↪tokens=self.max_chunker_tokens),
            takes_self=True,
        ),
        kw_only=True,
```

(continues on next page)

(continued from previous page)

```
    )

    @max_token_multiplier.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_max_token_multiplier(self, _: Attribute, max_token_multiplier: int) -
↪> None:
        if max_token_multiplier > 1:
            raise ValueError("has to be less than or equal to 1")
        if max_token_multiplier <= 0:
            raise ValueError("has to be greater than 0")

    @property
    def max_chunker_tokens(self) -> int:
        return round(self.prompt_driver.tokenizer.max_input_tokens * self.max_token_
↪multiplier)

    @property
    def min_response_tokens(self) -> int:
        return round(
            self.prompt_driver.tokenizer.max_input_tokens
            - self.prompt_driver.tokenizer.max_input_tokens * self.max_token_
↪multiplier,
        )

    def extract_text(
        self,
        text: str,
        *,
        rulesets: Optional[list[Ruleset]] = None,
        **kwargs,
    ) -> ListArtifact:
        return self.extract_artifacts(ListArtifact([TextArtifact(text)]),␣
↪rulesets=rulesets, **kwargs)

    @abstractmethod
    def extract_artifacts(
        self,
        artifacts: ListArtifact[TextArtifact],
        *,
        rulesets: Optional[list[Ruleset]] = None,
        **kwargs,
    ) -> ListArtifact: ...
```

- `chunk_joiner = field(default='\n\n', kw_only=True)` class-attribute instance-attribute
- `chunker = field(default=Factory(lambda self: TextChunker(tokenizer=self.prompt_driver.tokenizer, max_tokens=self.max_chunker_tokens), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `max_chunker_tokens` property
- `max_token_multiplier = field(default=0.5, kw_only=True)` class-attribute instance-attribute
- `min_response_tokens` property
- `prompt_driver = field(default=Factory(lambda: Defaults.drivers_config.`

`prompt_driver), kw_only=True)` class-attribute instance-attribute

extract_artifacts(artifacts, *, rulesets=None, **kwargs)abstractmethod

Source Code in griptape/engines/extraction/base_extraction_engine.py

```
@abstractmethod
def extract_artifacts(
    self,
    artifacts: ListArtifact[TextArtifact],
    *,
    rulesets: Optional[list[Ruleset]] = None,
    **kwargs,
) -> ListArtifact: ...
```

extract_text(text, *, rulesets=None, **kwargs)

Source Code in griptape/engines/extraction/base_extraction_engine.py

```
def extract_text(
    self,
    text: str,
    *,
    rulesets: Optional[list[Ruleset]] = None,
    **kwargs,
) -> ListArtifact:
    return self.extract_artifacts(ListArtifact([TextArtifact(text)]),␣
↪rulesets=rulesets, **kwargs)
```

validate*max_token_multiplier(*, max_token_multiplier)

Source Code in griptape/engines/extraction/base_extraction_engine.py

```
@max_token_multiplier.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_max_token_multiplier(self, _: Attribute, max_token_multiplier: int) ->␣
↪None:
    if max_token_multiplier > 1:
        raise ValueError("has to be less than or equal to 1")
    if max_token_multiplier <= 0:
        raise ValueError("has to be greater than 0")
```

csv_extraction_engine

Bases: *BaseExtractionEngine* “)

Source Code in griptape/engines/extraction/csv_extraction_engine.py

```
@define
class CsvExtractionEngine(BaseExtractionEngine):
    column_names: list[str] = field(kw_only=True)
    generate_system_template: J2 = field(default=Factory(lambda: J2("engines/
↪extraction/csv/system.j2")), kw_only=True)
    generate_user_template: J2 = field(default=Factory(lambda: J2("engines/extraction/
```

(continues on next page)

(continued from previous page)

```
↪csv/user.j2")), kw_only=True)
    format_header: Callable[[list[str]], str] = field(
        default=Factory(lambda: lambda value: ",".join(value)), kw_only=True
    )
    format_row: Callable[[dict], str] = field(
        default=Factory(lambda: lambda value: ",".join([value or "" for value in␣
↪value.values()])), kw_only=True
    )

    def extract_artifacts(
        self,
        artifacts: ListArtifact[TextArtifact],
        *,
        rulesets: Optional[list[Ruleset]] = None,
        **kwargs,
    ) -> ListArtifact[TextArtifact]:
        return ListArtifact(
            self._extract_rec(
                cast("list[TextArtifact]", artifacts.value),
                [TextArtifact(self.format_header(self.column_names))],
                rulesets=rulesets,
            ),
            item_separator="\n",
        )

    def text_to_csv_rows(self, text: str) -> list[TextArtifact]:
        rows = []

        with io.StringIO(text) as f:
            for row in csv.DictReader(f):
                rows.append(TextArtifact(self.format_row(row)))

        return rows

    def _extract_rec(
        self,
        artifacts: list[TextArtifact],
        rows: list[TextArtifact],
        *,
        rulesets: Optional[list[Ruleset]] = None,
    ) -> list[TextArtifact]:
        artifacts_text = self.chunk_joiner.join([a.value for a in artifacts])
        system_prompt = self.generate_system_template.render(
            column_names=self.column_names,
            rulesets=J2("rulesets/rulesets.j2").render(rulesets=rulesets),
        )
        user_prompt = self.generate_user_template.render(
            text=artifacts_text,
        )

        if (
            self.prompt_driver.tokenizer.count_input_tokens_left(system_prompt + user_
```

(continues on next page)

(continued from previous page)

```
↪prompt)
            >= self.min_response_tokens
        ):
            rows.extend(
                self.text_to_csv_rows(
                    self.prompt_driver.run(
                        PromptStack(
                            messages=[
                                Message(system_prompt, role=Message.SYSTEM_ROLE),
                                Message(user_prompt, role=Message.USER_ROLE),
                            ]
                        )
                    ).value,
                ),
            )

            return rows
        chunks = self.chunker.chunk(artifacts_text)
        partial_text = self.generate_user_template.render(
            text=chunks[0].value,
        )

        rows.extend(
            self.text_to_csv_rows(
                self.prompt_driver.run(
                    PromptStack(
                        messages=[
                            Message(system_prompt, role=Message.SYSTEM_ROLE),
                            Message(partial_text, role=Message.USER_ROLE),
                        ]
                    )
                ).value,
            ),
        )

        return self._extract_rec(chunks[1:], rows, rulesets=rulesets)
```

- `column_names = field(kw_only=True)` class-attribute instance-attribute
- `format_header = field(default=Factory(lambda: lambda value: ','.join(value)), kw_only=True)` class-attribute instance-attribute
- `format_row = field(default=Factory(lambda: lambda value: ','.join([value or '' for value in value.values()])), kw_only=True)` class-attribute instance-attribute
- `generate_system_template = field(default=Factory(lambda: J2('engines/extraction/csv/system.j2')), kw_only=True)` class-attribute instance-attribute
- `generate_user_template = field(default=Factory(lambda: J2('engines/extraction/csv/user.j2')), kw_only=True)` class-attribute instance-attribute

_extract_rec(artifacts, rows, *, rulesets=None)

Source Code in griptape/engines/extraction/csv_extraction_engine.py

```
def _extract_rec(
    self,
    artifacts: list[TextArtifact],
    rows: list[TextArtifact],
    *,
    rulesets: Optional[list[Ruleset]] = None,
) -> list[TextArtifact]:
    artifacts_text = self.chunk_joiner.join([a.value for a in artifacts])
    system_prompt = self.generate_system_template.render(
        column_names=self.column_names,
        rulesets=J2("rulesets/rulesets.j2").render(rulesets=rulesets),
    )
    user_prompt = self.generate_user_template.render(
        text=artifacts_text,
    )

    if (
        self.prompt_driver.tokenizer.count_input_tokens_left(system_prompt + user_
↪prompt)
        >= self.min_response_tokens
    ):
        rows.extend(
            self.text_to_csv_rows(
                self.prompt_driver.run(
                    PromptStack(
                        messages=[
                            Message(system_prompt, role=Message.SYSTEM_ROLE),
                            Message(user_prompt, role=Message.USER_ROLE),
                        ]
                    )
                ).value,
            ),
        )

        return rows
    chunks = self.chunker.chunk(artifacts_text)
    partial_text = self.generate_user_template.render(
        text=chunks[0].value,
    )

    rows.extend(
        self.text_to_csv_rows(
            self.prompt_driver.run(
                PromptStack(
                    messages=[
                        Message(system_prompt, role=Message.SYSTEM_ROLE),
                        Message(partial_text, role=Message.USER_ROLE),
                    ]
                )
            ).value,
```

(continues on next page)

(continued from previous page)

```
        ),
    )

    return self._extract_rec(chunks[1:], rows, rulesets=rulesets)
```

extract_artifacts(artifacts, *, rulesets=None, **kwargs)

Source Code in griptape/engines/extraction/csv_extraction_engine.py

```
def extract_artifacts(
    self,
    artifacts: ListArtifact[TextArtifact],
    *,
    rulesets: Optional[list[Ruleset]] = None,
    **kwargs,
) -> ListArtifact[TextArtifact]:
    return ListArtifact(
        self._extract_rec(
            cast("list[TextArtifact]", artifacts.value),
            [TextArtifact(self.format_header(self.column_names))],
            rulesets=rulesets,
        ),
        item_separator="\n",
    )
```

text_to_csv_rows(text)

Source Code in griptape/engines/extraction/csv_extraction_engine.py

```
def text_to_csv_rows(self, text: str) -> list[TextArtifact]:
    rows = []

    with io.StringIO(text) as f:
        for row in csv.DictReader(f):
            rows.append(TextArtifact(self.format_row(row)))

    return rows
```

json_extraction_engine

Bases: *BaseExtractionEngine* “)

Source Code in griptape/engines/extraction/json_extraction_engine.py

```
@define
class JsonExtractionEngine(BaseExtractionEngine):
    JSON_PATTERN = r"(?s)[^\[]*(\[.*\])"

    template_schema: dict = field(kw_only=True)
    generate_system_template: J2 = field(default=Factory(lambda: J2("engines/
↪extraction/json/system.j2")), kw_only=True)
```

(continues on next page)

(continued from previous page)

```
    generate_user_template: J2 = field(default=Factory(lambda: J2("engines/extraction/
↪json/user.j2")), kw_only=True)

    def extract_artifacts(
        self,
        artifacts: ListArtifact[TextArtifact],
        *,
        rulesets: Optional[list[Ruleset]] = None,
        **kwargs,
    ) -> ListArtifact[JsonArtifact]:
        return ListArtifact(
            self._extract_rec(cast("list[TextArtifact]", artifacts.value), [],␣
↪rulesets=rulesets),
            item_separator="\n",
        )

    def json_to_text_artifacts(self, json_input: str) -> list[JsonArtifact]:
        json_matches = re.findall(self.JSON_PATTERN, json_input, re.DOTALL)

        if json_matches:
            return [JsonArtifact(e) for e in json.loads(json_matches[-1])]
        return []

    def _extract_rec(
        self,
        artifacts: list[TextArtifact],
        extractions: list[JsonArtifact],
        *,
        rulesets: Optional[list[Ruleset]] = None,
    ) -> list[JsonArtifact]:
        artifacts_text = self.chunk_joiner.join([a.value for a in artifacts])
        system_prompt = self.generate_system_template.render(
            json_template_schema=json.dumps(self.template_schema),
            rulesets=J2("rulesets/rulesets.j2").render(rulesets=rulesets),
        )
        user_prompt = self.generate_user_template.render(
            text=artifacts_text,
        )

        if (
            self.prompt_driver.tokenizer.count_input_tokens_left(user_prompt + system_
↪prompt)
            >= self.min_response_tokens
        ):
            extractions.extend(
                self.json_to_text_artifacts(
                    self.prompt_driver.run(
                        PromptStack(
                            messages=[
                                Message(system_prompt, role=Message.SYSTEM_ROLE),
                                Message(user_prompt, role=Message.USER_ROLE),
                            ]
```

(continues on next page)

(continued from previous page)

```
                        )
                    ).value
                ),
            )

            return extractions
        chunks = self.chunker.chunk(artifacts_text)
        partial_text = self.generate_user_template.render(
            text=chunks[0].value,
        )

        extractions.extend(
            self.json_to_text_artifacts(
                self.prompt_driver.run(
                    PromptStack(
                        messages=[
                            Message(system_prompt, role=Message.SYSTEM_ROLE),
                            Message(partial_text, role=Message.USER_ROLE),
                        ]
                    )
                ).value,
            ),
        )

        return self._extract_rec(chunks[1:], extractions, rulesets=rulesets)
```

- `JSON_PATTERN = '(?s)[^\\[]*(\\[.*\\])'` class-attribute instance-attribute
- `generate_system_template = field(default=Factory(lambda: J2('engines/extraction/json/system.j2')), kw_only=True)` class-attribute instance-attribute
- `generate_user_template = field(default=Factory(lambda: J2('engines/extraction/json/user.j2')), kw_only=True)` class-attribute instance-attribute
- `template_schema = field(kw_only=True)` class-attribute instance-attribute

_extract_rec(artifacts, extractions, *, rulesets=None)

Source Code in griptape/engines/extraction/json_extraction_engine.py

```
def _extract_rec(
    self,
    artifacts: list[TextArtifact],
    extractions: list[JsonArtifact],
    *,
    rulesets: Optional[list[Ruleset]] = None,
) -> list[JsonArtifact]:
    artifacts_text = self.chunk_joiner.join([a.value for a in artifacts])
    system_prompt = self.generate_system_template.render(
        json_template_schema=json.dumps(self.template_schema),
        rulesets=J2("rulesets/rulesets.j2").render(rulesets=rulesets),
    )
    user_prompt = self.generate_user_template.render(
        text=artifacts_text,
```

(continues on next page)

(continued from previous page)

```
    )

    if (
        self.prompt_driver.tokenizer.count_input_tokens_left(user_prompt + system_
↪prompt)
        >= self.min_response_tokens
    ):
        extractions.extend(
            self.json_to_text_artifacts(
                self.prompt_driver.run(
                    PromptStack(
                        messages=[
                            Message(system_prompt, role=Message.SYSTEM_ROLE),
                            Message(user_prompt, role=Message.USER_ROLE),
                        ]
                    )
                ).value
            ),
        )

        return extractions
    chunks = self.chunker.chunk(artifacts_text)
    partial_text = self.generate_user_template.render(
        text=chunks[0].value,
    )

    extractions.extend(
        self.json_to_text_artifacts(
            self.prompt_driver.run(
                PromptStack(
                    messages=[
                        Message(system_prompt, role=Message.SYSTEM_ROLE),
                        Message(partial_text, role=Message.USER_ROLE),
                    ]
                )
            ).value,
        ),
    )

    return self._extract_rec(chunks[1:], extractions, rulesets=rulesets)
```

extract_artifacts(artifacts, *, rulesets=None, **kwargs)

Source Code in griptape/engines/extraction/json_extraction_engine.py

```
def extract_artifacts(
    self,
    artifacts: ListArtifact[TextArtifact],
    *,
    rulesets: Optional[list[Ruleset]] = None,
    **kwargs,
) -> ListArtifact[JsonArtifact]:
```

(continues on next page)

(continued from previous page)

```
    return ListArtifact(
        self._extract_rec(cast("list[TextArtifact]", artifacts.value), [],␣
↪rulesets=rulesets),
        item_separator="\n",
    )
```

json_to_text_artifacts(json_input)

Source Code in griptape/engines/extraction/json_extraction_engine.py

```
def json_to_text_artifacts(self, json_input: str) -> list[JsonArtifact]:
    json_matches = re.findall(self.JSON_PATTERN, json_input, re.DOTALL)

    if json_matches:
        return [JsonArtifact(e) for e in json.loads(json_matches[-1])]
    return []
```

rag

- `__all__ = ['RagContext', 'RagEngine']` module-attribute

Bases: *SerializableMixin* “)

Attributes

Name	Type	Description
<code class="language-python">	*str*	Query provided by the user.
<code class="language-python">	*dict[str, dict]*	Dictionary of module configs. First key should be a module name and the second a dictionary of configs parameters.

continues on next page

Table 316 – continued from previous page

Name	Type	Description
<code class="language-python">	<code class="language-python">list[str]</code>	An optional list of strings to add before the query in response modules.
<code class="language-python">	<code class="language-python">list[str]</code>	An optional list of strings to add after the query in response modules.
<a href="./#griptape.engines.rag.RagContext.text_chunks" title="text_chunks = field(factory=list, metadata={'serializable': True}) class-attribute instance-attribute (griptape.engines.rag.RagContext.text_chunks)"><code class="language-python">	<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">list[TextArtifact]</code></a	A list of text chunks to pass around from the retrieval stage to the response stage.
<a href="./#griptape.engines.rag.RagContext.outputs" title="outputs = field(factory=list, metadata={'serializable': True}) class-attribute instance-attribute (griptape.engines.rag.RagContext.outputs)"><code class="language-python">	<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">list[BaseArtifact]</code></a	List of outputs from the response stage.

Source Code in griptape/engines/rag/rag_context.py

```
@define(kw_only=True)
class RagContext(SerializableMixin):
    """Used by RagEngine stages and module to pass context that individual modules
↪are expected to update in the `run` method.

    Attributes:
        query: Query provided by the user.
        module_configs: Dictionary of module configs. First key should be a module
↪name and the second a dictionary of configs parameters.
        before_query: An optional list of strings to add before the query in response
↪modules.
        after_query: An optional list of strings to add after the query in response
↪modules.
        text_chunks: A list of text chunks to pass around from the retrieval stage to
↪the response stage.
        outputs: List of outputs from the response stage.
    """

    query: str = field(metadata={"serializable": True})
    module_configs: dict[str, dict] = field(factory=dict, metadata={"serializable":
↪True})
    before_query: list[str] = field(factory=list, metadata={"serializable": True})
    after_query: list[str] = field(factory=list, metadata={"serializable": True})
    text_chunks: list[TextArtifact] = field(factory=list, metadata={"serializable":
↪True})
    outputs: list[BaseArtifact] = field(factory=list, metadata={"serializable": True})

    def get_references(self) -> list[Reference]:
        return utils.references_from_artifacts(self.text_chunks)
```

- `after_query = field(factory=list, metadata={'serializable': True})` class-attribute instance-attribute
- `before_query = field(factory=list, metadata={'serializable': True})` class-attribute instance-attribute
- `module_configs = field(factory=dict, metadata={'serializable': True})` class-attribute instance-attribute
- `outputs = field(factory=list, metadata={'serializable': True})` class-attribute instance-attribute
- `query = field(metadata={'serializable': True})` class-attribute instance-attribute
- `text_chunks = field(factory=list, metadata={'serializable': True})` class-attribute instance-attribute

get_references()

Source Code in griptape/engines/rag/rag_context.py

```
def get_references(self) -> list[Reference]:
    return utils.references_from_artifacts(self.text_chunks)
```

RagEngine

Source Code in griptape/engines/rag/rag_engine.py

```
@define(kw_only=True)
class RagEngine:
    query_stage: Optional[QueryRagStage] = field(default=None)
    retrieval_stage: Optional[RetrievalRagStage] = field(default=None)
    response_stage: Optional[ResponseRagStage] = field(default=None)

    def __attrs_post_init__(self) -> None:
        modules = []

        if self.query_stage is not None:
            modules.extend(self.query_stage.modules)

        if self.retrieval_stage is not None:
            modules.extend(self.retrieval_stage.modules)

        if self.response_stage is not None:
            modules.extend(self.response_stage.modules)

        module_names = [m.name for m in modules]

        if len(module_names) > len(set(module_names)):
            raise ValueError("module names have to be unique")

    def process_query(self, query: str) -> RagContext:
        return self.process(RagContext(query=query))

    def process(self, context: RagContext) -> RagContext:
        if self.query_stage:
            context = self.query_stage.run(context)

        if self.retrieval_stage:
            context = self.retrieval_stage.run(context)

        if self.response_stage:
            context = self.response_stage.run(context)

        return context
```

- `query_stage = field(default=None)` class-attribute instance-attribute
- `response_stage = field(default=None)` class-attribute instance-attribute
- `retrieval_stage = field(default=None)` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/engines/rag/rag_engine.py

```
def __attrs_post_init__(self) -> None:
    modules = []
```

(continues on next page)

(continued from previous page)

```
    if self.query_stage is not None:
        modules.extend(self.query_stage.modules)

    if self.retrieval_stage is not None:
        modules.extend(self.retrieval_stage.modules)

    if self.response_stage is not None:
        modules.extend(self.response_stage.modules)

    module_names = [m.name for m in modules]

    if len(module_names) > len(set(module_names)):
        raise ValueError("module names have to be unique")
```

process(context)

Source Code in griptape/engines/rag/rag_engine.py

```
def process(self, context: RagContext) -> RagContext:
    if self.query_stage:
        context = self.query_stage.run(context)

    if self.retrieval_stage:
        context = self.retrieval_stage.run(context)

    if self.response_stage:
        context = self.response_stage.run(context)

    return context
```

process_query(query)

Source Code in griptape/engines/rag/rag_engine.py

```
def process_query(self, query: str) -> RagContext:
    return self.process(RagContext(query=query))
```

modules

- `__all__ = ['BaseAfterResponseRagModule', 'BaseBeforeResponseRagModule', 'BaseQueryRagModule', 'BaseRagModule', 'BaseRerankRagModule', 'BaseResponseRagModule', 'BaseRetrievalRagModule', 'FootnotePromptResponseRagModule', 'PromptResponseRagModule', 'TextChunksRerankRagModule', 'TextChunksResponseRagModule', 'TextLoaderRetrievalRagModule', 'TranslateQueryRagModule', 'VectorStoreRetrievalRagModule']` module-attribute

Bases: *BaseRagModule* “) , ABC

Source Code in griptape/engines/rag/modules/response/base_after_response_rag_module.py

```
@define(kw_only=True)
class BaseAfterResponseRagModule(BaseRagModule, ABC):
```

(continues on next page)

(continued from previous page)

```
    @abstractmethod
    def run(self, context: RagContext) -> RagContext: ...
```

run(context)abstractmethod

Source Code in griptape/engines/rag/modules/response/base_after_response_rag_module.py

```
@abstractmethod
def run(self, context: RagContext) -> RagContext: ...
```

BaseBeforeResponseRagModule

Bases: *BaseRagModule* “) , ABC

Source Code in griptape/engines/rag/modules/response/base_before_response_rag_module.py

```
@define(kw_only=True)
class BaseBeforeResponseRagModule(BaseRagModule, ABC):
    @abstractmethod
    def run(self, context: RagContext) -> RagContext: ...
```

run(context)abstractmethod

Source Code in griptape/engines/rag/modules/response/base_before_response_rag_module.py

```
@abstractmethod
def run(self, context: RagContext) -> RagContext: ...
```

BaseQueryRagModule

Bases: *BaseRagModule* “) , ABC

Source Code in griptape/engines/rag/modules/query/base_query_rag_module.py

```
@define(kw_only=True)
class BaseQueryRagModule(BaseRagModule, ABC):
    @abstractmethod
    def run(self, context: RagContext) -> RagContext: ...
```

run(context)abstractmethod

Source Code in griptape/engines/rag/modules/query/base_query_rag_module.py

```
@abstractmethod
def run(self, context: RagContext) -> RagContext: ...
```

BaseRagModule

Bases: *FuturesExecutorMixin* “) , ABC

Source Code in griptape/engines/rag/modules/base_rag_module.py

```
@define(kw_only=True)
class BaseRagModule(FuturesExecutorMixin, ABC):
    name: str = field(
        default=Factory(lambda self: f"{self.__class__.__name__}-{uuid.uuid4().hex}",␣
↪takes_self=True), kw_only=True
    )

    def generate_prompt_stack(self, system_prompt: Optional[str], query: str) ->␣
↪PromptStack:
        messages = []

        if system_prompt is not None:
            messages.append(Message(system_prompt, role=Message.SYSTEM_ROLE))

        messages.append(Message(query, role=Message.USER_ROLE))

        return PromptStack(messages=messages)

    def get_context_param(self, context: RagContext, key: str) -> Optional[Any]:
        return context.module_configs.get(self.name, {}).get(key)

    def set_context_param(self, context: RagContext, key: str, value: Any) -> None:
        if not isinstance(context.module_configs.get(self.name), dict):
            context.module_configs[self.name] = {}

        context.module_configs[self.name][key] = value
```

- `name = field(default=Factory(lambda self: f'{self.__class__.__name__}-{uuid.uuid4().hex}', takes_self=True), kw_only=True)` class-attribute instance-attribute

generate_prompt_stack(system_prompt, query)

Source Code in griptape/engines/rag/modules/base_rag_module.py

```
def generate_prompt_stack(self, system_prompt: Optional[str], query: str) ->␣
↪PromptStack:
    messages = []

    if system_prompt is not None:
        messages.append(Message(system_prompt, role=Message.SYSTEM_ROLE))

    messages.append(Message(query, role=Message.USER_ROLE))

    return PromptStack(messages=messages)
```

get_context_param(context, key)

Source Code in griptape/engines/rag/modules/base_rag_module.py

```
def get_context_param(self, context: RagContext, key: str) -> Optional[Any]:
    return context.module_configs.get(self.name, {}).get(key)
```

set_context_param(context, key, value)

Source Code in griptape/engines/rag/modules/base_rag_module.py

```
def set_context_param(self, context: RagContext, key: str, value: Any) -> None:
    if not isinstance(context.module_configs.get(self.name), dict):
        context.module_configs[self.name] = {}

    context.module_configs[self.name][key] = value
```

BaseRerankRagModule

Bases: *BaseRagModule* “) , ABC

Source Code in griptape/engines/rag/modules/retrieval/base_rerank_rag_module.py

```
@define(kw_only=True)
class BaseRerankRagModule(BaseRagModule, ABC):
    rerank_driver: BaseRerankDriver = field()

    @abstractmethod
    def run(self, context: RagContext) -> Sequence[BaseArtifact]: ...
```

- `rerank_driver = field()` class-attribute instance-attribute

run(context)abstractmethod

Source Code in griptape/engines/rag/modules/retrieval/base_rerank_rag_module.py

```
@abstractmethod
def run(self, context: RagContext) -> Sequence[BaseArtifact]: ...
```

BaseResponseRagModule

Bases: *BaseRagModule* “) , ABC

Source Code in griptape/engines/rag/modules/response/base_response_rag_module.py

```
@define(kw_only=True)
class BaseResponseRagModule(BaseRagModule, ABC):
    @abstractmethod
    def run(self, context: RagContext) -> BaseArtifact: ...
```

run(context)abstractmethod

Source Code in griptape/engines/rag/modules/response/base_response_rag_module.py

```
@abstractmethod
def run(self, context: RagContext) -> BaseArtifact: ...
```

BaseRetrievalRagModule

Bases: *BaseRagModule* “) , ABC

Source Code in griptape/engines/rag/modules/retrieval/base_retrieval_rag_module.py

```
@define(kw_only=True)
class BaseRetrievalRagModule(BaseRagModule, ABC):
    @abstractmethod
    def run(self, context: RagContext) -> Sequence[BaseArtifact]: ...
```

run(context)abstractmethod

Source Code in griptape/engines/rag/modules/retrieval/base_retrieval_rag_module.py

```
@abstractmethod
def run(self, context: RagContext) -> Sequence[BaseArtifact]: ...
```

FootnotePromptResponseRagModule

Bases: *FootnotePromptResponseRagModule* “)

Source Code in griptape/engines/rag/modules/response/footnote_prompt_response_rag_module.py

```
@define(kw_only=True)
class FootnotePromptResponseRagModule(PromptResponseRagModule):
    def default_generate_system_template(self, context: RagContext, artifacts:␣
↪list[TextArtifact]) -> str:
        return J2("engines/rag/modules/response/footnote_prompt/system.j2").render(
            text_chunk_artifacts=artifacts,
            references=utils.references_from_artifacts(artifacts),
            before_system_prompt="\n\n".join(context.before_query),
            after_system_prompt="\n\n".join(context.after_query),
        )
```

default_generate_system_template(context, artifacts)

Source Code in griptape/engines/rag/modules/response/footnote_prompt_response_rag_module.py

```
def default_generate_system_template(self, context: RagContext, artifacts:␣
↪list[TextArtifact]) -> str:
    return J2("engines/rag/modules/response/footnote_prompt/system.j2").render(
        text_chunk_artifacts=artifacts,
        references=utils.references_from_artifacts(artifacts),
        before_system_prompt="\n\n".join(context.before_query),
        after_system_prompt="\n\n".join(context.after_query),
    )
```

PromptResponseRagModule

Bases: *BaseResponseRagModule* “) , *RuleMixin* ”)

Source Code in griptape/engines/rag/modules/response/prompt_response_rag_module.py

```
@define(kw_only=True)
class PromptResponseRagModule(BaseResponseRagModule, RuleMixin):
    prompt_driver: BasePromptDriver = field(default=Factory(lambda: Defaults.drivers_
↪config.prompt_driver))
    answer_token_offset: int = field(default=400)
    metadata: Optional[str] = field(default=None)
    generate_system_template: Callable[[RagContext, list[TextArtifact]], str] = field(
        default=Factory(lambda self: self.default_generate_system_template, takes_
↪self=True),
    )

    def run(self, context: RagContext) -> BaseArtifact:
        query = context.query
        tokenizer = self.prompt_driver.tokenizer
        included_chunks = []
        system_prompt = self.generate_system_template(context, included_chunks)

        for artifact in context.text_chunks:
            included_chunks.append(artifact)

            system_prompt = self.generate_system_template(context, included_chunks)
            message_token_count = self.prompt_driver.tokenizer.count_tokens(
                self.prompt_driver.prompt_stack_to_string(self.generate_prompt_
↪stack(system_prompt, query)),
            )

            if message_token_count + self.answer_token_offset >= tokenizer.max_input_
↪tokens:
                included_chunks.pop()

                system_prompt = self.generate_system_template(context, included_
↪chunks)

                break

        output = self.prompt_driver.run(self.generate_prompt_stack(system_prompt,␣
↪query)).to_artifact()

        if isinstance(output, TextArtifact):
            return output
        raise ValueError("Prompt driver did not return a TextArtifact")

    def default_generate_system_template(self, context: RagContext, artifacts:␣
↪list[TextArtifact]) -> str:
        params: dict[str, Any] = {"text_chunks": [c.to_text() for c in artifacts]}

        if len(self.rulesets) > 0:
            params["rulesets"] = J2("rulesets/rulesets.j2").render(rulesets=self.
↪rulesets)

        if self.metadata is not None:
            params["metadata"] = J2("engines/rag/modules/response/metadata/system.j2
↪").render(metadata=self.metadata)
```

(continues on next page)

(continued from previous page)

```
        return J2("engines/rag/modules/response/prompt/system.j2").render(**params)
```

- `answer_token_offset = field(default=400)` class-attribute instance-attribute
- `generate_system_template = field(default=Factory(lambda self: self.default_generate_system_template, takes_self=True))` class-attribute instance-attribute
- `metadata = field(default=None)` class-attribute instance-attribute
- `prompt_driver = field(default=Factory(lambda: Defaults.drivers_config.prompt_driver))` class-attribute instance-attribute

default_generate_system_template(context, artifacts)

Source Code in griptape/engines/rag/modules/response/prompt_response_rag_module.py

```
def default_generate_system_template(self, context: RagContext, artifacts:␣
↪list[TextArtifact]) -> str:
    params: dict[str, Any] = {"text_chunks": [c.to_text() for c in artifacts]}

    if len(self.rulesets) > 0:
        params["rulesets"] = J2("rulesets/rulesets.j2").render(rulesets=self.rulesets)

    if self.metadata is not None:
        params["metadata"] = J2("engines/rag/modules/response/metadata/system.j2").
↪render(metadata=self.metadata)

    return J2("engines/rag/modules/response/prompt/system.j2").render(**params)
```

run(context)

Source Code in griptape/engines/rag/modules/response/prompt_response_rag_module.py

```
def run(self, context: RagContext) -> BaseArtifact:
    query = context.query
    tokenizer = self.prompt_driver.tokenizer
    included_chunks = []
    system_prompt = self.generate_system_template(context, included_chunks)

    for artifact in context.text_chunks:
        included_chunks.append(artifact)

        system_prompt = self.generate_system_template(context, included_chunks)
        message_token_count = self.prompt_driver.tokenizer.count_tokens(
            self.prompt_driver.prompt_stack_to_string(self.generate_prompt_
↪stack(system_prompt, query)),
        )

        if message_token_count + self.answer_token_offset >= tokenizer.max_input_
↪tokens:
            included_chunks.pop()
```

(continues on next page)

(continued from previous page)

```
            system_prompt = self.generate_system_template(context, included_chunks)

            break

    output = self.prompt_driver.run(self.generate_prompt_stack(system_prompt, query)).
↪to_artifact()

    if isinstance(output, TextArtifact):
        return output
    raise ValueError("Prompt driver did not return a TextArtifact")
```

TextChunksRerankRagModule

Bases: *BaseRerankRagModule* “)

Source Code in griptape/engines/rag/modules/retrieval/text_chunks_rerank_rag_module.py

```
@define(kw_only=True)
class TextChunksRerankRagModule(BaseRerankRagModule):
    def run(self, context: RagContext) -> Sequence[BaseArtifact]:
        return self.rerank_driver.run(context.query, context.text_chunks)
```

run(context)

Source Code in griptape/engines/rag/modules/retrieval/text_chunks_rerank_rag_module.py

```
def run(self, context: RagContext) -> Sequence[BaseArtifact]:
    return self.rerank_driver.run(context.query, context.text_chunks)
```

TextChunksResponseRagModule

Bases: *BaseResponseRagModule* “)

Source Code in griptape/engines/rag/modules/response/text_chunks_response_rag_module.py

```
@define(kw_only=True)
class TextChunksResponseRagModule(BaseResponseRagModule):
    def run(self, context: RagContext) -> BaseArtifact:
        return ListArtifact(context.text_chunks)
```

run(context)

Source Code in griptape/engines/rag/modules/response/text_chunks_response_rag_module.py

```
def run(self, context: RagContext) -> BaseArtifact:
    return ListArtifact(context.text_chunks)
```

TextLoaderRetrievalRagModule

Bases: *BaseRetrievalRagModule* “)

Source Code in griptape/engines/rag/modules/retrieval/text_loader_retrieval_rag_module.py

```
@define(kw_only=True)
class TextLoaderRetrievalRagModule(BaseRetrievalRagModule):
    loader: TextLoader = field()
    chunker: TextChunker = field(default=Factory(lambda: TextChunker()))
    vector_store_driver: BaseVectorStoreDriver = field()
    source: Any = field()
    query_params: dict[str, Any] = field(factory=dict)
    process_query_output: Callable[[list[BaseVectorStoreDriver.Entry]],␣
↪Sequence[TextArtifact]] = field(
        default=Factory(lambda: lambda es: [e.to_artifact() for e in es]),
    )

    def run(self, context: RagContext) -> Sequence[TextArtifact]:
        namespace = uuid.uuid4().hex
        context_source = self.get_context_param(context, "source")
        source = self.source if context_source is None else context_source

        query_params = utils.dict_merge(self.query_params, self.get_context_
↪param(context, "query_params"))

        query_params["namespace"] = namespace

        loader_output = self.loader.load(source)
        chunks = self.chunker.chunk(loader_output)

        self.vector_store_driver.upsert_collection({namespace: chunks})

        return self.process_query_output(self.vector_store_driver.query(context.query,
↪ **query_params))
```

- `chunker = field(default=Factory(lambda: TextChunker()))` class-attribute instance-attribute
- `loader = field()` class-attribute instance-attribute
- `process_query_output = field(default=Factory(lambda: lambda es: [e.to_artifact() for e in es]))` class-attribute instance-attribute
- `query_params = field(factory=dict)` class-attribute instance-attribute
- `source = field()` class-attribute instance-attribute
- `vector_store_driver = field()` class-attribute instance-attribute

run(context)

Source Code in griptape/engines/rag/modules/retrieval/text_loader_retrieval_rag_module.py

```
def run(self, context: RagContext) -> Sequence[TextArtifact]:
    namespace = uuid.uuid4().hex
    context_source = self.get_context_param(context, "source")
    source = self.source if context_source is None else context_source

    query_params = utils.dict_merge(self.query_params, self.get_context_param(context,
↪ "query_params"))
```

(continues on next page)

(continued from previous page)

```
    query_params["namespace"] = namespace

    loader_output = self.loader.load(source)
    chunks = self.chunker.chunk(loader_output)

    self.vector_store_driver.upsert_collection({namespace: chunks})

    return self.process_query_output(self.vector_store_driver.query(context.query,␣
↪**query_params))
```

TranslateQueryRagModule

Bases: *BaseQueryRagModule* “)

Source Code in griptape/engines/rag/modules/query/translate_query_rag_module.py

```
@define(kw_only=True)
class TranslateQueryRagModule(BaseQueryRagModule):
    prompt_driver: BasePromptDriver = field()
    language: str = field()
    generate_user_template: Callable[[str, str], str] = field(
        default=Factory(lambda self: self.default_generate_user_template, takes_
↪self=True),
    )

    def run(self, context: RagContext) -> RagContext:
        user_prompt = self.generate_user_template(context.query, self.language)
        output = self.prompt_driver.run(self.generate_prompt_stack(None, user_
↪prompt)).to_artifact()

        context.query = output.to_text()

        return context

    def default_generate_user_template(self, query: str, language: str) -> str:
        return J2("engines/rag/modules/query/translate/user.j2").render(query=query,␣
↪language=language)
```

- `generate_user_template = field(default=Factory(lambda self: self.default_generate_user_template, takes_self=True))` class-attribute instance-attribute
- `language = field()` class-attribute instance-attribute
- `prompt_driver = field()` class-attribute instance-attribute

default_generate_user_template(query, language)

Source Code in griptape/engines/rag/modules/query/translate_query_rag_module.py

```
def default_generate_user_template(self, query: str, language: str) -> str:
    return J2("engines/rag/modules/query/translate/user.j2").render(query=query,␣
↪language=language)
```

run(context)

Source Code in griptape/engines/rag/modules/query/translate_query_rag_module.py

```
def run(self, context: RagContext) -> RagContext:
    user_prompt = self.generate_user_template(context.query, self.language)
    output = self.prompt_driver.run(self.generate_prompt_stack(None, user_prompt)).to_
↪artifact()

    context.query = output.to_text()

    return context
```

VectorStoreRetrievalRagModule

Bases: *BaseRetrievalRagModule* “)

Source Code in griptape/engines/rag/modules/retrieval/vector_store_retrieval_rag_module.py

```
@define(kw_only=True)
class VectorStoreRetrievalRagModule(BaseRetrievalRagModule):
    vector_store_driver: BaseVectorStoreDriver = field(
        default=Factory(lambda: Defaults.drivers_config.vector_store_driver)
    )
    query_params: dict[str, Any] = field(factory=dict)
    process_query_output: Callable[[list[BaseVectorStoreDriver.Entry]],␣
↪Sequence[TextArtifact]] = field(
        default=Factory(lambda: lambda es: [e.to_artifact() for e in es]),
    )

    def run(self, context: RagContext) -> Sequence[TextArtifact]:
        query_params = utils.dict_merge(self.query_params, self.get_context_
↪param(context, "query_params"))

        return self.process_query_output(self.vector_store_driver.query(context.query,
↪ **query_params))
```

- `process_query_output = field(default=Factory(lambda: lambda es: [e.to_artifact() for e in es]))` class-attribute instance-attribute
- `query_params = field(factory=dict)` class-attribute instance-attribute
- `vector_store_driver = field(default=Factory(lambda: Defaults.drivers_config.vector_store_driver))` class-attribute instance-attribute

run(context)

Source Code in griptape/engines/rag/modules/retrieval/vector_store_retrieval_rag_module.py

```
def run(self, context: RagContext) -> Sequence[TextArtifact]:
    query_params = utils.dict_merge(self.query_params, self.get_context_param(context,
↪ "query_params"))

    return self.process_query_output(self.vector_store_driver.query(context.query,␣
↪**query_params))
```

base_rag_module

Bases: *FuturesExecutorMixin* “) , ABC

Source Code in griptape/engines/rag/modules/base_rag_module.py

```
@define(kw_only=True)
class BaseRagModule(FuturesExecutorMixin, ABC):
    name: str = field(
        default=Factory(lambda self: f"{self.__class__.__name__}-{uuid.uuid4().hex}",␣
↪takes_self=True), kw_only=True
    )

    def generate_prompt_stack(self, system_prompt: Optional[str], query: str) ->␣
↪PromptStack:
        messages = []

        if system_prompt is not None:
            messages.append(Message(system_prompt, role=Message.SYSTEM_ROLE))

        messages.append(Message(query, role=Message.USER_ROLE))

        return PromptStack(messages=messages)

    def get_context_param(self, context: RagContext, key: str) -> Optional[Any]:
        return context.module_configs.get(self.name, {}).get(key)

    def set_context_param(self, context: RagContext, key: str, value: Any) -> None:
        if not isinstance(context.module_configs.get(self.name), dict):
            context.module_configs[self.name] = {}

        context.module_configs[self.name][key] = value
```

- `name = field(default=Factory(lambda self: f'{self.__class__.__name__}-{uuid.uuid4().hex}', takes_self=True), kw_only=True)` class-attribute instance-attribute

generate_prompt_stack(system_prompt, query)

Source Code in griptape/engines/rag/modules/base_rag_module.py

```
def generate_prompt_stack(self, system_prompt: Optional[str], query: str) ->␣
↪PromptStack:
    messages = []

    if system_prompt is not None:
        messages.append(Message(system_prompt, role=Message.SYSTEM_ROLE))

    messages.append(Message(query, role=Message.USER_ROLE))

    return PromptStack(messages=messages)
```

get_context_param(context, key)

Source Code in griptape/engines/rag/modules/base_rag_module.py

```
def get_context_param(self, context: RagContext, key: str) -> Optional[Any]:
    return context.module_configs.get(self.name, {}).get(key)
```

set_context_param(context, key, value)

Source Code in griptape/engines/rag/modules/base_rag_module.py

```
def set_context_param(self, context: RagContext, key: str, value: Any) -> None:
    if not isinstance(context.module_configs.get(self.name), dict):
        context.module_configs[self.name] = {}

    context.module_configs[self.name][key] = value
```

query

base_query_rag_module

Bases: *BaseRagModule* “) , ABC

Source Code in griptape/engines/rag/modules/query/base_query_rag_module.py

```
@define(kw_only=True)
class BaseQueryRagModule(BaseRagModule, ABC):
    @abstractmethod
    def run(self, context: RagContext) -> RagContext: ...
```

run(context)abstractmethod

Source Code in griptape/engines/rag/modules/query/base_query_rag_module.py

```
@abstractmethod
def run(self, context: RagContext) -> RagContext: ...
```

translate_query_rag_module

Bases: *BaseQueryRagModule* “)

Source Code in griptape/engines/rag/modules/query/translate_query_rag_module.py

```
@define(kw_only=True)
class TranslateQueryRagModule(BaseQueryRagModule):
    prompt_driver: BasePromptDriver = field()
    language: str = field()
    generate_user_template: Callable[[str, str], str] = field(
        default=Factory(lambda self: self.default_generate_user_template, takes_
↪self=True),
    )

    def run(self, context: RagContext) -> RagContext:
```

(continues on next page)

(continued from previous page)

```
        user_prompt = self.generate_user_template(context.query, self.language)
        output = self.prompt_driver.run(self.generate_prompt_stack(None, user_
↪prompt)).to_artifact()

        context.query = output.to_text()

        return context

    def default_generate_user_template(self, query: str, language: str) -> str:
        return J2("engines/rag/modules/query/translate/user.j2").render(query=query,␣
↪language=language)
```

- `generate_user_template = field(default=Factory(lambda self: self.default_generate_user_template, takes_self=True))` class-attribute instance-attribute
- `language = field()` class-attribute instance-attribute
- `prompt_driver = field()` class-attribute instance-attribute

default_generate_user_template(query, language)

Source Code in griptape/engines/rag/modules/query/translate_query_rag_module.py

```
def default_generate_user_template(self, query: str, language: str) -> str:
    return J2("engines/rag/modules/query/translate/user.j2").render(query=query,␣
↪language=language)
```

run(context)

Source Code in griptape/engines/rag/modules/query/translate_query_rag_module.py

```
def run(self, context: RagContext) -> RagContext:
    user_prompt = self.generate_user_template(context.query, self.language)
    output = self.prompt_driver.run(self.generate_prompt_stack(None, user_prompt)).to_
↪artifact()

    context.query = output.to_text()

    return context
```

response

base_after_response_rag_module

Bases: *BaseRagModule* “) , ABC

Source Code in griptape/engines/rag/modules/response/base_after_response_rag_module.py

```
@define(kw_only=True)
class BaseAfterResponseRagModule(BaseRagModule, ABC):
    @abstractmethod
    def run(self, context: RagContext) -> RagContext: ...
```

run(context)abstractmethod

Source Code in griptape/engines/rag/modules/response/base_after_response_rag_module.py

```
@abstractmethod
def run(self, context: RagContext) -> RagContext: ...
```

base_before_response_rag_module

Bases: *BaseRagModule* “) , ABC

Source Code in griptape/engines/rag/modules/response/base_before_response_rag_module.py

```
@define(kw_only=True)
class BaseBeforeResponseRagModule(BaseRagModule, ABC):
    @abstractmethod
    def run(self, context: RagContext) -> RagContext: ...
```

run(context)abstractmethod

Source Code in griptape/engines/rag/modules/response/base_before_response_rag_module.py

```
@abstractmethod
def run(self, context: RagContext) -> RagContext: ...
```

base_response_rag_module

Bases: *BaseRagModule* “) , ABC

Source Code in griptape/engines/rag/modules/response/base_response_rag_module.py

```
@define(kw_only=True)
class BaseResponseRagModule(BaseRagModule, ABC):
    @abstractmethod
    def run(self, context: RagContext) -> BaseArtifact: ...
```

run(context)abstractmethod

Source Code in griptape/engines/rag/modules/response/base_response_rag_module.py

```
@abstractmethod
def run(self, context: RagContext) -> BaseArtifact: ...
```

footnote_prompt_response_rag_module

Bases: *FootnotePromptResponseRagModule* “)

Source Code in griptape/engines/rag/modules/response/footnote_prompt_response_rag_module.py

```
@define(kw_only=True)
class FootnotePromptResponseRagModule(PromptResponseRagModule):
    def default_generate_system_template(self, context: RagContext, artifacts:␣
↪list[TextArtifact]) -> str:
```

(continues on next page)

(continued from previous page)

```
        return J2("engines/rag/modules/response/footnote_prompt/system.j2").render(
            text_chunk_artifacts=artifacts,
            references=utils.references_from_artifacts(artifacts),
            before_system_prompt="\n\n".join(context.before_query),
            after_system_prompt="\n\n".join(context.after_query),
        )
```

default_generate_system_template(context, artifacts)

Source Code in griptape/engines/rag/modules/response/footnote_prompt_response_rag_module.py

```
def default_generate_system_template(self, context: RagContext, artifacts:␣
↪list[TextArtifact]) -> str:
    return J2("engines/rag/modules/response/footnote_prompt/system.j2").render(
        text_chunk_artifacts=artifacts,
        references=utils.references_from_artifacts(artifacts),
        before_system_prompt="\n\n".join(context.before_query),
        after_system_prompt="\n\n".join(context.after_query),
    )
```

prompt_response_rag_module

Bases: *BaseResponseRagModule* “) , *RuleMixin* ”)

Source Code in griptape/engines/rag/modules/response/prompt_response_rag_module.py

```
@define(kw_only=True)
class PromptResponseRagModule(BaseResponseRagModule, RuleMixin):
    prompt_driver: BasePromptDriver = field(default=Factory(lambda: Defaults.drivers_
↪config.prompt_driver))
    answer_token_offset: int = field(default=400)
    metadata: Optional[str] = field(default=None)
    generate_system_template: Callable[[RagContext, list[TextArtifact]], str] = field(
        default=Factory(lambda self: self.default_generate_system_template, takes_
↪self=True),
    )

    def run(self, context: RagContext) -> BaseArtifact:
        query = context.query
        tokenizer = self.prompt_driver.tokenizer
        included_chunks = []
        system_prompt = self.generate_system_template(context, included_chunks)

        for artifact in context.text_chunks:
            included_chunks.append(artifact)

            system_prompt = self.generate_system_template(context, included_chunks)
            message_token_count = self.prompt_driver.tokenizer.count_tokens(
                self.prompt_driver.prompt_stack_to_string(self.generate_prompt_
↪stack(system_prompt, query)),
            )
```

(continues on next page)

(continued from previous page)

```
            if message_token_count + self.answer_token_offset >= tokenizer.max_input_
↪tokens:
                included_chunks.pop()

                system_prompt = self.generate_system_template(context, included_
↪chunks)

                break

        output = self.prompt_driver.run(self.generate_prompt_stack(system_prompt,␣
↪query)).to_artifact()

        if isinstance(output, TextArtifact):
            return output
        raise ValueError("Prompt driver did not return a TextArtifact")

    def default_generate_system_template(self, context: RagContext, artifacts:␣
↪list[TextArtifact]) -> str:
        params: dict[str, Any] = {"text_chunks": [c.to_text() for c in artifacts]}

        if len(self.rulesets) > 0:
            params["rulesets"] = J2("rulesets/rulesets.j2").render(rulesets=self.
↪rulesets)

        if self.metadata is not None:
            params["metadata"] = J2("engines/rag/modules/response/metadata/system.j2
↪").render(metadata=self.metadata)

        return J2("engines/rag/modules/response/prompt/system.j2").render(**params)
```

- `answer_token_offset = field(default=400)` class-attribute instance-attribute
- `generate_system_template = field(default=Factory(lambda self: self.default_generate_system_template, takes_self=True))` class-attribute instance-attribute
- `metadata = field(default=None)` class-attribute instance-attribute
- `prompt_driver = field(default=Factory(lambda: Defaults.drivers_config.prompt_driver))` class-attribute instance-attribute

default_generate_system_template(context, artifacts)

Source Code in griptape/engines/rag/modules/response/prompt_response_rag_module.py

```
def default_generate_system_template(self, context: RagContext, artifacts:␣
↪list[TextArtifact]) -> str:
    params: dict[str, Any] = {"text_chunks": [c.to_text() for c in artifacts]}

    if len(self.rulesets) > 0:
        params["rulesets"] = J2("rulesets/rulesets.j2").render(rulesets=self.rulesets)

    if self.metadata is not None:
        params["metadata"] = J2("engines/rag/modules/response/metadata/system.j2").
↪render(metadata=self.metadata)
```

(continues on next page)

(continued from previous page)

```
    return J2("engines/rag/modules/response/prompt/system.j2").render(**params)
```

run(context)

Source Code in griptape/engines/rag/modules/response/prompt_response_rag_module.py

```
def run(self, context: RagContext) -> BaseArtifact:
    query = context.query
    tokenizer = self.prompt_driver.tokenizer
    included_chunks = []
    system_prompt = self.generate_system_template(context, included_chunks)

    for artifact in context.text_chunks:
        included_chunks.append(artifact)

        system_prompt = self.generate_system_template(context, included_chunks)
        message_token_count = self.prompt_driver.tokenizer.count_tokens(
            self.prompt_driver.prompt_stack_to_string(self.generate_prompt_
↪stack(system_prompt, query)),
        )

        if message_token_count + self.answer_token_offset >= tokenizer.max_input_
↪tokens:
            included_chunks.pop()

            system_prompt = self.generate_system_template(context, included_chunks)

            break

    output = self.prompt_driver.run(self.generate_prompt_stack(system_prompt, query)).
↪to_artifact()

    if isinstance(output, TextArtifact):
        return output
    raise ValueError("Prompt driver did not return a TextArtifact")
```

text_chunks_response_rag_module

Bases: *BaseResponseRagModule* “)

Source Code in griptape/engines/rag/modules/response/text_chunks_response_rag_module.py

```
@define(kw_only=True)
class TextChunksResponseRagModule(BaseResponseRagModule):
    def run(self, context: RagContext) -> BaseArtifact:
        return ListArtifact(context.text_chunks)
```

run(context)

Source Code in griptape/engines/rag/modules/response/text_chunks_response_rag_module.py

```
def run(self, context: RagContext) -> BaseArtifact:
    return ListArtifact(context.text_chunks)
```

retrieval

base_rerank_rag_module

Bases: *BaseRagModule* “) , ABC

Source Code in griptape/engines/rag/modules/retrieval/base_rerank_rag_module.py

```
@define(kw_only=True)
class BaseRerankRagModule(BaseRagModule, ABC):
    rerank_driver: BaseRerankDriver = field()

    @abstractmethod
    def run(self, context: RagContext) -> Sequence[BaseArtifact]: ...
```

- `rerank_driver = field()` class-attribute instance-attribute

run(context)abstractmethod

Source Code in griptape/engines/rag/modules/retrieval/base_rerank_rag_module.py

```
@abstractmethod
def run(self, context: RagContext) -> Sequence[BaseArtifact]: ...
```

base_retrieval_rag_module

Bases: *BaseRagModule* “) , ABC

Source Code in griptape/engines/rag/modules/retrieval/base_retrieval_rag_module.py

```
@define(kw_only=True)
class BaseRetrievalRagModule(BaseRagModule, ABC):
    @abstractmethod
    def run(self, context: RagContext) -> Sequence[BaseArtifact]: ...
```

run(context)abstractmethod

Source Code in griptape/engines/rag/modules/retrieval/base_retrieval_rag_module.py

```
@abstractmethod
def run(self, context: RagContext) -> Sequence[BaseArtifact]: ...
```

text_chunks_rerank_rag_module

Bases: *BaseRerankRagModule* “)

Source Code in griptape/engines/rag/modules/retrieval/text_chunks_rerank_rag_module.py

```
@define(kw_only=True)
class TextChunksRerankRagModule(BaseRerankRagModule):
    def run(self, context: RagContext) -> Sequence[BaseArtifact]:
        return self.rerank_driver.run(context.query, context.text_chunks)
```

run(context)

Source Code in griptape/engines/rag/modules/retrieval/text_chunks_rerank_rag_module.py

```
def run(self, context: RagContext) -> Sequence[BaseArtifact]:
    return self.rerank_driver.run(context.query, context.text_chunks)
```

text_loader_retrieval_rag_module

Bases: *BaseRetrievalRagModule* “)

Source Code in griptape/engines/rag/modules/retrieval/text_loader_retrieval_rag_module.py

```
@define(kw_only=True)
class TextLoaderRetrievalRagModule(BaseRetrievalRagModule):
    loader: TextLoader = field()
    chunker: TextChunker = field(default=Factory(lambda: TextChunker()))
    vector_store_driver: BaseVectorStoreDriver = field()
    source: Any = field()
    query_params: dict[str, Any] = field(factory=dict)
    process_query_output: Callable[[list[BaseVectorStoreDriver.Entry]],␣
↪Sequence[TextArtifact]] = field(
        default=Factory(lambda: lambda es: [e.to_artifact() for e in es]),
    )

    def run(self, context: RagContext) -> Sequence[TextArtifact]:
        namespace = uuid.uuid4().hex
        context_source = self.get_context_param(context, "source")
        source = self.source if context_source is None else context_source

        query_params = utils.dict_merge(self.query_params, self.get_context_
↪param(context, "query_params"))

        query_params["namespace"] = namespace

        loader_output = self.loader.load(source)
        chunks = self.chunker.chunk(loader_output)

        self.vector_store_driver.upsert_collection({namespace: chunks})

        return self.process_query_output(self.vector_store_driver.query(context.query,
↪ **query_params))
```

- `chunker = field(default=Factory(lambda: TextChunker()))` class-attribute instance-attribute
- `loader = field()` class-attribute instance-attribute
- `process_query_output = field(default=Factory(lambda: lambda es: [e.to_artifact() for e in es]))` class-attribute instance-attribute

- `query_params = field(factory=dict)` class-attribute instance-attribute
- `source = field()` class-attribute instance-attribute
- `vector_store_driver = field()` class-attribute instance-attribute

run(context)

Source Code in griptape/engines/rag/modules/retrieval/text_loader_retrieval_rag_module.py

```
def run(self, context: RagContext) -> Sequence[TextArtifact]:
    namespace = uuid.uuid4().hex
    context_source = self.get_context_param(context, "source")
    source = self.source if context_source is None else context_source

    query_params = utils.dict_merge(self.query_params, self.get_context_param(context,
↪ "query_params"))

    query_params["namespace"] = namespace

    loader_output = self.loader.load(source)
    chunks = self.chunker.chunk(loader_output)

    self.vector_store_driver.upsert_collection({namespace: chunks})

    return self.process_query_output(self.vector_store_driver.query(context.query,␣
↪**query_params))
```

vector_store_retrieval_rag_module

Bases: *BaseRetrievalRagModule* “)

Source Code in griptape/engines/rag/modules/retrieval/vector_store_retrieval_rag_module.py

```
@define(kw_only=True)
class VectorStoreRetrievalRagModule(BaseRetrievalRagModule):
    vector_store_driver: BaseVectorStoreDriver = field(
        default=Factory(lambda: Defaults.drivers_config.vector_store_driver)
    )
    query_params: dict[str, Any] = field(factory=dict)
    process_query_output: Callable[[list[BaseVectorStoreDriver.Entry]],␣
↪Sequence[TextArtifact]] = field(
        default=Factory(lambda: lambda es: [e.to_artifact() for e in es]),
    )

    def run(self, context: RagContext) -> Sequence[TextArtifact]:
        query_params = utils.dict_merge(self.query_params, self.get_context_
↪param(context, "query_params"))

        return self.process_query_output(self.vector_store_driver.query(context.query,
↪ **query_params))
```

- `process_query_output = field(default=Factory(lambda: lambda es: [e.to_artifact() for e in es]))` class-attribute instance-attribute
- `query_params = field(factory=dict)` class-attribute instance-attribute

- `vector_store_driver = field(default=Factory(lambda: Defaults.drivers_config.vector_store_driver))` class-attribute instance-attribute

run(context)

Source Code in griptape/engines/rag/modules/retrieval/vector_store_retrieval_rag_module.py

```
def run(self, context: RagContext) -> Sequence[TextArtifact]:
    query_params = utils.dict_merge(self.query_params, self.get_context_param(context,
↪ "query_params"))

    return self.process_query_output(self.vector_store_driver.query(context.query,␣
↪**query_params))
```

rag_context

Bases: *SerializableMixin* “)

Attributes

Name	Type	Description
<code class="language-python">	*str*	Query provided by the user.
<code class="language-python">	*dict[str, dict]*	Dictionary of module configs. First key should be a module name and the second a dictionary of configs parameters.

continues on next page

Table 317 – continued from previous page

Name	Type	Description
<code class="language-python">	<code class="language-python">list[str]</code>	An optional list of strings to add before the query in response modules.
<code class="language-python">	<code class="language-python">list[str]</code>	An optional list of strings to add after the query in response modules.
<a href="rag_context.mdx#griptape.engines.rag.rag_context.RagContext.text_chunks" title="text_chunks = field(factory=list, metadata={'serializable': True}) class-attribute instance-attribute (griptape.engines.rag.rag_context.RagContext.text_chunks)"><code class="language-python">	<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">list[TextArtifact]</code></a	A list of text chunks to pass around from the retrieval stage to the response stage.

continues on next page

Table 317 – continued from previous page

Name	Type	Description
<a href="rag_context.mdx#griptape.engines.rag.rag_context.RagContext.outputs" title="outputs = field(factory=list, metadata={'serializable': True}) class-attribute instance-attribute (griptape.engines.rag.rag_context.RagContext.outputs)"><code class="language-python">	<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">list[BaseArtifact]</code></a	List of outputs from the response stage.

Source Code in griptape/engines/rag/rag_context.py

```
@define(kw_only=True)
class RagContext(SerializableMixin):
    """Used by RagEngine stages and module to pass context that individual modules are expected to update in the `run` method.

    Attributes:
        query: Query provided by the user.
        module_configs: Dictionary of module configs. First key should be a module name and the second a dictionary of configs parameters.
        before_query: An optional list of strings to add before the query in response modules.
        after_query: An optional list of strings to add after the query in response modules.
        text_chunks: A list of text chunks to pass around from the retrieval stage to the response stage.
        outputs: List of outputs from the response stage.
    """

    query: str = field(metadata={"serializable": True})
    module_configs: dict[str, dict] = field(factory=dict, metadata={"serializable": True})
    before_query: list[str] = field(factory=list, metadata={"serializable": True})
    after_query: list[str] = field(factory=list, metadata={"serializable": True})
    text_chunks: list[TextArtifact] = field(factory=list, metadata={"serializable": True})
    outputs: list[BaseArtifact] = field(factory=list, metadata={"serializable": True})

    def get_references(self) -> list[Reference]:
        return utils.references_from_artifacts(self.text_chunks)
```

- `after_query = field(factory=list, metadata={'serializable': True})` class-attribute instance-attribute
- `before_query = field(factory=list, metadata={'serializable': True})` class-attribute instance-attribute

- `module_configs = field(factory=dict, metadata={'serializable': True})` class-attribute instance-attribute
- `outputs = field(factory=list, metadata={'serializable': True})` class-attribute instance-attribute
- `query = field(metadata={'serializable': True})` class-attribute instance-attribute
- `text_chunks = field(factory=list, metadata={'serializable': True})` class-attribute instance-attribute

get_references()

Source Code in griptape/engines/rag/rag_context.py

```
def get_references(self) -> list[Reference]:
    return utils.references_from_artifacts(self.text_chunks)
```

rag_engine

Source Code in griptape/engines/rag/rag_engine.py

```
@define(kw_only=True)
class RagEngine:
    query_stage: Optional[QueryRagStage] = field(default=None)
    retrieval_stage: Optional[RetrievalRagStage] = field(default=None)
    response_stage: Optional[ResponseRagStage] = field(default=None)

    def __attrs_post_init__(self) -> None:
        modules = []

        if self.query_stage is not None:
            modules.extend(self.query_stage.modules)

        if self.retrieval_stage is not None:
            modules.extend(self.retrieval_stage.modules)

        if self.response_stage is not None:
            modules.extend(self.response_stage.modules)

        module_names = [m.name for m in modules]

        if len(module_names) > len(set(module_names)):
            raise ValueError("module names have to be unique")

    def process_query(self, query: str) -> RagContext:
        return self.process(RagContext(query=query))

    def process(self, context: RagContext) -> RagContext:
        if self.query_stage:
            context = self.query_stage.run(context)

        if self.retrieval_stage:
            context = self.retrieval_stage.run(context)
```

(continues on next page)

(continued from previous page)

```
        if self.response_stage:
            context = self.response_stage.run(context)

        return context
```

- `query_stage = field(default=None)` class-attribute instance-attribute
- `response_stage = field(default=None)` class-attribute instance-attribute
- `retrieval_stage = field(default=None)` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/engines/rag/rag_engine.py

```
def __attrs_post_init__(self) -> None:
    modules = []

    if self.query_stage is not None:
        modules.extend(self.query_stage.modules)

    if self.retrieval_stage is not None:
        modules.extend(self.retrieval_stage.modules)

    if self.response_stage is not None:
        modules.extend(self.response_stage.modules)

    module_names = [m.name for m in modules]

    if len(module_names) > len(set(module_names)):
        raise ValueError("module names have to be unique")
```

process(context)

Source Code in griptape/engines/rag/rag_engine.py

```
def process(self, context: RagContext) -> RagContext:
    if self.query_stage:
        context = self.query_stage.run(context)

    if self.retrieval_stage:
        context = self.retrieval_stage.run(context)

    if self.response_stage:
        context = self.response_stage.run(context)

    return context
```

process_query(query)

Source Code in griptape/engines/rag/rag_engine.py

```
def process_query(self, query: str) -> RagContext:
    return self.process(RagContext(query=query))
```

stages

- `__all__ = ['BaseRagStage', 'QueryRagStage', 'ResponseRagStage', 'RetrievalRagStage']` module-attribute

Bases: *FuturesExecutorMixin* “) , ABC

Source Code in griptape/engines/rag/stages/base_rag_stage.py

```
@define(kw_only=True)
class BaseRagStage(FuturesExecutorMixin, ABC):
    @abstractmethod
    def run(self, context: RagContext) -> RagContext: ...

    @property
    @abstractmethod
    def modules(self) -> Sequence[BaseRagModule]: ...
```

- `modules` abstractmethod property

run(context)abstractmethod

Source Code in griptape/engines/rag/stages/base_rag_stage.py

```
@abstractmethod
def run(self, context: RagContext) -> RagContext: ...
```

QueryRagStage

Bases: BaseRagStage “)

Source Code in griptape/engines/rag/stages/query_rag_stage.py

```
@define(kw_only=True)
class QueryRagStage(BaseRagStage):
    query_modules: list[BaseQueryRagModule] = field()

    @property
    def modules(self) -> Sequence[BaseRagModule]:
        return self.query_modules

    def run(self, context: RagContext) -> RagContext:
        logging.info("QueryRagStage: running %s query generation modules sequentially
↪", len(self.query_modules))

        [qm.run(context) for qm in self.query_modules]

        return context
```

- `modules` property
- `query_modules = field()` class-attribute instance-attribute

run(context)

Source Code in griptape/engines/rag/stages/query_rag_stage.py

```
def run(self, context: RagContext) -> RagContext:
    logging.info("QueryRagStage: running %s query generation modules sequentially",␣
↪len(self.query_modules))

    [qm.run(context) for qm in self.query_modules]

    return context
```

ResponseRagStage

Bases: BaseRagStage “)

Source Code in griptape/engines/rag/stages/response_rag_stage.py

```
@define(kw_only=True)
class ResponseRagStage(BaseRagStage):
    response_modules: list[BaseResponseRagModule] = field()

    @property
    def modules(self) -> list[BaseRagModule]:
        ms = []

        ms.extend(self.response_modules)

        return ms

    def run(self, context: RagContext) -> RagContext:
        logging.info("ResponseRagStage: running %s retrieval modules in parallel",␣
↪len(self.response_modules))

        with self.create_futures_executor() as futures_executor:
            results = utils.execute_futures_list(
                [futures_executor.submit(with_contextvars(r.run), context) for r in␣
↪self.response_modules]
            )

        context.outputs = results

        return context
```

- `modules` property
- `response_modules = field()` class-attribute instance-attribute

run(context)

Source Code in griptape/engines/rag/stages/response_rag_stage.py

```
def run(self, context: RagContext) -> RagContext:
    logging.info("ResponseRagStage: running %s retrieval modules in parallel",␣
```

(continues on next page)

(continued from previous page)

```
↪len(self.response_modules))

    with self.create_futures_executor() as futures_executor:
        results = utils.execute_futures_list(
            [futures_executor.submit(with_contextvars(r.run), context) for r in self.
↪response_modules]
        )

    context.outputs = results

    return context
```

RetrievalRagStage

Bases: BaseRagStage “)

Source Code in griptape/engines/rag/stages/retrieval_rag_stage.py

```
@define(kw_only=True)
class RetrievalRagStage(BaseRagStage):
    retrieval_modules: list[BaseRetrievalRagModule] = field()
    rerank_module: Optional[BaseRerankRagModule] = field(default=None)
    max_chunks: Optional[int] = field(default=None)

    @property
    def modules(self) -> list[BaseRagModule]:
        ms = []

        ms.extend(self.retrieval_modules)

        if self.rerank_module is not None:
            ms.append(self.rerank_module)

        return ms

    def run(self, context: RagContext) -> RagContext:
        logging.info("RetrievalRagStage: running %s retrieval modules in parallel",␣
↪len(self.retrieval_modules))

        with self.create_futures_executor() as futures_executor:
            results = utils.execute_futures_list(
                [futures_executor.submit(with_contextvars(r.run), context) for r in␣
↪self.retrieval_modules]
            )

        # flatten the list of lists
        results = list(itertools.chain.from_iterable(results))

        # deduplicate the list
        chunks_before_dedup = len(results)
        results = list({str(c.value): c for c in results}.values())
        chunks_after_dedup = len(results)
```

(continues on next page)

(continued from previous page)

```
        logging.info(
            "RetrievalRagStage: deduplicated %s chunks (%s - %s)",
            chunks_before_dedup - chunks_after_dedup,
            chunks_before_dedup,
            chunks_after_dedup,
        )

        context.text_chunks = [a for a in results if isinstance(a, TextArtifact)]

        if self.rerank_module:
            logging.info("RetrievalRagStage: running rerank module on %s chunks",␣
↪chunks_after_dedup)

            context.text_chunks = [a for a in self.rerank_module.run(context) if␣
↪isinstance(a, TextArtifact)]

        if self.max_chunks:
            context.text_chunks = context.text_chunks[: self.max_chunks]

        return context
```

- `max_chunks = field(default=None)` class-attribute instance-attribute
- `modules` property
- `rerank_module = field(default=None)` class-attribute instance-attribute
- `retrieval_modules = field()` class-attribute instance-attribute

run(context)

Source Code in griptape/engines/rag/stages/retrieval_rag_stage.py

```
def run(self, context: RagContext) -> RagContext:
    logging.info("RetrievalRagStage: running %s retrieval modules in parallel",␣
↪len(self.retrieval_modules))

    with self.create_futures_executor() as futures_executor:
        results = utils.execute_futures_list(
            [futures_executor.submit(with_contextvars(r.run), context) for r in self.
↪retrieval_modules]
        )

    # flatten the list of lists
    results = list(itertools.chain.from_iterable(results))

    # deduplicate the list
    chunks_before_dedup = len(results)
    results = list({str(c.value): c for c in results}.values())
    chunks_after_dedup = len(results)

    logging.info(
        "RetrievalRagStage: deduplicated %s chunks (%s - %s)",
```

(continues on next page)

(continued from previous page)

```
        chunks_before_dedup - chunks_after_dedup,
        chunks_before_dedup,
        chunks_after_dedup,
    )

    context.text_chunks = [a for a in results if isinstance(a, TextArtifact)]

    if self.rerank_module:
        logging.info("RetrievalRagStage: running rerank module on %s chunks", chunks_
↪after_dedup)

        context.text_chunks = [a for a in self.rerank_module.run(context) if␣
↪isinstance(a, TextArtifact)]

    if self.max_chunks:
        context.text_chunks = context.text_chunks[: self.max_chunks]

    return context
```

base_rag_stage

Bases: *FuturesExecutorMixin* “) , ABC

Source Code in griptape/engines/rag/stages/base_rag_stage.py

```
@define(kw_only=True)
class BaseRagStage(FuturesExecutorMixin, ABC):
    @abstractmethod
    def run(self, context: RagContext) -> RagContext: ...

    @property
    @abstractmethod
    def modules(self) -> Sequence[BaseRagModule]: ...
```

- `modules` abstractmethod property

run(context)abstractmethod

Source Code in griptape/engines/rag/stages/base_rag_stage.py

```
@abstractmethod
def run(self, context: RagContext) -> RagContext: ...
```

query_rag_stage

Bases: BaseRagStage “)

Source Code in griptape/engines/rag/stages/query_rag_stage.py

```
@define(kw_only=True)
class QueryRagStage(BaseRagStage):
    query_modules: list[BaseQueryRagModule] = field()
```

(continues on next page)

(continued from previous page)

```
    @property
    def modules(self) -> Sequence[BaseRagModule]:
        return self.query_modules

    def run(self, context: RagContext) -> RagContext:
        logging.info("QueryRagStage: running %s query generation modules sequentially
↪", len(self.query_modules))

        [qm.run(context) for qm in self.query_modules]

        return context
```

- `modules` property
- `query_modules = field()` class-attribute instance-attribute

run(context)

Source Code in griptape/engines/rag/stages/query_rag_stage.py

```
def run(self, context: RagContext) -> RagContext:
    logging.info("QueryRagStage: running %s query generation modules sequentially",␣
↪len(self.query_modules))

    [qm.run(context) for qm in self.query_modules]

    return context
```

response_rag_stage

Bases: BaseRagStage “)

Source Code in griptape/engines/rag/stages/response_rag_stage.py

```
@define(kw_only=True)
class ResponseRagStage(BaseRagStage):
    response_modules: list[BaseResponseRagModule] = field()

    @property
    def modules(self) -> list[BaseRagModule]:
        ms = []

        ms.extend(self.response_modules)

        return ms

    def run(self, context: RagContext) -> RagContext:
        logging.info("ResponseRagStage: running %s retrieval modules in parallel",␣
↪len(self.response_modules))

        with self.create_futures_executor() as futures_executor:
            results = utils.execute_futures_list(
```

(continues on next page)

(continued from previous page)

```
                [futures_executor.submit(with_contextvars(r.run), context) for r in␣
↪self.response_modules]
            )

        context.outputs = results

        return context
```

- `modules` property
- `response_modules = field()` class-attribute instance-attribute

run(context)

Source Code in griptape/engines/rag/stages/response_rag_stage.py

```
def run(self, context: RagContext) -> RagContext:
    logging.info("ResponseRagStage: running %s retrieval modules in parallel",␣
↪len(self.response_modules))

    with self.create_futures_executor() as futures_executor:
        results = utils.execute_futures_list(
            [futures_executor.submit(with_contextvars(r.run), context) for r in self.
↪response_modules]
        )

    context.outputs = results

    return context
```

retrieval_rag_stage

Bases: BaseRagStage “)

Source Code in griptape/engines/rag/stages/retrieval_rag_stage.py

```
@define(kw_only=True)
class RetrievalRagStage(BaseRagStage):
    retrieval_modules: list[BaseRetrievalRagModule] = field()
    rerank_module: Optional[BaseRerankRagModule] = field(default=None)
    max_chunks: Optional[int] = field(default=None)

    @property
    def modules(self) -> list[BaseRagModule]:
        ms = []

        ms.extend(self.retrieval_modules)

        if self.rerank_module is not None:
            ms.append(self.rerank_module)

        return ms
```

(continues on next page)

(continued from previous page)

```
    def run(self, context: RagContext) -> RagContext:
        logging.info("RetrievalRagStage: running %s retrieval modules in parallel",␣
↪len(self.retrieval_modules))

        with self.create_futures_executor() as futures_executor:
            results = utils.execute_futures_list(
                [futures_executor.submit(with_contextvars(r.run), context) for r in␣
↪self.retrieval_modules]
            )

        # flatten the list of lists
        results = list(itertools.chain.from_iterable(results))

        # deduplicate the list
        chunks_before_dedup = len(results)
        results = list({str(c.value): c for c in results}.values())
        chunks_after_dedup = len(results)

        logging.info(
            "RetrievalRagStage: deduplicated %s chunks (%s - %s)",
            chunks_before_dedup - chunks_after_dedup,
            chunks_before_dedup,
            chunks_after_dedup,
        )

        context.text_chunks = [a for a in results if isinstance(a, TextArtifact)]

        if self.rerank_module:
            logging.info("RetrievalRagStage: running rerank module on %s chunks",␣
↪chunks_after_dedup)

            context.text_chunks = [a for a in self.rerank_module.run(context) if␣
↪isinstance(a, TextArtifact)]

        if self.max_chunks:
            context.text_chunks = context.text_chunks[: self.max_chunks]

        return context
```

- `max_chunks = field(default=None)` class-attribute instance-attribute
- `modules` property
- `rerank_module = field(default=None)` class-attribute instance-attribute
- `retrieval_modules = field()` class-attribute instance-attribute

run(context)

Source Code in griptape/engines/rag/stages/retrieval_rag_stage.py

```
def run(self, context: RagContext) -> RagContext:
    logging.info("RetrievalRagStage: running %s retrieval modules in parallel",␣
↪len(self.retrieval_modules))
```

(continues on next page)

(continued from previous page)

```
    with self.create_futures_executor() as futures_executor:
        results = utils.execute_futures_list(
            [futures_executor.submit(with_contextvars(r.run), context) for r in self.
↪retrieval_modules]
        )

    # flatten the list of lists
    results = list(itertools.chain.from_iterable(results))

    # deduplicate the list
    chunks_before_dedup = len(results)
    results = list({str(c.value): c for c in results}.values())
    chunks_after_dedup = len(results)

    logging.info(
        "RetrievalRagStage: deduplicated %s chunks (%s - %s)",
        chunks_before_dedup - chunks_after_dedup,
        chunks_before_dedup,
        chunks_after_dedup,
    )

    context.text_chunks = [a for a in results if isinstance(a, TextArtifact)]

    if self.rerank_module:
        logging.info("RetrievalRagStage: running rerank module on %s chunks", chunks_
↪after_dedup)

        context.text_chunks = [a for a in self.rerank_module.run(context) if␣
↪isinstance(a, TextArtifact)]

    if self.max_chunks:
        context.text_chunks = context.text_chunks[: self.max_chunks]

    return context
```

summary

base_summary_engine

Bases:

ABC

Source Code in griptape/engines/summary/base_summary_engine.py

```
@define
class BaseSummaryEngine(ABC):
    def summarize_text(self, text: str, *, rulesets: Optional[list[Ruleset]] = None) -
↪> str:
        return self.summarize_artifacts(ListArtifact([TextArtifact(text)]),␣
↪rulesets=rulesets).value
```

(continues on next page)

(continued from previous page)

```
    @abstractmethod
    def summarize_artifacts(
        self,
        artifacts: ListArtifact,
        *,
        rulesets: Optional[list[Ruleset]] = None,
    ) -> TextArtifact: ...
```

summarize_artifacts(artifacts, *, rulesets=None)abstractmethod

Source Code in griptape/engines/summary/base_summary_engine.py

```
@abstractmethod
def summarize_artifacts(
    self,
    artifacts: ListArtifact,
    *,
    rulesets: Optional[list[Ruleset]] = None,
) -> TextArtifact: ...
```

summarize_text(text, *, rulesets=None)

Source Code in griptape/engines/summary/base_summary_engine.py

```
def summarize_text(self, text: str, *, rulesets: Optional[list[Ruleset]] = None) ->␣
↪str:
    return self.summarize_artifacts(ListArtifact([TextArtifact(text)]),␣
↪rulesets=rulesets).value
```

prompt_summary_engine

Bases: *BaseSummaryEngine* “)

Source Code in griptape/engines/summary/prompt_summary_engine.py

```
@define
class PromptSummaryEngine(BaseSummaryEngine):
    chunk_joiner: str = field(default="\n\n", kw_only=True)
    max_token_multiplier: float = field(default=0.5, kw_only=True)
    generate_system_template: J2 = field(default=Factory(lambda: J2("engines/summary/
↪system.j2")), kw_only=True)
    generate_user_template: J2 = field(default=Factory(lambda: J2("engines/summary/
↪user.j2")), kw_only=True)
    prompt_driver: BasePromptDriver = field(
        default=Factory(lambda: Defaults.drivers_config.prompt_driver), kw_only=True
    )
    chunker: BaseChunker = field(
        default=Factory(
            lambda self: TextChunker(tokenizer=self.prompt_driver.tokenizer, max_
↪tokens=self.max_chunker_tokens),
            takes_self=True,
```

(continues on next page)

(continued from previous page)

```
        ),
        kw_only=True,
    )

    @max_token_multiplier.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_allowlist(self, _: Attribute, max_token_multiplier: int) -> None:
        if max_token_multiplier > 1:
            raise ValueError("has to be less than or equal to 1")
        if max_token_multiplier <= 0:
            raise ValueError("has to be greater than 0")

    @property
    def max_chunker_tokens(self) -> int:
        return round(self.prompt_driver.tokenizer.max_input_tokens * self.max_token_
↪multiplier)

    @property
    def min_response_tokens(self) -> int:
        return round(
            self.prompt_driver.tokenizer.max_input_tokens
            - self.prompt_driver.tokenizer.max_input_tokens * self.max_token_
↪multiplier,
        )

    def summarize_artifacts(self, artifacts: ListArtifact, *, rulesets:␣
↪Optional[list[Ruleset]] = None) -> TextArtifact:
        return self.summarize_artifacts_rec(cast("list[TextArtifact]", artifacts.
↪value), None, rulesets=rulesets)

    def summarize_artifacts_rec(
        self,
        artifacts: list[TextArtifact],
        summary: Optional[str] = None,
        rulesets: Optional[list[Ruleset]] = None,
    ) -> TextArtifact:
        if not artifacts:
            if summary is None:
                raise ValueError("No artifacts to summarize")
            return TextArtifact(summary)

        artifacts_text = self.chunk_joiner.join([a.to_text() for a in artifacts])

        system_prompt = self.generate_system_template.render(
            summary=summary,
            rulesets=J2("rulesets/rulesets.j2").render(rulesets=rulesets),
        )

        user_prompt = self.generate_user_template.render(text=artifacts_text)

        if (
            self.prompt_driver.tokenizer.count_input_tokens_left(user_prompt + system_
↪prompt)
```

(continues on next page)

(continued from previous page)

```
            >= self.min_response_tokens
        ):
            result = self.prompt_driver.run(
                PromptStack(
                    messages=[
                        Message(system_prompt, role=Message.SYSTEM_ROLE),
                        Message(user_prompt, role=Message.USER_ROLE),
                    ],
                ),
            ).to_artifact()

            if isinstance(result, TextArtifact):
                return result
            raise ValueError("Prompt driver did not return a TextArtifact")
        chunks = self.chunker.chunk(artifacts_text)

        partial_text = self.generate_user_template.render(text=chunks[0].value)

        return self.summarize_artifacts_rec(
            chunks[1:],
            self.prompt_driver.run(
                PromptStack(
                    messages=[
                        Message(system_prompt, role=Message.SYSTEM_ROLE),
                        Message(partial_text, role=Message.USER_ROLE),
                    ],
                ),
            ).value,
            rulesets=rulesets,
        )
```

- `chunk_joiner = field(default='\n\n', kw_only=True)` class-attribute instance-attribute
- `chunker = field(default=Factory(lambda self: TextChunker(tokenizer=self.prompt_driver.tokenizer, max_tokens=self.max_chunker_tokens), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `generate_system_template = field(default=Factory(lambda: J2('engines/summary/system.j2')), kw_only=True)` class-attribute instance-attribute
- `generate_user_template = field(default=Factory(lambda: J2('engines/summary/user.j2')), kw_only=True)` class-attribute instance-attribute
- `max_chunker_tokens` property
- `max_token_multiplier = field(default=0.5, kw_only=True)` class-attribute instance-attribute
- `min_response_tokens` property
- `prompt_driver = field(default=Factory(lambda: Defaults.drivers_config.prompt_driver), kw_only=True)` class-attribute instance-attribute

summarize_artifacts(artifacts, *, rulesets=None)

Source Code in griptape/engines/summary/prompt_summary_engine.py

```
def summarize_artifacts(self, artifacts: ListArtifact, *, rulesets:␣
↪Optional[list[Ruleset]] = None) -> TextArtifact:
    return self.summarize_artifacts_rec(cast("list[TextArtifact]", artifacts.value),␣
↪None, rulesets=rulesets)
```

summarize_artifacts_rec(artifacts, summary=None, rulesets=None)

Source Code in griptape/engines/summary/prompt_summary_engine.py

```
def summarize_artifacts_rec(
    self,
    artifacts: list[TextArtifact],
    summary: Optional[str] = None,
    rulesets: Optional[list[Ruleset]] = None,
) -> TextArtifact:
    if not artifacts:
        if summary is None:
            raise ValueError("No artifacts to summarize")
        return TextArtifact(summary)

    artifacts_text = self.chunk_joiner.join([a.to_text() for a in artifacts])

    system_prompt = self.generate_system_template.render(
        summary=summary,
        rulesets=J2("rulesets/rulesets.j2").render(rulesets=rulesets),
    )

    user_prompt = self.generate_user_template.render(text=artifacts_text)

    if (
        self.prompt_driver.tokenizer.count_input_tokens_left(user_prompt + system_
↪prompt)
        >= self.min_response_tokens
    ):
        result = self.prompt_driver.run(
            PromptStack(
                messages=[
                    Message(system_prompt, role=Message.SYSTEM_ROLE),
                    Message(user_prompt, role=Message.USER_ROLE),
                ],
            ),
        ).to_artifact()

        if isinstance(result, TextArtifact):
            return result
        raise ValueError("Prompt driver did not return a TextArtifact")
    chunks = self.chunker.chunk(artifacts_text)

    partial_text = self.generate_user_template.render(text=chunks[0].value)
```

(continues on next page)

(continued from previous page)

```
    return self.summarize_artifacts_rec(
        chunks[1:],
        self.prompt_driver.run(
            PromptStack(
                messages=[
                    Message(system_prompt, role=Message.SYSTEM_ROLE),
                    Message(partial_text, role=Message.USER_ROLE),
                ],
            ),
        ).value,
        rulesets=rulesets,
    )
```

validate*allowlist*(, max_token_multiplier)

Source Code in griptape/engines/summary/prompt_summary_engine.py

```
@max_token_multiplier.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_allowlist(self, _: Attribute, max_token_multiplier: int) -> None:
    if max_token_multiplier > 1:
        raise ValueError("has to be less than or equal to 1")
    if max_token_multiplier <= 0:
        raise ValueError("has to be greater than 0")
```

11.5.9 events

- `EventBus = _EventBus()` module-attribute
- `__all__ = ['ActionChunkEvent', 'AudioChunkEvent', 'BaseActionsSubtaskEvent', 'BaseAudioTranscriptionEvent', 'BaseChunkEvent', 'BaseEvent', 'BasePromptEvent', 'BaseTaskEvent', 'BaseTextToSpeechEvent', 'EventBus', 'EventListener', 'FinishActionsSubtaskEvent', 'FinishAudioTranscriptionEvent', 'FinishImageGenerationEvent', 'FinishImageQueryEvent', 'FinishPromptEvent', 'FinishStructureRunEvent', 'FinishTaskEvent', 'FinishTextToSpeechEvent', 'StartActionsSubtaskEvent', 'StartAudioTranscriptionEvent', 'StartImageGenerationEvent', 'StartImageQueryEvent', 'StartPromptEvent', 'StartStructureRunEvent', 'StartTaskEvent', 'StartTextToSpeechEvent', 'TextChunkEvent']` module-attribute

Bases: *BaseChunkEvent* “)

Source Code in griptape/events/action_chunk_event.py

```
@define
class ActionChunkEvent(BaseChunkEvent):
    partial_input: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    tag: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    name: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    path: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
```

(continues on next page)

(continued from previous page)

```
↪True})

    def __str__(self) -> str:
        parts = []

        if self.name:
            parts.append(self.name)
            if self.path:
                parts.append(f".{self.path}")
                if self.tag:
                    parts.append(f" ({self.tag})")

        if self.partial_input:
            if parts:
                parts.append(f"\n{self.partial_input}")
            else:
                parts.append(self.partial_input)

        return "".join(parts)
```

- `name = field(default=None, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute
- `partial_input = field(default=None, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute
- `path = field(default=None, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute
- `tag = field(default=None, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute

str()

Source Code in griptape/events/action_chunk_event.py

```
def __str__(self) -> str:
    parts = []

    if self.name:
        parts.append(self.name)
        if self.path:
            parts.append(f".{self.path}")
            if self.tag:
                parts.append(f" ({self.tag})")

    if self.partial_input:
        if parts:
            parts.append(f"\n{self.partial_input}")
        else:
            parts.append(self.partial_input)

    return "".join(parts)
```

AudioChunkEvent

Bases: *BaseChunkEvent* “)

Attributes

Name	Type	Description
<a href="./#griptape.events.AudioChunkEvent.data" title="data = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.events.AudioChunkEvent.data)"><code class="language-python">	*str*	Base64 encoded audio data.

Source Code in griptape/events/audio_chunk_event.py

```
@define
class AudioChunkEvent(BaseChunkEvent):
    """Stores a chunk of audio data.

    Attributes:
        data: Base64 encoded audio data.
    """

    data: str = field(kw_only=True, metadata={"serializable": True})

    def __str__(self) -> str:
        return self.data
```

- `data = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

str()

Source Code in griptape/events/audio_chunk_event.py

```
def __str__(self) -> str:
    return self.data
```

BaseActionsSubtaskEvent

Bases: *BaseTaskEvent* “) , ABC

Source Code in griptape/events/base_actions_subtask_event.py

```
@define
class BaseActionsSubtaskEvent(BaseTaskEvent, ABC):
    subtask_parent_task_id: Optional[str] = field(kw_only=True, metadata={
↪"serializable": True})
    subtask_thought: Optional[str] = field(kw_only=True, metadata={"serializable":␣
↪True})
    subtask_actions: Optional[list[dict]] = field(kw_only=True, metadata={
↪"serializable": True})

    @classmethod
    def from_task(cls, task: BaseTask) -> BaseActionsSubtaskEvent:
        from griptape.tasks import ActionsSubtask

        if not isinstance(task, ActionsSubtask):
            raise ValueError("Event must be of instance ActionSubtask.")
        return cls(
            task_id=task.id,
            task_parent_ids=task.parent_ids,
            task_child_ids=task.child_ids,
            task_input=task.input,
            task_output=task.output,
            subtask_parent_task_id=task.origin_task.id,
            subtask_thought=task.thought,
            subtask_actions=task.actions_to_dicts(),
        )
```

- `subtask_actions = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `subtask_parent_task_id = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `subtask_thought = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

from_task(task)classmethod

Source Code in griptape/events/base_actions_subtask_event.py

```
@classmethod
def from_task(cls, task: BaseTask) -> BaseActionsSubtaskEvent:
    from griptape.tasks import ActionsSubtask

    if not isinstance(task, ActionsSubtask):
        raise ValueError("Event must be of instance ActionSubtask.")
    return cls(
        task_id=task.id,
        task_parent_ids=task.parent_ids,
        task_child_ids=task.child_ids,
        task_input=task.input,
        task_output=task.output,
        subtask_parent_task_id=task.origin_task.id,
        subtask_thought=task.thought,
        subtask_actions=task.actions_to_dicts(),
```

(continues on next page)

(continued from previous page)

```
    )
```

BaseAudioTranscriptionEvent

Bases: Return `BaseEvent or *dict* <Return *BaseEvent* or *dict*>` “)

Source Code in griptape/events/base_audio_transcription_event.py

```
class BaseAudioTranscriptionEvent(BaseEvent): ...
```

BaseChunkEvent

Bases: Return `BaseEvent or *dict* <Return *BaseEvent* or *dict*>` “)

Source Code in griptape/events/base_chunk_event.py

```
@define
class BaseChunkEvent(BaseEvent):
    index: int = field(default=0, metadata={"serializable": True})

    @abstractmethod
    def __str__(self) -> str: ...
```

- `index = field(default=0, metadata={'serializable': True})` class-attribute instance-attribute

str()abstractmethod

Source Code in griptape/events/base_chunk_event.py

```
@abstractmethod
def __str__(self) -> str: ...
```

BaseEvent

Bases: *SerializableMixin* “) , ABC

Source Code in griptape/events/base_event.py

```
@define
class BaseEvent(SerializableMixin, ABC):
    id: str = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata=
↪{"serializable": True})
    timestamp: float = field(default=Factory(lambda: time.time()), kw_only=True,␣
↪metadata={"serializable": True})
    meta: dict[str, Any] = field(factory=dict, kw_only=True, metadata={"serializable
↪": True})
```

- `id = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `meta = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `timestamp = field(default=Factory(lambda: time.time()), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

BasePromptEvent

Bases: Return `BaseEvent or *dict* <Return *BaseEvent* or *dict*>` “) , ABC

Source Code in griptape/events/base_prompt_event.py

```
@define
class BasePromptEvent(BaseEvent, ABC):
    model: str = field(kw_only=True, metadata={"serializable": True})
```

- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

BaseTaskEvent

Bases: Return `BaseEvent or *dict* <Return *BaseEvent* or *dict*>` “) , ABC

Source Code in griptape/events/base_task_event.py

```
@define
class BaseTaskEvent(BaseEvent, ABC):
    task_id: str = field(kw_only=True, metadata={"serializable": True})
    task_parent_ids: list[str] = field(kw_only=True, metadata={"serializable": True})
    task_child_ids: list[str] = field(kw_only=True, metadata={"serializable": True})

    task_input: BaseArtifact = field(kw_only=True, metadata={"serializable": True})
    task_output: Optional[BaseArtifact] = field(kw_only=True, metadata={"serializable
↪": True})
```

- `task_child_ids = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `task_id = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `task_input = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `task_output = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `task_parent_ids = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

BaseTextToSpeechEvent

Bases: *BaseMediaGenerationEvent* “) , ABC

Source Code in griptape/events/base_text_to_speech_event.py

```
@define
class BaseTextToSpeechEvent(BaseMediaGenerationEvent, ABC): ...
```

EventListener

Bases: *Generic[T*

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">Optional[Callable[[T], Optional[BaseEvent I dict]]]</code>	The on_event function that will be called when an event is published. The on_event function should accept an event and return either the event or a dictionary. If the on_event returns None, the event will not be published.
<a href="./#griptape.events.EventListener.event_types" title="event_types = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.events.EventListener.event_types)"><cod class="language-python">	<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">Optional[list[type[T]]]</cod	A list of event types that the event listener should listen for. If not provided, the event listener will listen for all event types.
<a href="./#griptape.events.EventListener.event_listener_driver" title="event_listener_driver = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.events.EventListener.event_listener_driver)"><code class="language-python">	<a href="../drivers/event_listener#griptape.drivers.event_listener.BaseEventListenerDriver" title="BaseEventListenerDriver (griptape.drivers.event_listener.BaseEventListenerDriver)"><code class="language-python">Optional[BaseEventListener	The driver that will be used to publish events.

Source Code in griptape/events/event_listener.py

```
@define
class EventListener(Generic[T]):
    """An event listener that listens for events and handles them.

    Attributes:
        on_event: The on_event function that will be called when an event is␣
↪published.
```

(continues on next page)

(continued from previous page)

```
            The on_event function should accept an event and return either the event␣
↪or a dictionary.
            If the on_event returns None, the event will not be published.
        event_types: A list of event types that the event listener should listen for.
            If not provided, the event listener will listen for all event types.
        event_listener_driver: The driver that will be used to publish events.
    """

    on_event: Optional[Callable[[T], Optional[BaseEvent | dict]]] =␣
↪field(default=None)
    event_types: Optional[list[type[T]]] = field(default=None, kw_only=True)
    event_listener_driver: Optional[BaseEventListenerDriver] = field(default=None, kw_
↪only=True)

    def __enter__(self) -> Self:
        from griptape.events import EventBus

        EventBus.add_event_listener(self)

        return self

    def __exit__(self, type, value, traceback) -> None:  # noqa: ANN001, A002
        from griptape.events import EventBus

        EventBus.remove_event_listener(self)

    def publish_event(self, event: T, *, flush: bool = False) -> None:
        event_types = self.event_types

        if event_types is None or any(isinstance(event, event_type) for event_type in␣
↪event_types):
            handled_event = event
            if self.on_event is not None:
                handled_event = self.on_event(event)

            if self.event_listener_driver is not None and handled_event is not None:
                self.event_listener_driver.publish_event(handled_event)

        if self.event_listener_driver is not None and flush:
            self.event_listener_driver.flush_events()
```

- `event_listener_driver = field(default=None, kw_only=True)` class-attribute instance-attribute
- `event_types = field(default=None, kw_only=True)` class-attribute instance-attribute
- `on_event = field(default=None)` class-attribute instance-attribute

enter()

Source Code in griptape/events/event_listener.py

```
def __enter__(self) -> Self:
    from griptape.events import EventBus
```

(continues on next page)

(continued from previous page)

```
    EventBus.add_event_listener(self)

    return self
```

exit(type, value, traceback)

Source Code in griptape/events/event_listener.py

```
def __exit__(self, type, value, traceback) -> None:  # noqa: ANN001, A002
    from griptape.events import EventBus

    EventBus.remove_event_listener(self)
```

publish_event(event, *, flush=False)

Source Code in griptape/events/event_listener.py

```
def publish_event(self, event: T, *, flush: bool = False) -> None:
    event_types = self.event_types

    if event_types is None or any(isinstance(event, event_type) for event_type in␣
↪event_types):
        handled_event = event
        if self.on_event is not None:
            handled_event = self.on_event(event)

        if self.event_listener_driver is not None and handled_event is not None:
            self.event_listener_driver.publish_event(handled_event)

    if self.event_listener_driver is not None and flush:
        self.event_listener_driver.flush_events()
```

FinishActionsSubtaskEvent

Bases: *BaseActionsSubtaskEvent* “)

Source Code in griptape/events/finish_actions_subtask_event.py

```
@define
class FinishActionsSubtaskEvent(BaseActionsSubtaskEvent): ...
```

FinishAudioTranscriptionEvent

Bases: *BaseAudioTranscriptionEvent* “)

Source Code in griptape/events/finish_audio_transcription_event.py

```
class FinishAudioTranscriptionEvent(BaseAudioTranscriptionEvent): ...
```

FinishImageGenerationEvent

Bases: *BaseImageGenerationEvent* “)

Source Code in griptape/events/finish_image_generation_event.py

```
@define
class FinishImageGenerationEvent(BaseImageGenerationEvent): ...
```

FinishImageQueryEvent

Bases: *BaseImageQueryEvent* “)

Source Code in griptape/events/finish_image_query_event.py

```
@define
class FinishImageQueryEvent(BaseImageQueryEvent):
    result: str = field(kw_only=True, metadata={"serializable": True})
```

- `result = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

FinishPromptEvent

Bases: *BasePromptEvent* “)

Source Code in griptape/events/finish_prompt_event.py

```
@define
class FinishPromptEvent(BasePromptEvent):
    result: str = field(kw_only=True, metadata={"serializable": True})
    input_token_count: Optional[float] = field(kw_only=True, metadata={"serializable
↪": True})
    output_token_count: Optional[float] = field(kw_only=True, metadata={"serializable
↪": True})
```

- `input_token_count = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output_token_count = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `result = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

FinishStructureRunEvent

Bases: Return `BaseEvent or *dict* <Return *BaseEvent* or *dict*>` “)

Source Code in griptape/events/finish_structure_run_event.py

```
@define
class FinishStructureRunEvent(BaseEvent):
    structure_id: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": True})
    output_task_input: Optional[BaseArtifact] = field(kw_only=True, default=None,␣
```

(continues on next page)

(continued from previous page)

```
↪metadata={"serializable": True})
    output_task_output: Optional[BaseArtifact] = field(kw_only=True, default=None,␣
↪metadata={"serializable": True})
```

- `output_task_input = field(kw_only=True, default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `output_task_output = field(kw_only=True, default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `structure_id = field(kw_only=True, default=None, metadata={'serializable': True})` class-attribute instance-attribute

FinishTaskEvent

Bases: *BaseTaskEvent* “)

Source Code in griptape/events/finish_task_event.py

```
@define
class FinishTaskEvent(BaseTaskEvent): ...
```

FinishTextToSpeechEvent

Bases: *BaseTextToSpeechEvent* “)

Source Code in griptape/events/finish_text_to_speech_event.py

```
@define
class FinishTextToSpeechEvent(BaseTextToSpeechEvent): ...
```

StartActionsSubtaskEvent

Bases: *BaseActionsSubtaskEvent* “)

Source Code in griptape/events/start_actions_subtask_event.py

```
@define
class StartActionsSubtaskEvent(BaseActionsSubtaskEvent): ...
```

StartAudioTranscriptionEvent

Bases: *BaseAudioTranscriptionEvent* “)

Source Code in griptape/events/start_audio_transcription_event.py

```
class StartAudioTranscriptionEvent(BaseAudioTranscriptionEvent): ...
```

StartImageGenerationEvent

Bases: *BaseImageGenerationEvent* “)

Source Code in griptape/events/start_image_generation_event.py

```
@define
class StartImageGenerationEvent(BaseImageGenerationEvent):
    prompts: list[str] = field(kw_only=True, metadata={"serializable": True})
    negative_prompts: Optional[list[str]] = field(default=None, kw_only=True,␣
↪metadata={"serializable": True})
```

- `negative_prompts = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `prompts = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

StartImageQueryEvent

Bases: *BaseImageQueryEvent* “)

Source Code in griptape/events/start_image_query_event.py

```
@define
class StartImageQueryEvent(BaseImageQueryEvent):
    query: str = field(kw_only=True, metadata={"serializable": True})
    images_info: list[str] = field(kw_only=True, metadata={"serializable": True})
```

- `images_info = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `query = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

StartPromptEvent

Bases: *BasePromptEvent* “)

Source Code in griptape/events/start_prompt_event.py

```
@define
class StartPromptEvent(BasePromptEvent):
    prompt_stack: PromptStack = field(kw_only=True, metadata={"serializable": True})
```

- `prompt_stack = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

StartStructureRunEvent

Bases: Return `BaseEvent or *dict* <Return *BaseEvent* or *dict*>` “)

Source Code in griptape/events/start_structure_run_event.py

```
@define
class StartStructureRunEvent(BaseEvent):
    structure_id: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": True})
    input_task_input: BaseArtifact = field(kw_only=True, metadata={"serializable":␣
↪True})
    input_task_output: Optional[BaseArtifact] = field(kw_only=True, metadata={
↪"serializable": True})
```

- `input_task_input = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `input_task_output = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structure_id = field(kw_only=True, default=None, metadata={'serializable': True})` class-attribute instance-attribute

StartTaskEvent

Bases: *BaseTaskEvent* “)

Source Code in griptape/events/start_task_event.py

```
@define
class StartTaskEvent(BaseTaskEvent): ...
```

StartTextToSpeechEvent

Bases: *BaseTextToSpeechEvent* “)

Source Code in griptape/events/start_text_to_speech_event.py

```
@define
class StartTextToSpeechEvent(BaseTextToSpeechEvent):
    prompts: list[str] = field(kw_only=True, metadata={"serializable": True})
```

- `prompts = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

TextChunkEvent

Bases: *BaseChunkEvent* “)

Source Code in griptape/events/text_chunk_event.py

```
@define
class TextChunkEvent(BaseChunkEvent):
    token: str = field(kw_only=True, metadata={"serializable": True})

    def __str__(self) -> str:
        return self.token
```

- `token = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

str()

Source Code in griptape/events/text_chunk_event.py

```
def __str__(self) -> str:
    return self.token
```

action_chunk_event

Bases: *BaseChunkEvent* “)

Source Code in griptape/events/action_chunk_event.py

```
@define
class ActionChunkEvent(BaseChunkEvent):
    partial_input: Optional[str] = field(default=None, kw_only=True, metadata={
↪"serializable": True})
    tag: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    name: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})
    path: Optional[str] = field(default=None, kw_only=True, metadata={"serializable":␣
↪True})

    def __str__(self) -> str:
        parts = []

        if self.name:
            parts.append(self.name)
            if self.path:
                parts.append(f".{self.path}")
                if self.tag:
                    parts.append(f" ({self.tag})")

        if self.partial_input:
            if parts:
                parts.append(f"\n{self.partial_input}")
            else:
                parts.append(self.partial_input)

        return "".join(parts)
```

- `name = field(default=None, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute
- `partial_input = field(default=None, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute
- `path = field(default=None, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute
- `tag = field(default=None, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute

str()

Source Code in griptape/events/action_chunk_event.py

```
def __str__(self) -> str:
    parts = []

    if self.name:
        parts.append(self.name)
```

(continues on next page)

(continued from previous page)

```
        if self.path:
            parts.append(f".{self.path}")
            if self.tag:
                parts.append(f" ({self.tag})")

    if self.partial_input:
        if parts:
            parts.append(f"\n{self.partial_input}")
        else:
            parts.append(self.partial_input)

    return "".join(parts)
```

audio_chunk_event

Bases: *BaseChunkEvent* “)

Attributes

Name	Type	Description
<a href="audio_chunk_event.mdx#griptape.events.audio_chunk_event.AudioChunkEvent.data" title="data = field(kw_only=True, metadata={'serializable': True})" class="attribute instance-attribute">(griptape.events.audio_chunk_event.AudioChunkEvent.data)"><code class="language-python">	*str*	Base64 encoded audio data.

Source Code in griptape/events/audio_chunk_event.py

```
@define
class AudioChunkEvent(BaseChunkEvent):
    """Stores a chunk of audio data.

    Attributes:
        data: Base64 encoded audio data.
    """

    data: str = field(kw_only=True, metadata={"serializable": True})
```

(continues on next page)

(continued from previous page)

```
    def __str__(self) -> str:
        return self.data
```

- `data = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

str()

Source Code in griptape/events/audio_chunk_event.py

```
def __str__(self) -> str:
    return self.data
```

base_actions_subtask_event

Bases: *BaseTaskEvent* “) , ABC

Source Code in griptape/events/base_actions_subtask_event.py

```
@define
class BaseActionsSubtaskEvent(BaseTaskEvent, ABC):
    subtask_parent_task_id: Optional[str] = field(kw_only=True, metadata={
↪"serializable": True})
    subtask_thought: Optional[str] = field(kw_only=True, metadata={"serializable":␣
↪True})
    subtask_actions: Optional[list[dict]] = field(kw_only=True, metadata={
↪"serializable": True})

    @classmethod
    def from_task(cls, task: BaseTask) -> BaseActionsSubtaskEvent:
        from griptape.tasks import ActionsSubtask

        if not isinstance(task, ActionsSubtask):
            raise ValueError("Event must be of instance ActionSubtask.")
        return cls(
            task_id=task.id,
            task_parent_ids=task.parent_ids,
            task_child_ids=task.child_ids,
            task_input=task.input,
            task_output=task.output,
            subtask_parent_task_id=task.origin_task.id,
            subtask_thought=task.thought,
            subtask_actions=task.actions_to_dicts(),
        )
```

- `subtask_actions = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `subtask_parent_task_id = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `subtask_thought = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

from_task(task)classmethod

Source Code in griptape/events/base_actions_subtask_event.py

```
@classmethod
def from_task(cls, task: BaseTask) -> BaseActionsSubtaskEvent:
    from griptape.tasks import ActionsSubtask

    if not isinstance(task, ActionsSubtask):
        raise ValueError("Event must be of instance ActionSubtask.")
    return cls(
        task_id=task.id,
        task_parent_ids=task.parent_ids,
        task_child_ids=task.child_ids,
        task_input=task.input,
        task_output=task.output,
        subtask_parent_task_id=task.origin_task.id,
        subtask_thought=task.thought,
        subtask_actions=task.actions_to_dicts(),
    )
```

base_audio_transcription_event

Bases: Return `BaseEvent or *dict* <Return *BaseEvent* or *dict*>` “)

Source Code in griptape/events/base_audio_transcription_event.py

```
class BaseAudioTranscriptionEvent(BaseEvent): ...
```

base_chunk_event

Bases: Return `BaseEvent or *dict* <Return *BaseEvent* or *dict*>` “)

Source Code in griptape/events/base_chunk_event.py

```
@define
class BaseChunkEvent(BaseEvent):
    index: int = field(default=0, metadata={"serializable": True})

    @abstractmethod
    def __str__(self) -> str: ...
```

- `index = field(default=0, metadata={'serializable': True})` class-attribute instance-attribute

str()abstractmethod

Source Code in griptape/events/base_chunk_event.py

```
@abstractmethod
def __str__(self) -> str: ...
```

base_event

Bases: *SerializableMixin* “) , ABC

Source Code in griptape/events/base_event.py

```
@define
class BaseEvent(SerializableMixin, ABC):
    id: str = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata=
↪{"serializable": True})
    timestamp: float = field(default=Factory(lambda: time.time()), kw_only=True,␣
↪metadata={"serializable": True})
    meta: dict[str, Any] = field(factory=dict, kw_only=True, metadata={"serializable
↪": True})
```

- `id = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `meta = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `timestamp = field(default=Factory(lambda: time.time()), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

base_image_generation_event

Bases: *BaseMediaGenerationEvent* “) , ABC

Source Code in griptape/events/base_image_generation_event.py

```
@define
class BaseImageGenerationEvent(BaseMediaGenerationEvent, ABC): ...
```

base_image_query_event

Bases: Return `BaseEvent or *dict* <Return *BaseEvent* or *dict*>` “) , ABC

Source Code in griptape/events/base_image_query_event.py

```
@define
class BaseImageQueryEvent(BaseEvent, ABC): ...
```

base_media_generation_event

Bases: Return `BaseEvent or *dict* <Return *BaseEvent* or *dict*>` “) , ABC

Source Code in griptape/events/base_media_generation_event.py

```
@define
class BaseMediaGenerationEvent(BaseEvent, ABC): ...
```

base_prompt_event

Bases: Return `BaseEvent or *dict* <Return *BaseEvent* or *dict*>` “) , ABC

Source Code in griptape/events/base_prompt_event.py

```
@define
class BasePromptEvent(BaseEvent, ABC):
    model: str = field(kw_only=True, metadata={"serializable": True})
```

- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

base_task_event

Bases: Return `BaseEvent or *dict* <Return *BaseEvent* or *dict*>` “) , ABC

Source Code in griptape/events/base_task_event.py

```
@define
class BaseTaskEvent(BaseEvent, ABC):
    task_id: str = field(kw_only=True, metadata={"serializable": True})
    task_parent_ids: list[str] = field(kw_only=True, metadata={"serializable": True})
    task_child_ids: list[str] = field(kw_only=True, metadata={"serializable": True})

    task_input: BaseArtifact = field(kw_only=True, metadata={"serializable": True})
    task_output: Optional[BaseArtifact] = field(kw_only=True, metadata={"serializable
↪": True})
```

- `task_child_ids = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `task_id = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `task_input = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `task_output = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `task_parent_ids = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

base_text_to_speech_event

Bases: *BaseMediaGenerationEvent* “) , ABC

Source Code in griptape/events/base_text_to_speech_event.py

```
@define
class BaseTextToSpeechEvent(BaseMediaGenerationEvent, ABC): ...
```

event_bus

- `EventBus = _EventBus()` module-attribute
- `_event_listeners = ContextVar('event_listeners', default=None)` module-attribute

Bases: *SingletonMixin* “)

Source Code in griptape/events/event_bus.py

```
@define
class _EventBus(SingletonMixin):
    @property
    def event_listeners(self) -> list[EventListener]:
        event_listeners_val = _event_listeners.get()
        if event_listeners_val is None:
            event_listeners_val = []
            _event_listeners.set(event_listeners_val)
        return event_listeners_val

    @event_listeners.setter
    def event_listeners(self, event_listeners: list[EventListener]) -> None:
        _event_listeners.set(event_listeners)

    def add_event_listeners(self, event_listeners: list[EventListener]) ->␣
↪list[EventListener]:
        return [self.add_event_listener(event_listener) for event_listener in event_
↪listeners]

    def remove_event_listeners(self, event_listeners: list[EventListener]) -> None:
        for event_listener in event_listeners:
            self.remove_event_listener(event_listener)

    def add_event_listener(self, event_listener: EventListener) -> EventListener:
        if event_listener not in self.event_listeners:
            self.event_listeners = self.event_listeners + [event_listener]

        return event_listener

    def remove_event_listener(self, event_listener: EventListener) -> None:
        if event_listener in self.event_listeners:
            self.event_listeners = [listener for listener in self.event_listeners if␣
↪listener != event_listener]

    def publish_event(self, event: BaseEvent, *, flush: bool = False) -> None:
        for event_listener in self.event_listeners:
            event_listener.publish_event(event, flush=flush)

    def clear_event_listeners(self) -> None:
        self.event_listeners = []
```

- `event_listeners` property writable

add_event_listener(event_listener)

Source Code in griptape/events/event_bus.py

```
def add_event_listener(self, event_listener: EventListener) -> EventListener:
    if event_listener not in self.event_listeners:
        self.event_listeners = self.event_listeners + [event_listener]

    return event_listener
```

add_event_listeners(event_listeners)

Source Code in griptape/events/event_bus.py

```
def add_event_listeners(self, event_listeners: list[EventListener]) ->␣
↪list[EventListener]:
    return [self.add_event_listener(event_listener) for event_listener in event_
↪listeners]
```

clear_event_listeners()

Source Code in griptape/events/event_bus.py

```
def clear_event_listeners(self) -> None:
    self.event_listeners = []
```

publish_event(event, *, flush=False)

Source Code in griptape/events/event_bus.py

```
def publish_event(self, event: BaseEvent, *, flush: bool = False) -> None:
    for event_listener in self.event_listeners:
        event_listener.publish_event(event, flush=flush)
```

remove_event_listener(event_listener)

Source Code in griptape/events/event_bus.py

```
def remove_event_listener(self, event_listener: EventListener) -> None:
    if event_listener in self.event_listeners:
        self.event_listeners = [listener for listener in self.event_listeners if␣
↪listener != event_listener]
```

remove_event_listeners(event_listeners)

Source Code in griptape/events/event_bus.py

```
def remove_event_listeners(self, event_listeners: list[EventListener]) -> None:
    for event_listener in event_listeners:
        self.remove_event_listener(event_listener)
```

event_listener

- `T = TypeVar('T', bound=BaseEvent)` module-attribute

Bases: *Generic[T*

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">Optional[Callable[[T], Optional[BaseEvent I dict]]]</code>	The on_event function that will be called when an event is published. The on_event function should accept an event and return either the event or a dictionary. If the on_event returns None, the event will not be published.
<a href="event_listener.mdx#griptape.events.event_listener.EventListener.event_types" title="event_types = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.events.event_listener.EventListener.event_types)"><cod class="language-python">	<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">Optional[list[type[T]]]</cod	A list of event types that the event listener should listen for. If not provided, the event listener will listen for all event types.
<a href="event_listener.mdx#griptape.events.event_listener.EventListener.event_listener_driver" title="event_listener_driver = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.events.event_listener.EventListener.event_listener_driver)"><code class="language-python">	<a href="../drivers/event_listener#griptape.drivers.event_listener.BaseEventListenerDriver" title="BaseEventListenerDriver (griptape.drivers.event_listener.BaseEventListenerDriver)"><code class="language-python">Optional[BaseEventListener	The driver that will be used to publish events.

Source Code in griptape/events/event_listener.py

```
@define
class EventListener(Generic[T]):
    """An event listener that listens for events and handles them.
```

(continues on next page)

(continued from previous page)

```
    Attributes:
        on_event: The on_event function that will be called when an event is␣
↪published.
            The on_event function should accept an event and return either the event␣
↪or a dictionary.
            If the on_event returns None, the event will not be published.
        event_types: A list of event types that the event listener should listen for.
            If not provided, the event listener will listen for all event types.
        event_listener_driver: The driver that will be used to publish events.
    """

    on_event: Optional[Callable[[T], Optional[BaseEvent | dict]]] =␣
↪field(default=None)
    event_types: Optional[list[type[T]]] = field(default=None, kw_only=True)
    event_listener_driver: Optional[BaseEventListenerDriver] = field(default=None, kw_
↪only=True)

    def __enter__(self) -> Self:
        from griptape.events import EventBus

        EventBus.add_event_listener(self)

        return self

    def __exit__(self, type, value, traceback) -> None:  # noqa: ANN001, A002
        from griptape.events import EventBus

        EventBus.remove_event_listener(self)

    def publish_event(self, event: T, *, flush: bool = False) -> None:
        event_types = self.event_types

        if event_types is None or any(isinstance(event, event_type) for event_type in␣
↪event_types):
            handled_event = event
            if self.on_event is not None:
                handled_event = self.on_event(event)

            if self.event_listener_driver is not None and handled_event is not None:
                self.event_listener_driver.publish_event(handled_event)

        if self.event_listener_driver is not None and flush:
            self.event_listener_driver.flush_events()
```

- `event_listener_driver = field(default=None, kw_only=True)` class-attribute instance-attribute
- `event_types = field(default=None, kw_only=True)` class-attribute instance-attribute
- `on_event = field(default=None)` class-attribute instance-attribute

enter()

Source Code in griptape/events/event_listener.py

```
def __enter__(self) -> Self:
    from griptape.events import EventBus

    EventBus.add_event_listener(self)

    return self
```

exit(type, value, traceback)

Source Code in griptape/events/event_listener.py

```
def __exit__(self, type, value, traceback) -> None:  # noqa: ANN001, A002
    from griptape.events import EventBus

    EventBus.remove_event_listener(self)
```

publish_event(event, *, flush=False)

Source Code in griptape/events/event_listener.py

```
def publish_event(self, event: T, *, flush: bool = False) -> None:
    event_types = self.event_types

    if event_types is None or any(isinstance(event, event_type) for event_type in␣
↪event_types):
        handled_event = event
        if self.on_event is not None:
            handled_event = self.on_event(event)

        if self.event_listener_driver is not None and handled_event is not None:
            self.event_listener_driver.publish_event(handled_event)

    if self.event_listener_driver is not None and flush:
        self.event_listener_driver.flush_events()
```

finish_actions_subtask_event

Bases: *BaseActionsSubtaskEvent* “)

Source Code in griptape/events/finish_actions_subtask_event.py

```
@define
class FinishActionsSubtaskEvent(BaseActionsSubtaskEvent): ...
```

finish_audio_transcription_event

Bases: *BaseAudioTranscriptionEvent* “)

Source Code in griptape/events/finish_audio_transcription_event.py

```
class FinishAudioTranscriptionEvent(BaseAudioTranscriptionEvent): ...
```

finish_image_generation_event

Bases: *BaseImageGenerationEvent* “)

Source Code in griptape/events/finish_image_generation_event.py

```
@define
class FinishImageGenerationEvent(BaseImageGenerationEvent): ...
```

finish_image_query_event

Bases: *BaseImageQueryEvent* “)

Source Code in griptape/events/finish_image_query_event.py

```
@define
class FinishImageQueryEvent(BaseImageQueryEvent):
    result: str = field(kw_only=True, metadata={"serializable": True})
```

- `result = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

finish_prompt_event

Bases: *BasePromptEvent* “)

Source Code in griptape/events/finish_prompt_event.py

```
@define
class FinishPromptEvent(BasePromptEvent):
    result: str = field(kw_only=True, metadata={"serializable": True})
    input_token_count: Optional[float] = field(kw_only=True, metadata={"serializable
↪": True})
    output_token_count: Optional[float] = field(kw_only=True, metadata={"serializable
↪": True})
```

- `input_token_count = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output_token_count = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `result = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

finish_structure_run_event

Bases: Return `BaseEvent or *dict* <Return *BaseEvent* or *dict*>` “)

Source Code in griptape/events/finish_structure_run_event.py

```
@define
class FinishStructureRunEvent(BaseEvent):
```

(continues on next page)

(continued from previous page)

```
    structure_id: Optional[str] = field(kw_only=True, default=None, metadata={
↪"serializable": True})
    output_task_input: Optional[BaseArtifact] = field(kw_only=True, default=None,␣
↪metadata={"serializable": True})
    output_task_output: Optional[BaseArtifact] = field(kw_only=True, default=None,␣
↪metadata={"serializable": True})
```

- `output_task_input = field(kw_only=True, default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `output_task_output = field(kw_only=True, default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `structure_id = field(kw_only=True, default=None, metadata={'serializable': True})` class-attribute instance-attribute

finish_task_event

Bases: *BaseTaskEvent* “)

Source Code in griptape/events/finish_task_event.py

```
@define
class FinishTaskEvent(BaseTaskEvent): ...
```

finish_text_to_speech_event

Bases: *BaseTextToSpeechEvent* “)

Source Code in griptape/events/finish_text_to_speech_event.py

```
@define
class FinishTextToSpeechEvent(BaseTextToSpeechEvent): ...
```

start_actions_subtask_event

Bases: *BaseActionsSubtaskEvent* “)

Source Code in griptape/events/start_actions_subtask_event.py

```
@define
class StartActionsSubtaskEvent(BaseActionsSubtaskEvent): ...
```

start_audio_transcription_event

Bases: *BaseAudioTranscriptionEvent* “)

Source Code in griptape/events/start_audio_transcription_event.py

```
class StartAudioTranscriptionEvent(BaseAudioTranscriptionEvent): ...
```

start_image_generation_event

Bases: *BaseImageGenerationEvent* “)

Source Code in griptape/events/start_image_generation_event.py

```
@define
class StartImageGenerationEvent(BaseImageGenerationEvent):
    prompts: list[str] = field(kw_only=True, metadata={"serializable": True})
    negative_prompts: Optional[list[str]] = field(default=None, kw_only=True,␣
↪metadata={"serializable": True})
```

- `negative_prompts = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `prompts = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

start_image_query_event

Bases: *BaseImageQueryEvent* “)

Source Code in griptape/events/start_image_query_event.py

```
@define
class StartImageQueryEvent(BaseImageQueryEvent):
    query: str = field(kw_only=True, metadata={"serializable": True})
    images_info: list[str] = field(kw_only=True, metadata={"serializable": True})
```

- `images_info = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `query = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

start_prompt_event

Bases: *BasePromptEvent* “)

Source Code in griptape/events/start_prompt_event.py

```
@define
class StartPromptEvent(BasePromptEvent):
    prompt_stack: PromptStack = field(kw_only=True, metadata={"serializable": True})
```

- `prompt_stack = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

start_structure_run_event

Bases: Return `BaseEvent or *dict* <Return *BaseEvent* or *dict*>` “)

Source Code in griptape/events/start_structure_run_event.py

```
@define
class StartStructureRunEvent(BaseEvent):
    structure_id: Optional[str] = field(kw_only=True, default=None, metadata={
```

(continues on next page)

(continued from previous page)

```
↪"serializable": True})
    input_task_input: BaseArtifact = field(kw_only=True, metadata={"serializable":␣
↪True})
    input_task_output: Optional[BaseArtifact] = field(kw_only=True, metadata={
↪"serializable": True})
```

- `input_task_input = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `input_task_output = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structure_id = field(kw_only=True, default=None, metadata={'serializable': True})` class-attribute instance-attribute

start_task_event

Bases: *BaseTaskEvent* “)

Source Code in griptape/events/start_task_event.py

```
@define
class StartTaskEvent(BaseTaskEvent): ...
```

start_text_to_speech_event

Bases: *BaseTextToSpeechEvent* “)

Source Code in griptape/events/start_text_to_speech_event.py

```
@define
class StartTextToSpeechEvent(BaseTextToSpeechEvent):
    prompts: list[str] = field(kw_only=True, metadata={"serializable": True})
```

- `prompts = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

text_chunk_event

Bases: *BaseChunkEvent* “)

Source Code in griptape/events/text_chunk_event.py

```
@define
class TextChunkEvent(BaseChunkEvent):
    token: str = field(kw_only=True, metadata={"serializable": True})

    def __str__(self) -> str:
        return self.token
```

- `token = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

str()

Source Code in griptape/events/text_chunk_event.py

```
def __str__(self) -> str:
    return self.token
```

11.5.10 exceptions

- `__all__ = ['DummyError']` module-attribute

Bases:

Exception

Source Code in griptape/exceptions/dummy_exception.py

```
class DummyError(Exception):
    def __init__(self, dummy_class_name: str, dummy_method_name: str) -> None:
        message = (
            f"You have attempted to use a {dummy_class_name}s {dummy_method_name}␣
↪method. "
            "This likely originated from using a `DriversConfig` without providing a␣
↪Driver required for this feature."
        )

        super().__init__(message)
```

init(dummy_class_name, dummy_method_name)

Source Code in griptape/exceptions/dummy_exception.py

```
def __init__(self, dummy_class_name: str, dummy_method_name: str) -> None:
    message = (
        f"You have attempted to use a {dummy_class_name}s {dummy_method_name} method.
↪"
        "This likely originated from using a `DriversConfig` without providing a␣
↪Driver required for this feature."
    )

    super().__init__(message)
```

dummy_exception

Bases:

Exception

Source Code in griptape/exceptions/dummy_exception.py

```
class DummyError(Exception):
    def __init__(self, dummy_class_name: str, dummy_method_name: str) -> None:
        message = (
            f"You have attempted to use a {dummy_class_name}s {dummy_method_name}␣
↪method. "
```

(continues on next page)

(continued from previous page)

```
            "This likely originated from using a `DriversConfig` without providing a␣
↪Driver required for this feature."
        )

        super().__init__(message)
```

init(dummy_class_name, dummy_method_name)

Source Code in griptape/exceptions/dummy_exception.py

```
def __init__(self, dummy_class_name: str, dummy_method_name: str) -> None:
    message = (
        f"You have attempted to use a {dummy_class_name}s {dummy_method_name} method.
↪"
        "This likely originated from using a `DriversConfig` without providing a␣
↪Driver required for this feature."
    )

    super().__init__(message)
```

11.5.11 loaders

- `__all__ = ['AudioLoader', 'BaseFileLoader', 'BaseLoader', 'BlobLoader', 'CsvLoader', 'EmailLoader', 'ImageLoader', 'JsonLoader', 'PdfLoader', 'SqlLoader', 'TextLoader', 'WebLoader']` module-attribute

Bases: *BaseFileLoader[AudioArtifact* “)

Source Code in griptape/loaders/audio_loader.py

```
@define
class AudioLoader(BaseFileLoader[AudioArtifact]):
    """Loads audio content into audio artifacts."""

    def try_parse(self, data: bytes) -> AudioArtifact:
        filetype_guess = filetype.guess(data)
        if filetype_guess is None:
            raise ValueError("Could not determine the file type of the audio data")
        return AudioArtifact(data, format=filetype_guess.extension)
```

try_parse(data)

Source Code in griptape/loaders/audio_loader.py

```
def try_parse(self, data: bytes) -> AudioArtifact:
    filetype_guess = filetype.guess(data)
    if filetype_guess is None:
        raise ValueError("Could not determine the file type of the audio data")
    return AudioArtifact(data, format=filetype_guess.extension)
```

BaseFileLoader

Bases: *BaseLoader[Union[str, PathLike “) , ABC*

Source Code in griptape/loaders/base_file_loader.py

```
@define
class BaseFileLoader(BaseLoader[Union[str, PathLike], bytes, A], ABC):
    file_manager_driver: BaseFileManagerDriver = field(
        default=Factory(lambda: LocalFileManagerDriver()),
        kw_only=True,
    )
    encoding: str = field(default="utf-8", kw_only=True)

    def fetch(self, source: str | PathLike) -> bytes:
        # TODO: This is silly. `load_file` decodes the bytes and then we immediately␣
↪re-encode them.
        data = self.file_manager_driver.load_file(str(source)).value
        if isinstance(data, str):
            return data.encode(self.encoding)
        return data

    def save(self, destination: str | PathLike, artifact: A) -> None:
        """Saves the Artifact to a destination."""
        artifact.encoding = self.encoding
        self.file_manager_driver.save_file(str(destination), artifact.to_bytes())
```

- `encoding = field(default='utf-8', kw_only=True)` class-attribute instance-attribute
- `file_manager_driver = field(default=Factory(lambda: LocalFileManagerDriver()), kw_only=True)` class-attribute instance-attribute

fetch(source)

Source Code in griptape/loaders/base_file_loader.py

```
def fetch(self, source: str | PathLike) -> bytes:
    # TODO: This is silly. `load_file` decodes the bytes and then we immediately re-
↪encode them.
    data = self.file_manager_driver.load_file(str(source)).value
    if isinstance(data, str):
        return data.encode(self.encoding)
    return data
```

save(destination, artifact)

Source Code in griptape/loaders/base_file_loader.py

```
def save(self, destination: str | PathLike, artifact: A) -> None:
    """Saves the Artifact to a destination."""
    artifact.encoding = self.encoding
    self.file_manager_driver.save_file(str(destination), artifact.to_bytes())
```

BaseLoader

Bases: *FuturesExecutorMixin* “) , ABC, *Generic[S, F, A*

Attributes

Name	Type	Description
<a href="./#griptape.loaders.BaseLoader.reference" title="reference = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.loaders.BaseLoader.reference)"><code class="language-python">	<a href="../common#griptape.common.Reference" title="Reference (griptape.common.Reference)"><code class="language-python">Optional[Reference]</code>	The optional *Reference* to set on the Artifact.

Source Code in griptape/loaders/base_loader.py

```
@define
class BaseLoader(FuturesExecutorMixin, ABC, Generic[S, F, A]):
    """Fetches data from a source, parses it, and returns an Artifact.

    Attributes:
        reference: The optional `Reference` to set on the Artifact.
    """

    reference: Optional[Reference] = field(default=None, kw_only=True)

    def load(self, source: S) -> A:
        data = self.fetch(source)

        return self.parse(data)

    @abstractmethod
    def fetch(self, source: S) -> F:
        """Fetches data from the source."""

    def parse(self, data: F) -> A:
        """Parses the fetched data and returns an Artifact."""
        artifact = self.try_parse(data)

        artifact.reference = self.reference

        return artifact

    def try_parse(self, data: F) -> A:
        """Parses the fetched data and returns an Artifact."""
        # TODO: Mark as abstract method for griptape 2.0
```

(continues on next page)

(continued from previous page)

```
        raise NotImplementedError

    def load_collection(
        self,
        sources: list[Any],
    ) -> Mapping[str, A]:
        """Loads a collection of sources and returns a dictionary of Artifacts."""
        # Create a dictionary before actually submitting the jobs to the executor
        # to avoid duplicate work.
        sources_by_key = {self.to_key(source): source for source in sources}

        with self.create_futures_executor() as futures_executor:
            return execute_futures_dict(
                {
                    key: futures_executor.submit(with_contextvars(self.load), source)
                    for key, source in sources_by_key.items()
                },
            )

    def to_key(self, source: S) -> str:
        """Converts the source to a key for the collection."""
        if isinstance(source, bytes):
            return bytes_to_hash(source)
        return str_to_hash(str(source))
```

- `reference = field(default=None, kw_only=True)` class-attribute instance-attribute

fetch(source)abstractmethod

Source Code in griptape/loaders/base_loader.py

```
@abstractmethod
def fetch(self, source: S) -> F:
    """Fetches data from the source."""
```

load(source)

Source Code in griptape/loaders/base_loader.py

```
def load(self, source: S) -> A:
    data = self.fetch(source)

    return self.parse(data)
```

load_collection(sources)

Source Code in griptape/loaders/base_loader.py

```
def load_collection(
    self,
    sources: list[Any],
) -> Mapping[str, A]:
```

(continues on next page)

(continued from previous page)

```
    """Loads a collection of sources and returns a dictionary of Artifacts."""
    # Create a dictionary before actually submitting the jobs to the executor
    # to avoid duplicate work.
    sources_by_key = {self.to_key(source): source for source in sources}

    with self.create_futures_executor() as futures_executor:
        return execute_futures_dict(
            {
                key: futures_executor.submit(with_contextvars(self.load), source)
                for key, source in sources_by_key.items()
            },
        )
```

parse(data)

Source Code in griptape/loaders/base_loader.py

```
def parse(self, data: F) -> A:
    """Parses the fetched data and returns an Artifact."""
    artifact = self.try_parse(data)

    artifact.reference = self.reference

    return artifact
```

to_key(source)

Source Code in griptape/loaders/base_loader.py

```
def to_key(self, source: S) -> str:
    """Converts the source to a key for the collection."""
    if isinstance(source, bytes):
        return bytes_to_hash(source)
    return str_to_hash(str(source))
```

try_parse(data)

Source Code in griptape/loaders/base_loader.py

```
def try_parse(self, data: F) -> A:
    """Parses the fetched data and returns an Artifact."""
    # TODO: Mark as abstract method for griptape 2.0
    raise NotImplementedError
```

BlobLoader

Bases: *BaseFileLoader[BlobArtifact* “)

Source Code in griptape/loaders/blob_loader.py

```
@define
class BlobLoader(BaseFileLoader[BlobArtifact]):
```

(continues on next page)

(continued from previous page)

```
    def try_parse(self, data: bytes) -> BlobArtifact:
        if self.encoding is None:
            return BlobArtifact(data)
        return BlobArtifact(data, encoding=self.encoding)
```

try_parse(data)

Source Code in griptape/loaders/blob_loader.py

```
def try_parse(self, data: bytes) -> BlobArtifact:
    if self.encoding is None:
        return BlobArtifact(data)
    return BlobArtifact(data, encoding=self.encoding)
```

CsvLoader

Bases: *BaseFileLoader[ListArtifact[TextArtifact* “)

Source Code in griptape/loaders/csv_loader.py

```
@define
class CsvLoader(BaseFileLoader[ListArtifact[TextArtifact]]):
    delimiter: str = field(default=",", kw_only=True)
    encoding: str = field(default="utf-8", kw_only=True)
    format_row: Callable[[dict], str] = field(
        default=lambda value: "\n".join(f"{key}: {val}" for key, val in value.
↪items()), kw_only=True
    )

    def try_parse(self, data: bytes) -> ListArtifact[TextArtifact]:
        reader = csv.DictReader(StringIO(data.decode(self.encoding)), delimiter=self.
↪delimiter)

        return ListArtifact(
            [TextArtifact(self.format_row(row), meta={"row_num": row_num}) for row_
↪num, row in enumerate(reader)]
        )
```

- `delimiter = field(default=',', kw_only=True)` class-attribute instance-attribute
- `encoding = field(default='utf-8', kw_only=True)` class-attribute instance-attribute
- `format_row = field(default=lambda value: '\n'.join(f'{key}: {val}' for (key, val) in value.items()), kw_only=True)` class-attribute instance-attribute

try_parse(data)

Source Code in griptape/loaders/csv_loader.py

```
def try_parse(self, data: bytes) -> ListArtifact[TextArtifact]:
    reader = csv.DictReader(StringIO(data.decode(self.encoding)), delimiter=self.
↪delimiter)
```

(continues on next page)

(continued from previous page)

```
    return ListArtifact(
        [TextArtifact(self.format_row(row), meta={"row_num": row_num}) for row_num,␣
↪row in enumerate(reader)]
    )
```

EmailLoader

Bases: *BaseLoader['EmailLoader.EmailQuery', list[bytes* “)

Source Code in griptape/loaders/email_loader.py

```
@define
class EmailLoader(BaseLoader["EmailLoader.EmailQuery", list[bytes], ListArtifact]):
↪# pyright: ignore[reportGeneralTypeIssues]
    @define(frozen=True)
    class EmailQuery:
        """An email retrieval query.

        Attributes:
            label: Label to retrieve emails from such as INBOX or SENT.
            key: Optional key for filtering such as FROM or SUBJECT.
            search_criteria: Optional search criteria to filter emails by key.
            max_count: Optional max email count.
        """

        label: str = field(kw_only=True)
        key: Optional[str] = field(default=None, kw_only=True)
        search_criteria: Optional[str] = field(default=None, kw_only=True)
        max_count: Optional[int] = field(default=None, kw_only=True)

    imap_url: str = field(kw_only=True)
    username: str = field(kw_only=True)
    password: str = field(kw_only=True)

    def fetch(self, source: EmailLoader.EmailQuery) -> list[bytes]:
        label, key, search_criteria, max_count = astuple(source)

        mail_bytes = []
        with imaplib.IMAP4_SSL(self.imap_url) as client:
            client.login(self.username, self.password)

            mailbox = client.select(f"{label}", readonly=True)
            if mailbox[0] != "OK":
                raise Exception(mailbox[1][0].decode())  # pyright:␣
↪ignore[reportOptionalMemberAccess] Unsure what mailbox[1][0] is, so leaving as-is

            if key and search_criteria:
                _, [message_numbers] = client.search(None, key, f"{search_criteria}")
                messages_count = self._count_messages(message_numbers)
            elif len(mailbox) > 1 and mailbox[1] and mailbox[1][0] is not None:
                messages_count = int(mailbox[1][0])
            else:
```

(continues on next page)

(continued from previous page)

```
                raise Exception("unable to parse number of messages")

            top_n = max(0, messages_count - max_count) if max_count else 0
            for i in range(messages_count, top_n, -1):
                _result, data = client.fetch(str(i), "(RFC822)")

                if data is None or not data or data[0] is None:
                    continue

                mail_bytes.append(data[0][1])

            client.close()

        return mail_bytes

    def try_parse(self, data: list[bytes]) -> ListArtifact[TextArtifact]:
        mailparser = import_optional_dependency("mailparser")
        artifacts = []
        for byte in data:
            message = mailparser.parse_from_bytes(byte)

            # Note: mailparser only populates the text_plain field
            # if the message content type is explicitly set to text/plain.
            if message.text_plain:
                artifacts.append(TextArtifact("\n".join(message.text_plain)))

        return ListArtifact(artifacts)

    def _count_messages(self, message_numbers: bytes) -> int:
        return len(list(filter(None, message_numbers.decode().split(" "))))
```

- `imap_url = field(kw_only=True)` class-attribute instance-attribute
- `password = field(kw_only=True)` class-attribute instance-attribute
- `username = field(kw_only=True)` class-attribute instance-attribute

EmailQuery

An email retrieval query.

Attributes

Name	Type	Description
<a href="./#griptape.loaders.EmailLoader.EmailQuery.label" title="label = field(kw_only=True) class-attribute instance-attribute (griptape.loaders.EmailLoader.EmailQuery.label)"><code class="language-python">	*str*	Label to retrieve emails from such as 'INBOX' or 'SENT'.
<a href="./#griptape.loaders.EmailLoader.EmailQuery.key" title="key = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.loaders.EmailLoader.EmailQuery.key)"><code class="language-python">	*Optional[str]*	Optional key for filtering such as 'FROM' or 'SUBJECT'.
<a href="./#griptape.loaders.EmailLoader.EmailQuery.search_criteria" title="search_criteria = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.loaders.EmailLoader.EmailQuery.search_criteria)"><code class="language-python">	*Optional[str]*	Optional search criteria to filter emails by key.
<a href="./#griptape.loaders.EmailLoader.EmailQuery.max_count" title="max_count = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.loaders.EmailLoader.EmailQuery.max_count)"><coc class="language-python">	*Optional[int]*	Optional max email count.

Source Code in griptape/loaders/email_loader.py

```
@define(frozen=True)
class EmailQuery:
    """An email retrieval query.

    Attributes:
        label: Label to retrieve emails from such as INBOX or SENT.
        key: Optional key for filtering such as FROM or SUBJECT.
        search_criteria: Optional search criteria to filter emails by key.
        max_count: Optional max email count.
    """

    label: str = field(kw_only=True)
    key: Optional[str] = field(default=None, kw_only=True)
    search_criteria: Optional[str] = field(default=None, kw_only=True)
    max_count: Optional[int] = field(default=None, kw_only=True)
```

- `key = field(default=None, kw_only=True)` class-attribute instance-attribute
- `label = field(kw_only=True)` class-attribute instance-attribute
- `max_count = field(default=None, kw_only=True)` class-attribute instance-attribute
- `search_criteria = field(default=None, kw_only=True)` class-attribute instance-attribute

_count_messages(message_numbers)

Source Code in griptape/loaders/email_loader.py

```
def _count_messages(self, message_numbers: bytes) -> int:
    return len(list(filter(None, message_numbers.decode().split(" "))))
```

fetch(source)

Source Code in griptape/loaders/email_loader.py

```
def fetch(self, source: EmailLoader.EmailQuery) -> list[bytes]:
    label, key, search_criteria, max_count = astuple(source)

    mail_bytes = []
    with imaplib.IMAP4_SSL(self.imap_url) as client:
        client.login(self.username, self.password)

        mailbox = client.select(f"{label}", readonly=True)
        if mailbox[0] != "OK":
            raise Exception(mailbox[1][0].decode())  # pyright:␣
↪ignore[reportOptionalMemberAccess] Unsure what mailbox[1][0] is, so leaving as-is

        if key and search_criteria:
            _, [message_numbers] = client.search(None, key, f"{search_criteria}")
            messages_count = self._count_messages(message_numbers)
        elif len(mailbox) > 1 and mailbox[1] and mailbox[1][0] is not None:
            messages_count = int(mailbox[1][0])
        else:
            raise Exception("unable to parse number of messages")
```

(continues on next page)

(continued from previous page)

```
        top_n = max(0, messages_count - max_count) if max_count else 0
        for i in range(messages_count, top_n, -1):
            _result, data = client.fetch(str(i), "(RFC822)")

            if data is None or not data or data[0] is None:
                continue

            mail_bytes.append(data[0][1])

        client.close()

    return mail_bytes
```

try_parse(data)

Source Code in griptape/loaders/email_loader.py

```
def try_parse(self, data: list[bytes]) -> ListArtifact[TextArtifact]:
    mailparser = import_optional_dependency("mailparser")
    artifacts = []
    for byte in data:
        message = mailparser.parse_from_bytes(byte)

        # Note: mailparser only populates the text_plain field
        # if the message content type is explicitly set to text/plain.
        if message.text_plain:
            artifacts.append(TextArtifact("\n".join(message.text_plain)))

    return ListArtifact(artifacts)
```

ImageLoader

Bases: *BaseFileLoader[ImageArtifact* “)

Attributes

Name	Type	Description
<a href="./#griptape.loaders.ImageLoader.format" title="format = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.loaders.ImageLoader.format)"><code class="language-python">	*Optional[str]*	If provided, attempts to ensure image artifacts are in this format when loaded. For example, when set to 'PNG', loading image.jpg will return an ImageArtifact containing the image bytes in PNG format.

Source Code in griptape/loaders/image_loader.py

```
@define
class ImageLoader(BaseFileLoader[ImageArtifact]):
    """Loads images into image artifacts.

    Attributes:
        format: If provided, attempts to ensure image artifacts are in this format␣
↪when loaded.
                For example, when set to PNG, loading image.jpg will return an␣
↪ImageArtifact containing the image
                    bytes in PNG format.
    """

    format: Optional[str] = field(default=None, kw_only=True)

    def try_parse(self, data: bytes) -> ImageArtifact:
        pil_image = import_optional_dependency("PIL.Image")
        image = pil_image.open(BytesIO(data))

        # Normalize format only if requested.
        if self.format is not None:
            byte_stream = BytesIO()
            image.save(byte_stream, format=self.format)
            image = pil_image.open(byte_stream)
            data = byte_stream.getvalue()

        return ImageArtifact(data, format=image.format.lower(), width=image.width,␣
↪height=image.height)
```

- `format = field(default=None, kw_only=True)` class-attribute instance-attribute

try_parse(data)

Source Code in griptape/loaders/image_loader.py

```
def try_parse(self, data: bytes) -> ImageArtifact:
    pil_image = import_optional_dependency("PIL.Image")
    image = pil_image.open(BytesIO(data))
```

(continues on next page)

(continued from previous page)

```
    # Normalize format only if requested.
    if self.format is not None:
        byte_stream = BytesIO()
        image.save(byte_stream, format=self.format)
        image = pil_image.open(byte_stream)
        data = byte_stream.getvalue()

    return ImageArtifact(data, format=image.format.lower(), width=image.width,␣
↪height=image.height)
```

JsonLoader

Bases: *BaseFileLoader[JsonArtifact* “)

Source Code in griptape/loaders/json_loader.py

```
@define
class JsonLoader(BaseFileLoader[JsonArtifact]):
    def parse(self, data: bytes) -> JsonArtifact:
        return JsonArtifact(json.loads(data), encoding=self.encoding)
```

parse(data)

Source Code in griptape/loaders/json_loader.py

```
def parse(self, data: bytes) -> JsonArtifact:
    return JsonArtifact(json.loads(data), encoding=self.encoding)
```

PdfLoader

Bases: *BaseFileLoader* “)

Source Code in griptape/loaders/pdf_loader.py

```
@define
class PdfLoader(BaseFileLoader):
    def try_parse(
        self,
        data: bytes,
        *,
        password: Optional[str] = None,
    ) -> ListArtifact:
        pypdf = import_optional_dependency("pypdf")
        reader = pypdf.PdfReader(BytesIO(data), strict=True, password=password)
        pages = [TextArtifact(p.extract_text()) for p in reader.pages]

        return ListArtifact(pages)
```

try_parse(data, *, password=None)

Source Code in griptape/loaders/pdf_loader.py

```
def try_parse(
    self,
    data: bytes,
    *,
    password: Optional[str] = None,
) -> ListArtifact:
    pypdf = import_optional_dependency("pypdf")
    reader = pypdf.PdfReader(BytesIO(data), strict=True, password=password)
    pages = [TextArtifact(p.extract_text()) for p in reader.pages]

    return ListArtifact(pages)
```

SqlLoader

Bases: *BaseLoader[str, list[RowResult* “)

Source Code in griptape/loaders/sql_loader.py

```
@define
class SqlLoader(BaseLoader[str, list[BaseSqlDriver.RowResult],␣
↪ListArtifact[TextArtifact]]):
    sql_driver: BaseSqlDriver = field(kw_only=True)
    format_row: Callable[[dict], str] = field(
        default=lambda value: "\n".join(f"{key}: {val}" for key, val in value.
↪items()), kw_only=True
    )

    def fetch(self, source: str) -> list[BaseSqlDriver.RowResult]:
        return self.sql_driver.execute_query(source) or []

    def parse(self, data: list[BaseSqlDriver.RowResult]) ->␣
↪ListArtifact[TextArtifact]:
        return ListArtifact([TextArtifact(self.format_row(row.cells)) for row in␣
↪data])
```

- `format_row = field(default=lambda value: '\n'.join(f'{key}: {val}' for (key, val) in value.items()), kw_only=True)` class-attribute instance-attribute
- `sql_driver = field(kw_only=True)` class-attribute instance-attribute

fetch(source)

Source Code in griptape/loaders/sql_loader.py

```
def fetch(self, source: str) -> list[BaseSqlDriver.RowResult]:
    return self.sql_driver.execute_query(source) or []
```

parse(data)

Source Code in griptape/loaders/sql_loader.py

```
def parse(self, data: list[BaseSqlDriver.RowResult]) -> ListArtifact[TextArtifact]:
    return ListArtifact([TextArtifact(self.format_row(row.cells)) for row in data])
```

TextLoader

Bases: *BaseFileLoader[TextArtifact ")*

Source Code in griptape/loaders/text_loader.py

```
@define
class TextLoader(BaseFileLoader[TextArtifact]):
    encoding: str = field(default="utf-8", kw_only=True)

    def try_parse(self, data: str | bytes) -> TextArtifact:
        if isinstance(data, str):
            return TextArtifact(data, encoding=self.encoding)
        return TextArtifact(data.decode(self.encoding), encoding=self.encoding)
```

- `encoding = field(default='utf-8', kw_only=True)` class-attribute instance-attribute

try_parse(data)

Source Code in griptape/loaders/text_loader.py

```
def try_parse(self, data: str | bytes) -> TextArtifact:
    if isinstance(data, str):
        return TextArtifact(data, encoding=self.encoding)
    return TextArtifact(data.decode(self.encoding), encoding=self.encoding)
```

WebLoader

Bases: *BaseLoader[str, str, TextArtifact ")*

Source Code in griptape/loaders/web_loader.py

```
@define
class WebLoader(BaseLoader[str, str, TextArtifact]):
    web_scraper_driver: BaseWebScraperDriver = field(
        default=Factory(lambda: TrafilaturaWebScraperDriver()),
        kw_only=True,
    )

    def fetch(self, source: str) -> str:
        return self.web_scraper_driver.fetch_url(source)

    def try_parse(self, data: str) -> TextArtifact:
        return self.web_scraper_driver.extract_page(data)
```

- `web_scraper_driver = field(default=Factory(lambda: TrafilaturaWebScraperDriver()), kw_only=True)` class-attribute instance-attribute

fetch(source)

Source Code in griptape/loaders/web_loader.py

```
def fetch(self, source: str) -> str:
    return self.web_scraper_driver.fetch_url(source)
```

try_parse(data)

Source Code in griptape/loaders/web_loader.py

```
def try_parse(self, data: str) -> TextArtifact:
    return self.web_scraper_driver.extract_page(data)
```

audio_loader

Bases: *BaseFileLoader[AudioArtifact* “)

Source Code in griptape/loaders/audio_loader.py

```
@define
class AudioLoader(BaseFileLoader[AudioArtifact]):
    """Loads audio content into audio artifacts."""

    def try_parse(self, data: bytes) -> AudioArtifact:
        filetype_guess = filetype.guess(data)
        if filetype_guess is None:
            raise ValueError("Could not determine the file type of the audio data")
        return AudioArtifact(data, format=filetype_guess.extension)
```

try_parse(data)

Source Code in griptape/loaders/audio_loader.py

```
def try_parse(self, data: bytes) -> AudioArtifact:
    filetype_guess = filetype.guess(data)
    if filetype_guess is None:
        raise ValueError("Could not determine the file type of the audio data")
    return AudioArtifact(data, format=filetype_guess.extension)
```

base_file_loader

- `A = TypeVar('A', bound=BaseArtifact)` module-attribute

Bases: *BaseLoader[Union[str, PathLike* “) , ABC

Source Code in griptape/loaders/base_file_loader.py

```
@define
class BaseFileLoader(BaseLoader[Union[str, PathLike], bytes, A], ABC):
    file_manager_driver: BaseFileManagerDriver = field(
        default=Factory(lambda: LocalFileManagerDriver()),
        kw_only=True,
    )
```

(continues on next page)

(continued from previous page)

```
    encoding: str = field(default="utf-8", kw_only=True)

    def fetch(self, source: str | PathLike) -> bytes:
        # TODO: This is silly. `load_file` decodes the bytes and then we immediately␣
↪re-encode them.
        data = self.file_manager_driver.load_file(str(source)).value
        if isinstance(data, str):
            return data.encode(self.encoding)
        return data

    def save(self, destination: str | PathLike, artifact: A) -> None:
        """Saves the Artifact to a destination."""
        artifact.encoding = self.encoding
        self.file_manager_driver.save_file(str(destination), artifact.to_bytes())
```

- `encoding = field(default='utf-8', kw_only=True)` class-attribute instance-attribute
- `file_manager_driver = field(default=Factory(lambda: LocalFileManagerDriver()), kw_only=True)` class-attribute instance-attribute

fetch(source)

Source Code in griptape/loaders/base_file_loader.py

```
def fetch(self, source: str | PathLike) -> bytes:
    # TODO: This is silly. `load_file` decodes the bytes and then we immediately re-
↪encode them.
    data = self.file_manager_driver.load_file(str(source)).value
    if isinstance(data, str):
        return data.encode(self.encoding)
    return data
```

save(destination, artifact)

Source Code in griptape/loaders/base_file_loader.py

```
def save(self, destination: str | PathLike, artifact: A) -> None:
    """Saves the Artifact to a destination."""
    artifact.encoding = self.encoding
    self.file_manager_driver.save_file(str(destination), artifact.to_bytes())
```

base_loader

- `A = TypeVar('A', bound=BaseArtifact)` module-attribute
- `F = TypeVar('F')` module-attribute
- `S = TypeVar('S')` module-attribute

Bases: *FuturesExecutorMixin* “) , ABC, *Generic[S, F, A*

Attributes

Name	Type	Description
<a href="base_loader.mdx#griptape.loaders.base_loader.BaseLoaderArtifact.reference" title="reference = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.loaders.base_loader.BaseLoaderArtifact.reference)"><code class="language-python">	<a href="../common#griptape.common.Reference" title="Reference (griptape.common.Reference)"><code class="language-python">Optional[Reference]</code>	The optional *Reference* to set on the Artifact.

Source Code in griptape/loaders/base_loader.py

```
@define
class BaseLoader(FuturesExecutorMixin, ABC, Generic[S, F, A]):
    """Fetches data from a source, parses it, and returns an Artifact.

    Attributes:
        reference: The optional `Reference` to set on the Artifact.
    """

    reference: Optional[Reference] = field(default=None, kw_only=True)

    def load(self, source: S) -> A:
        data = self.fetch(source)

        return self.parse(data)

    @abstractmethod
    def fetch(self, source: S) -> F:
        """Fetches data from the source."""

    def parse(self, data: F) -> A:
        """Parses the fetched data and returns an Artifact."""
        artifact = self.try_parse(data)

        artifact.reference = self.reference

        return artifact

    def try_parse(self, data: F) -> A:
        """Parses the fetched data and returns an Artifact."""
        # TODO: Mark as abstract method for griptape 2.0
        raise NotImplementedError

    def load_collection(
```

(continues on next page)

(continued from previous page)

```
        self,
        sources: list[Any],
    ) -> Mapping[str, A]:
        """Loads a collection of sources and returns a dictionary of Artifacts."""
        # Create a dictionary before actually submitting the jobs to the executor
        # to avoid duplicate work.
        sources_by_key = {self.to_key(source): source for source in sources}

        with self.create_futures_executor() as futures_executor:
            return execute_futures_dict(
                {
                    key: futures_executor.submit(with_contextvars(self.load), source)
                    for key, source in sources_by_key.items()
                },
            )

    def to_key(self, source: S) -> str:
        """Converts the source to a key for the collection."""
        if isinstance(source, bytes):
            return bytes_to_hash(source)
        return str_to_hash(str(source))
```

- `reference = field(default=None, kw_only=True)` class-attribute instance-attribute

fetch(source)abstractmethod

Source Code in griptape/loaders/base_loader.py

```
@abstractmethod
def fetch(self, source: S) -> F:
    """Fetches data from the source."""
```

load(source)

Source Code in griptape/loaders/base_loader.py

```
def load(self, source: S) -> A:
    data = self.fetch(source)

    return self.parse(data)
```

load_collection(sources)

Source Code in griptape/loaders/base_loader.py

```
def load_collection(
    self,
    sources: list[Any],
) -> Mapping[str, A]:
    """Loads a collection of sources and returns a dictionary of Artifacts."""
    # Create a dictionary before actually submitting the jobs to the executor
```

(continues on next page)

(continued from previous page)

```
    # to avoid duplicate work.
    sources_by_key = {self.to_key(source): source for source in sources}

    with self.create_futures_executor() as futures_executor:
        return execute_futures_dict(
            {
                key: futures_executor.submit(with_contextvars(self.load), source)
                for key, source in sources_by_key.items()
            },
        )
```

parse(data)

Source Code in griptape/loaders/base_loader.py

```
def parse(self, data: F) -> A:
    """Parses the fetched data and returns an Artifact."""
    artifact = self.try_parse(data)

    artifact.reference = self.reference

    return artifact
```

to_key(source)

Source Code in griptape/loaders/base_loader.py

```
def to_key(self, source: S) -> str:
    """Converts the source to a key for the collection."""
    if isinstance(source, bytes):
        return bytes_to_hash(source)
    return str_to_hash(str(source))
```

try_parse(data)

Source Code in griptape/loaders/base_loader.py

```
def try_parse(self, data: F) -> A:
    """Parses the fetched data and returns an Artifact."""
    # TODO: Mark as abstract method for griptape 2.0
    raise NotImplementedError
```

blob_loader

Bases: *BaseFileLoader[BlobArtifact* “)

Source Code in griptape/loaders/blob_loader.py

```
@define
class BlobLoader(BaseFileLoader[BlobArtifact]):
    def try_parse(self, data: bytes) -> BlobArtifact:
```

(continues on next page)

(continued from previous page)

```
        if self.encoding is None:
            return BlobArtifact(data)
        return BlobArtifact(data, encoding=self.encoding)
```

try_parse(data)

Source Code in griptape/loaders/blob_loader.py

```
def try_parse(self, data: bytes) -> BlobArtifact:
    if self.encoding is None:
        return BlobArtifact(data)
    return BlobArtifact(data, encoding=self.encoding)
```

csv_loader

Bases: *BaseFileLoader[ListArtifact[TextArtifact* “)

Source Code in griptape/loaders/csv_loader.py

```
@define
class CsvLoader(BaseFileLoader[ListArtifact[TextArtifact]]):
    delimiter: str = field(default=",", kw_only=True)
    encoding: str = field(default="utf-8", kw_only=True)
    format_row: Callable[[dict], str] = field(
        default=lambda value: "\n".join(f"{key}: {val}" for key, val in value.
↪items()), kw_only=True
    )

    def try_parse(self, data: bytes) -> ListArtifact[TextArtifact]:
        reader = csv.DictReader(StringIO(data.decode(self.encoding)), delimiter=self.
↪delimiter)

        return ListArtifact(
            [TextArtifact(self.format_row(row), meta={"row_num": row_num}) for row_
↪num, row in enumerate(reader)]
        )
```

- `delimiter = field(default=',', kw_only=True)` class-attribute instance-attribute
- `encoding = field(default='utf-8', kw_only=True)` class-attribute instance-attribute
- `format_row = field(default=lambda value: '\n'.join(f'{key}: {val}' for (key, val) in value.items()), kw_only=True)` class-attribute instance-attribute

try_parse(data)

Source Code in griptape/loaders/csv_loader.py

```
def try_parse(self, data: bytes) -> ListArtifact[TextArtifact]:
    reader = csv.DictReader(StringIO(data.decode(self.encoding)), delimiter=self.
↪delimiter)

    return ListArtifact(
```

(continues on next page)

(continued from previous page)

```
        [TextArtifact(self.format_row(row), meta={"row_num": row_num}) for row_num,␣
↪row in enumerate(reader)]
    )
```

email_loader

Bases: *BaseLoader['EmailLoader.EmailQuery', list[bytes* “)

Source Code in griptape/loaders/email_loader.py

```
@define
class EmailLoader(BaseLoader["EmailLoader.EmailQuery", list[bytes], ListArtifact]):
↪# pyright: ignore[reportGeneralTypeIssues]
    @define(frozen=True)
    class EmailQuery:
        """An email retrieval query.

        Attributes:
            label: Label to retrieve emails from such as INBOX or SENT.
            key: Optional key for filtering such as FROM or SUBJECT.
            search_criteria: Optional search criteria to filter emails by key.
            max_count: Optional max email count.
        """

        label: str = field(kw_only=True)
        key: Optional[str] = field(default=None, kw_only=True)
        search_criteria: Optional[str] = field(default=None, kw_only=True)
        max_count: Optional[int] = field(default=None, kw_only=True)

    imap_url: str = field(kw_only=True)
    username: str = field(kw_only=True)
    password: str = field(kw_only=True)

    def fetch(self, source: EmailLoader.EmailQuery) -> list[bytes]:
        label, key, search_criteria, max_count = astuple(source)

        mail_bytes = []
        with imaplib.IMAP4_SSL(self.imap_url) as client:
            client.login(self.username, self.password)

            mailbox = client.select(f"{label}", readonly=True)
            if mailbox[0] != "OK":
                raise Exception(mailbox[1][0].decode())  # pyright:␣
↪ignore[reportOptionalMemberAccess] Unsure what mailbox[1][0] is, so leaving as-is

            if key and search_criteria:
                _, [message_numbers] = client.search(None, key, f"{search_criteria}")
                messages_count = self._count_messages(message_numbers)
            elif len(mailbox) > 1 and mailbox[1] and mailbox[1][0] is not None:
                messages_count = int(mailbox[1][0])
            else:
                raise Exception("unable to parse number of messages")
```

(continues on next page)

(continued from previous page)

```
            top_n = max(0, messages_count - max_count) if max_count else 0
            for i in range(messages_count, top_n, -1):
                _result, data = client.fetch(str(i), "(RFC822)")

                if data is None or not data or data[0] is None:
                    continue

                mail_bytes.append(data[0][1])

            client.close()

        return mail_bytes

    def try_parse(self, data: list[bytes]) -> ListArtifact[TextArtifact]:
        mailparser = import_optional_dependency("mailparser")
        artifacts = []
        for byte in data:
            message = mailparser.parse_from_bytes(byte)

            # Note: mailparser only populates the text_plain field
            # if the message content type is explicitly set to text/plain.
            if message.text_plain:
                artifacts.append(TextArtifact("\n".join(message.text_plain)))

        return ListArtifact(artifacts)

    def _count_messages(self, message_numbers: bytes) -> int:
        return len(list(filter(None, message_numbers.decode().split(" "))))
```

- `imap_url = field(kw_only=True)` class-attribute instance-attribute
- `password = field(kw_only=True)` class-attribute instance-attribute
- `username = field(kw_only=True)` class-attribute instance-attribute

EmailQuery

An email retrieval query.

Attributes

Name	Type	Description
<a href="email_loader.mdx#tape.loaders.email_loader.EmailLoader.EmailQuery.label" title="label = field(kw_only=True) class-attribute instance-attribute (griptape.loaders.email_loader.EmailLoader.EmailQuery.label)"><code class="language-python">	*str*	Label to retrieve emails from such as 'INBOX' or 'SENT'.
<a href="email_loader.mdx#tape.loaders.email_loader.EmailLoader.EmailQuery.key" title="key = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.loaders.email_loader.EmailLoader.EmailQuery.key)"><code class="language-python">	*Optional[str]*	Optional key for filtering such as 'FROM' or 'SUBJECT'.
<a href="email_loader.mdx#tape.loaders.email_loader.EmailLoader.EmailQuery.search_criteria" title="search_criteria = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.loaders.email_loader.EmailLoader.EmailQuery.search_criteria)"><code class="language-python">	*Optional[str]*	Optional search criteria to filter emails by key.

continues on next page

Table 326 – continued from previous page

Name	Type	Description
<a href="email_loader.mdx#tape.loaders.email_loader.EmailLoader.EmailQuery.max_count" title="max_count = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.loaders.email_loader.EmailLoader.EmailQuery.max_count)"><coc class="language-python">	*Optional[int]*	Optional max email count.

Source Code in griptape/loaders/email_loader.py

```
@define(frozen=True)
class EmailQuery:
    """An email retrieval query.

    Attributes:
        label: Label to retrieve emails from such as INBOX or SENT.
        key: Optional key for filtering such as FROM or SUBJECT.
        search_criteria: Optional search criteria to filter emails by key.
        max_count: Optional max email count.
    """

    label: str = field(kw_only=True)
    key: Optional[str] = field(default=None, kw_only=True)
    search_criteria: Optional[str] = field(default=None, kw_only=True)
    max_count: Optional[int] = field(default=None, kw_only=True)
```

- `key = field(default=None, kw_only=True)` class-attribute instance-attribute
- `label = field(kw_only=True)` class-attribute instance-attribute
- `max_count = field(default=None, kw_only=True)` class-attribute instance-attribute
- `search_criteria = field(default=None, kw_only=True)` class-attribute instance-attribute

_count_messages(message_numbers)

Source Code in griptape/loaders/email_loader.py

```
def _count_messages(self, message_numbers: bytes) -> int:
    return len(list(filter(None, message_numbers.decode().split(" "))))
```

fetch(source)

Source Code in griptape/loaders/email_loader.py

```
def fetch(self, source: EmailLoader.EmailQuery) -> list[bytes]:
    label, key, search_criteria, max_count = astuple(source)

    mail_bytes = []
    with imaplib.IMAP4_SSL(self.imap_url) as client:
        client.login(self.username, self.password)

        mailbox = client.select(f"{label}", readonly=True)
        if mailbox[0] != "OK":
            raise Exception(mailbox[1][0].decode())  # pyright:␣
↪ignore[reportOptionalMemberAccess] Unsure what mailbox[1][0] is, so leaving as-is

        if key and search_criteria:
            _, [message_numbers] = client.search(None, key, f"{search_criteria}")
            messages_count = self._count_messages(message_numbers)
        elif len(mailbox) > 1 and mailbox[1] and mailbox[1][0] is not None:
            messages_count = int(mailbox[1][0])
        else:
            raise Exception("unable to parse number of messages")

        top_n = max(0, messages_count - max_count) if max_count else 0
        for i in range(messages_count, top_n, -1):
            _result, data = client.fetch(str(i), "(RFC822)")

            if data is None or not data or data[0] is None:
                continue

            mail_bytes.append(data[0][1])

        client.close()

    return mail_bytes
```

try_parse(data)

Source Code in griptape/loaders/email_loader.py

```
def try_parse(self, data: list[bytes]) -> ListArtifact[TextArtifact]:
    mailparser = import_optional_dependency("mailparser")
    artifacts = []
    for byte in data:
        message = mailparser.parse_from_bytes(byte)

        # Note: mailparser only populates the text_plain field
        # if the message content type is explicitly set to text/plain.
        if message.text_plain:
            artifacts.append(TextArtifact("\n".join(message.text_plain)))

    return ListArtifact(artifacts)
```

image_loader

Bases: *BaseFileLoader[ImageArtifact ")*

Attributes

Name	Type	Description
<a href="image_loader.mdx#griptape.loaders.image_loader.ImageLoader.format" title="format = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.loaders.image_loader.ImageLoader.format)"><code class="language-python">	*Optional[str]*	If provided, attempts to ensure image artifacts are in this format when loaded. For example, when set to 'PNG', loading image.jpg will return an ImageArtifact containing the image bytes in PNG format.

Source Code in griptape/loaders/image_loader.py

```
@define
class ImageLoader(BaseFileLoader[ImageArtifact]):
    """Loads images into image artifacts.

    Attributes:
        format: If provided, attempts to ensure image artifacts are in this format␣
↪when loaded.
                For example, when set to PNG, loading image.jpg will return an␣
↪ImageArtifact containing the image
                    bytes in PNG format.
    """

    format: Optional[str] = field(default=None, kw_only=True)

    def try_parse(self, data: bytes) -> ImageArtifact:
        pil_image = import_optional_dependency("PIL.Image")
        image = pil_image.open(BytesIO(data))

        # Normalize format only if requested.
        if self.format is not None:
            byte_stream = BytesIO()
            image.save(byte_stream, format=self.format)
            image = pil_image.open(byte_stream)
            data = byte_stream.getvalue()

        return ImageArtifact(data, format=image.format.lower(), width=image.width,␣
↪height=image.height)
```

- `format = field(default=None, kw_only=True)` class-attribute instance-attribute

try_parse(data)

Source Code in griptape/loaders/image_loader.py

```
def try_parse(self, data: bytes) -> ImageArtifact:
    pil_image = import_optional_dependency("PIL.Image")
    image = pil_image.open(BytesIO(data))

    # Normalize format only if requested.
    if self.format is not None:
        byte_stream = BytesIO()
        image.save(byte_stream, format=self.format)
        image = pil_image.open(byte_stream)
        data = byte_stream.getvalue()

    return ImageArtifact(data, format=image.format.lower(), width=image.width,␣
↪height=image.height)
```

json_loader

Bases: *BaseFileLoader[JsonArtifact* “)

Source Code in griptape/loaders/json_loader.py

```
@define
class JsonLoader(BaseFileLoader[JsonArtifact]):
    def parse(self, data: bytes) -> JsonArtifact:
        return JsonArtifact(json.loads(data), encoding=self.encoding)
```

parse(data)

Source Code in griptape/loaders/json_loader.py

```
def parse(self, data: bytes) -> JsonArtifact:
    return JsonArtifact(json.loads(data), encoding=self.encoding)
```

pdf_loader

Bases: *BaseFileLoader* “)

Source Code in griptape/loaders/pdf_loader.py

```
@define
class PdfLoader(BaseFileLoader):
    def try_parse(
        self,
        data: bytes,
        *,
        password: Optional[str] = None,
    ) -> ListArtifact:
        pypdf = import_optional_dependency("pypdf")
```

(continues on next page)

(continued from previous page)

```
        reader = pypdf.PdfReader(BytesIO(data), strict=True, password=password)
        pages = [TextArtifact(p.extract_text()) for p in reader.pages]

        return ListArtifact(pages)
```

try_parse(data, *, password=None)

Source Code in griptape/loaders/pdf_loader.py

```
def try_parse(
    self,
    data: bytes,
    *,
    password: Optional[str] = None,
) -> ListArtifact:
    pypdf = import_optional_dependency("pypdf")
    reader = pypdf.PdfReader(BytesIO(data), strict=True, password=password)
    pages = [TextArtifact(p.extract_text()) for p in reader.pages]

    return ListArtifact(pages)
```

sql_loader

Bases: *BaseLoader[str, list[RowResult* “)

Source Code in griptape/loaders/sql_loader.py

```
@define
class SqlLoader(BaseLoader[str, list[BaseSqlDriver.RowResult],␣
↪ListArtifact[TextArtifact]]):
    sql_driver: BaseSqlDriver = field(kw_only=True)
    format_row: Callable[[dict], str] = field(
        default=lambda value: "\n".join(f"{key}: {val}" for key, val in value.
↪items()), kw_only=True
    )

    def fetch(self, source: str) -> list[BaseSqlDriver.RowResult]:
        return self.sql_driver.execute_query(source) or []

    def parse(self, data: list[BaseSqlDriver.RowResult]) ->␣
↪ListArtifact[TextArtifact]:
        return ListArtifact([TextArtifact(self.format_row(row.cells)) for row in␣
↪data])
```

- `format_row = field(default=lambda value: '\n'.join(f'{key}: {val}' for (key, val) in value.items()), kw_only=True)` class-attribute instance-attribute
- `sql_driver = field(kw_only=True)` class-attribute instance-attribute

fetch(source)

Source Code in griptape/loaders/sql_loader.py

```
def fetch(self, source: str) -> list[BaseSqlDriver.RowResult]:
    return self.sql_driver.execute_query(source) or []
```

parse(data)

Source Code in griptape/loaders/sql_loader.py

```
def parse(self, data: list[BaseSqlDriver.RowResult]) -> ListArtifact[TextArtifact]:
    return ListArtifact([TextArtifact(self.format_row(row.cells)) for row in data])
```

text_loader

Bases: *BaseFileLoader[TextArtifact* “)

Source Code in griptape/loaders/text_loader.py

```
@define
class TextLoader(BaseFileLoader[TextArtifact]):
    encoding: str = field(default="utf-8", kw_only=True)

    def try_parse(self, data: str | bytes) -> TextArtifact:
        if isinstance(data, str):
            return TextArtifact(data, encoding=self.encoding)
        return TextArtifact(data.decode(self.encoding), encoding=self.encoding)
```

- `encoding = field(default='utf-8', kw_only=True)` class-attribute instance-attribute

try_parse(data)

Source Code in griptape/loaders/text_loader.py

```
def try_parse(self, data: str | bytes) -> TextArtifact:
    if isinstance(data, str):
        return TextArtifact(data, encoding=self.encoding)
    return TextArtifact(data.decode(self.encoding), encoding=self.encoding)
```

web_loader

Bases: *BaseLoader[str, str, TextArtifact* “)

Source Code in griptape/loaders/web_loader.py

```
@define
class WebLoader(BaseLoader[str, str, TextArtifact]):
    web_scraper_driver: BaseWebScraperDriver = field(
        default=Factory(lambda: TrafilaturaWebScraperDriver()),
        kw_only=True,
    )

    def fetch(self, source: str) -> str:
```

(continues on next page)

(continued from previous page)

```
        return self.web_scraper_driver.fetch_url(source)

    def try_parse(self, data: str) -> TextArtifact:
        return self.web_scraper_driver.extract_page(data)
```

- `web_scraper_driver = field(default=Factory(lambda: TrafilaturaWebScraperDriver()), kw_only=True)` class-attribute instance-attribute

fetch(source)

Source Code in griptape/loaders/web_loader.py

```
def fetch(self, source: str) -> str:
    return self.web_scraper_driver.fetch_url(source)
```

try_parse(data)

Source Code in griptape/loaders/web_loader.py

```
def try_parse(self, data: str) -> TextArtifact:
    return self.web_scraper_driver.extract_page(data)
```

11.5.12 memory

- `__all__ = ['TaskMemory']` module-attribute

Bases: *ActivityMixin* “) , *SerializableMixin* ”)

Source Code in griptape/memory/task/task_memory.py

```
@define
class TaskMemory(ActivityMixin, SerializableMixin):
    name: str = field(
        default=Factory(lambda self: self.__class__.__name__, takes_self=True),
        kw_only=True,
        metadata={"serializable": True},
    )
    artifact_storages: dict[type, BaseArtifactStorage] = field(
        default=Factory(
            lambda: {
                TextArtifact: TextArtifactStorage(),
                BlobArtifact: BlobArtifactStorage(),
            }
        ),
        kw_only=True,
    )
    namespace_storage: dict[str, BaseArtifactStorage] = field(
        factory=dict, kw_only=True, metadata={"serializable": True}
    )
    namespace_metadata: dict[str, Any] = field(factory=dict, kw_only=True, metadata={
↪"serializable": True})

    @artifact_storages.validator  # pyright: ignore[reportAttributeAccessIssue]
```

(continues on next page)

(continued from previous page)

```
    def validate_artifact_storages(self, _: Attribute, artifact_storage: dict[type,␣
↪BaseArtifactStorage]) -> None:
        seen_types = []

        for storage in artifact_storage.values():
            if type(storage) in seen_types:
                raise ValueError("cant have more than memory storage of the same type
↪")

            seen_types.append(type(storage))

    def get_storage_for(self, artifact: BaseArtifact) ->␣
↪Optional[BaseArtifactStorage]:
        def find_storage(a: BaseArtifact) -> Optional[BaseArtifactStorage]:
            return next((v for k, v in self.artifact_storages.items() if isinstance(a,
↪ k)), None)

        if isinstance(artifact, ListArtifact):
            if artifact.has_items():
                return find_storage(artifact.value[0])
            return None
        return find_storage(artifact)

    def process_output(
        self,
        tool_activity: Callable,
        subtask: ActionsSubtask,
        output_artifact: BaseArtifact,
    ) -> BaseArtifact:
        from griptape.utils import J2

        tool_name = getattr(getattr(tool_activity, "__self__"), "name")
        activity_name = getattr(tool_activity, "name")
        namespace = output_artifact.name

        if output_artifact:
            result = self.store_artifact(namespace, output_artifact)

            if result:
                return result
            self.namespace_metadata[namespace] = subtask.actions_to_json()

            output = J2("memory/tool.j2").render(
                memory_name=self.name,
                tool_name=tool_name,
                activity_name=activity_name,
                artifact_namespace=namespace,
            )

            if subtask.structure and subtask.structure.meta_memory:
                subtask.structure.meta_memory.add_entry(
                    ActionSubtaskMetaEntry(
```

(continues on next page)

(continued from previous page)

```
                        thought=subtask.thought,
                        actions=subtask.actions_to_json(),
                        answer=output,
                    ),
                )

            return InfoArtifact(output, name=namespace)
        return InfoArtifact("tool output is empty")

    def store_artifact(self, namespace: str, artifact: BaseArtifact) ->␣
↪Optional[BaseArtifact]:
        namespace_storage = self.namespace_storage.get(namespace)
        storage = self.get_storage_for(artifact)

        if not storage:
            return artifact
        if namespace_storage and namespace_storage != storage:
            return ErrorArtifact("error storing tool output in memory")
        if storage:
            if isinstance(artifact, ListArtifact):
                for a in artifact.value:
                    storage.store_artifact(namespace, a)

                self.namespace_storage[namespace] = storage

                return None
            if isinstance(artifact, BaseArtifact):
                storage.store_artifact(namespace, artifact)

                self.namespace_storage[namespace] = storage

                return None
            return ErrorArtifact("error storing tool output in memory")
        return ErrorArtifact("error storing tool output in memory")

    def load_artifacts(self, namespace: str) -> ListArtifact:
        storage = self.namespace_storage.get(namespace)

        if storage:
            return storage.load_artifacts(namespace)
        return ListArtifact()

    def find_input_memory(self, memory_name: str) -> Optional[TaskMemory]:
        if memory_name == self.name:
            return self
        return None
```

- `artifact_storages = field(default=Factory(lambda: {TextArtifact: TextArtifactStorage(), BlobArtifact: BlobArtifactStorage()}), kw_only=True)` class-attribute instance-attribute
- `name = field(default=Factory(lambda self: self.__class__.__name__, takes_self=True), kw_only=True, metadata={'serializable': True})` class-attribute

instance-attribute

- `namespace_metadata = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `namespace_storage = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

find_input_memory(memory_name)

Source Code in griptape/memory/task/task_memory.py

```
def find_input_memory(self, memory_name: str) -> Optional[TaskMemory]:
    if memory_name == self.name:
        return self
    return None
```

get_storage_for(artifact)

Source Code in griptape/memory/task/task_memory.py

```
def get_storage_for(self, artifact: BaseArtifact) -> Optional[BaseArtifactStorage]:
    def find_storage(a: BaseArtifact) -> Optional[BaseArtifactStorage]:
        return next((v for k, v in self.artifact_storages.items() if isinstance(a,␣
↪k)), None)

    if isinstance(artifact, ListArtifact):
        if artifact.has_items():
            return find_storage(artifact.value[0])
        return None
    return find_storage(artifact)
```

load_artifacts(namespace)

Source Code in griptape/memory/task/task_memory.py

```
def load_artifacts(self, namespace: str) -> ListArtifact:
    storage = self.namespace_storage.get(namespace)

    if storage:
        return storage.load_artifacts(namespace)
    return ListArtifact()
```

process_output(tool_activity, subtask, output_artifact)

Source Code in griptape/memory/task/task_memory.py

```
def process_output(
    self,
    tool_activity: Callable,
    subtask: ActionsSubtask,
    output_artifact: BaseArtifact,
) -> BaseArtifact:
    from griptape.utils import J2
```

(continues on next page)

(continued from previous page)

```
    tool_name = getattr(getattr(tool_activity, "__self__"), "name")
    activity_name = getattr(tool_activity, "name")
    namespace = output_artifact.name

    if output_artifact:
        result = self.store_artifact(namespace, output_artifact)

        if result:
            return result
        self.namespace_metadata[namespace] = subtask.actions_to_json()

        output = J2("memory/tool.j2").render(
            memory_name=self.name,
            tool_name=tool_name,
            activity_name=activity_name,
            artifact_namespace=namespace,
        )

        if subtask.structure and subtask.structure.meta_memory:
            subtask.structure.meta_memory.add_entry(
                ActionSubtaskMetaEntry(
                    thought=subtask.thought,
                    actions=subtask.actions_to_json(),
                    answer=output,
                ),
            )

        return InfoArtifact(output, name=namespace)
    return InfoArtifact("tool output is empty")
```

store_artifact(namespace, artifact)

Source Code in griptape/memory/task/task_memory.py

```
def store_artifact(self, namespace: str, artifact: BaseArtifact) ->␣
↪Optional[BaseArtifact]:
    namespace_storage = self.namespace_storage.get(namespace)
    storage = self.get_storage_for(artifact)

    if not storage:
        return artifact
    if namespace_storage and namespace_storage != storage:
        return ErrorArtifact("error storing tool output in memory")
    if storage:
        if isinstance(artifact, ListArtifact):
            for a in artifact.value:
                storage.store_artifact(namespace, a)

            self.namespace_storage[namespace] = storage

            return None
```

(continues on next page)

(continued from previous page)

```
        if isinstance(artifact, BaseArtifact):
            storage.store_artifact(namespace, artifact)

            self.namespace_storage[namespace] = storage

            return None
        return ErrorArtifact("error storing tool output in memory")
    return ErrorArtifact("error storing tool output in memory")
```

validate*artifact_storages*(, artifact_storage)

Source Code in griptape/memory/task/task_memory.py

```
@artifact_storages.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_artifact_storages(self, _: Attribute, artifact_storage: dict[type,␣
↪BaseArtifactStorage]) -> None:
    seen_types = []

    for storage in artifact_storage.values():
        if type(storage) in seen_types:
            raise ValueError("cant have more than memory storage of the same type")

        seen_types.append(type(storage))
```

meta

- `__all__ = ['ActionSubtaskMetaEntry', 'BaseMetaEntry', 'MetaMemory']` module-attribute

Bases: *BaseMetaEntry* “)

Attributes

Name	Type	Description
<a href="./#griptape.memory.meta.ActionSubtaskMetaEntry.thought" title="thought = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.memory.meta.ActionSubtaskMetaEntry.thought)"><code class="language-python">	*Optional[str]*	CoT thought string from the LLM.

continues on next page

Table 328 – continued from previous page

Name	Type	Description
<a href="./#griptape.memory.meta.ActionSubtaskMetaEntry.actions" title="actions = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.memory.meta.ActionSubtaskMetaEntry.actions)"><code class="language-python">	*str*	ReAct actions JSON string from the LLM.
<a href="./#griptape.memory.meta.ActionSubtaskMetaEntry.answer" title="answer = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.memory.meta.ActionSubtaskMetaEntry.answer)"><code class="language-python">	*str*	tool-generated and memory-processed response from Gen AI Builder.

Source Code in griptape/memory/meta/action_subtask_meta_entry.py

```
@define
class ActionSubtaskMetaEntry(BaseMetaEntry):
    """Used to store ActionSubtask data to preserve TaskMemory pointers and context
↪in the form of thought and action.

    Attributes:
        thought: CoT thought string from the LLM.
        actions: ReAct actions JSON string from the LLM.
        answer: tool-generated and memory-processed response from Gen AI Builder.
    """

    type: str = field(default=BaseMetaEntry.__name__, kw_only=True, metadata={
↪"serializable": False})
    thought: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    actions: str = field(kw_only=True, metadata={"serializable": True})
    answer: str = field(kw_only=True, metadata={"serializable": True})
```

- `actions = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

- `answer = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `thought = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `type = field(default=BaseMetaEntry.__name__, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute

BaseMetaEntry

Bases: *SerializableMixin* “) , ABC

Source Code in griptape/memory/meta/base_meta_entry.py

```
@define
class BaseMetaEntry(SerializableMixin, ABC): ...
```

MetaMemory

Used to store meta entries that can be shared between tasks.

Attributes

Name	Type	Description
<a href="./#griptape.memory.meta.MetaMemory.entries" title="entries = field(factory=list, kw_only=True) class-attribute instance-attribute (griptape.memory.meta.MetaMemory.entries)"><code class="language-python">	<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">list[BaseMetaEntry]</code>	a list of meta entries for downstream tasks and subtasks to load.

Source Code in griptape/memory/meta/meta_memory.py

```
@define
class MetaMemory:
    """Used to store meta entries that can be shared between tasks.

    Attributes:
        entries: a list of meta entries for downstream tasks and subtasks to load.
    """

    entries: list[BaseMetaEntry] = field(factory=list, kw_only=True)

    def add_entry(self, entry: BaseMetaEntry) -> None:
        self.entries.append(entry)
```

- `entries = field(factory=list, kw_only=True)` class-attribute instance-attribute

add_entry(entry)

Source Code in griptape/memory/meta/meta_memory.py

```
def add_entry(self, entry: BaseMetaEntry) -> None:
    self.entries.append(entry)
```

action_subtask_meta_entry

Bases: *BaseMetaEntry* “)

Attributes

Name	Type	Description
<a href="action_subtask_meta_entry.mdx#griptape.memory.meta.action_subtask_meta_entry.ActionSubtaskMetaEntry.thought" title="thought = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.memory.meta.action_subtask_meta_entry.ActionSubtaskMetaEntry.thought)"><code class="language-python">	*Optional[str]*	CoT thought string from the LLM.

continues on next page

Table 330 – continued from previous page

Name	Type	Description
<a href="action_subtask_meta_entry.mdx#griptape.memory.meta.action_subtask_meta_entry.ActionSubtaskMetaEntry.actions" title="actions = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.memory.meta.action_subtask_meta_entry.ActionSubtaskMetaEntry.actions)"><code class="language-python">	*str*	ReAct actions JSON string from the LLM.
<a href="action_subtask_meta_entry.mdx#griptape.memory.meta.action_subtask_meta_entry.ActionSubtaskMetaEntry.answer" title="answer = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.memory.meta.action_subtask_meta_entry.ActionSubtaskMetaEntry.answer)"><code class="language-python">	*str*	tool-generated and memory-processed response from Gen AI Builder.

Source Code in griptape/memory/meta/action_subtask_meta_entry.py

```
@define
class ActionSubtaskMetaEntry(BaseMetaEntry):
    """Used to store ActionSubtask data to preserve TaskMemory pointers and context␣
↪in the form of thought and action.

    Attributes:
        thought: CoT thought string from the LLM.
        actions: ReAct actions JSON string from the LLM.
        answer: tool-generated and memory-processed response from Gen AI Builder.
```

(continues on next page)

(continued from previous page)

```
    """

    type: str = field(default=BaseMetaEntry.__name__, kw_only=True, metadata={
↪"serializable": False})
    thought: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    actions: str = field(kw_only=True, metadata={"serializable": True})
    answer: str = field(kw_only=True, metadata={"serializable": True})
```

- `actions = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `answer = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `thought = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `type = field(default=BaseMetaEntry.__name__, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute

base_meta_entry

Bases: *SerializableMixin* “) , ABC

Source Code in griptape/memory/meta/base_meta_entry.py

```
@define
class BaseMetaEntry(SerializableMixin, ABC): ...
```

meta_memory

Used to store meta entries that can be shared between tasks.

Attributes

Name	Type	Description
<a href="meta_memory.mdx#griptape.memory.meta.meta_memory.MetaMemory.entries" title="entries = field(factory=list, kw_only=True) class-attribute instance-attribute (griptape.memory.meta.meta_memory.MetaMemory.entries)"><code class="language-python">	<a href="../../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">list[BaseMetaEntry]</code>	a list of meta entries for downstream tasks and subtasks to load.

Source Code in griptape/memory/meta/meta_memory.py

```
@define
class MetaMemory:
    """Used to store meta entries that can be shared between tasks.

    Attributes:
        entries: a list of meta entries for downstream tasks and subtasks to load.
    """

    entries: list[BaseMetaEntry] = field(factory=list, kw_only=True)

    def add_entry(self, entry: BaseMetaEntry) -> None:
        self.entries.append(entry)
```

- `entries = field(factory=list, kw_only=True)` class-attribute instance-attribute

add_entry(entry)

Source Code in griptape/memory/meta/meta_memory.py

```
def add_entry(self, entry: BaseMetaEntry) -> None:
    self.entries.append(entry)
```

structure

- `__all__ = ['BaseConversationMemory', 'ConversationMemory', 'Run', 'SummaryConversationMemory']` module-attribute

Bases: *SerializableMixin* “) , ABC

Source Code in griptape/memory/structure/base_conversation_memory.py

```
@define
class BaseConversationMemory(SerializableMixin, ABC):
    conversation_memory_driver: BaseConversationMemoryDriver = field(
```

(continues on next page)

(continued from previous page)

```
        default=Factory(lambda: Defaults.drivers_config.conversation_memory_driver),␣
↪kw_only=True
    )
    runs: list[Run] = field(factory=list, kw_only=True, metadata={"serializable":␣
↪True})
    meta: dict[str, Any] = field(factory=dict, kw_only=True, metadata={"serializable
↪": True})
    autoload: bool = field(default=True, kw_only=True)
    autoprune: bool = field(default=True, kw_only=True)
    max_runs: Optional[int] = field(default=None, kw_only=True, metadata={
↪"serializable": True})

    def __attrs_post_init__(self) -> None:
        if self.autoload:
            self.load_runs()

    def before_add_run(self) -> None:
        pass

    def add_run(self, run: Run) -> BaseConversationMemory:
        self.before_add_run()
        self.try_add_run(run)
        self.after_add_run()

        return self

    def after_add_run(self) -> None:
        if self.max_runs:
            while len(self.runs) > self.max_runs:
                self.runs.pop(0)
        self.conversation_memory_driver.store(self.runs, self.meta)

    @abstractmethod
    def try_add_run(self, run: Run) -> None: ...

    @abstractmethod
    def to_prompt_stack(self, last_n: Optional[int] = None) -> PromptStack: ...

    def load_runs(self) -> list[Run]:
        runs, meta = self.conversation_memory_driver.load()
        self.runs.extend(runs)
        self.meta = dict_merge(self.meta, meta)

        return self.runs

    def add_to_prompt_stack(
        self, prompt_driver: BasePromptDriver, prompt_stack: PromptStack, index:␣
↪Optional[int] = None
    ) -> PromptStack:
        """Add the Conversation Memory runs to the Prompt Stack by modifying the␣
↪messages in place.
```

(continues on next page)

(continued from previous page)

```
        If autoprune is enabled, this will fit as many Conversation Memory runs into␣
↪the Prompt Stack
        as possible without exceeding the token limit.

        Args:
            prompt_driver: The Prompt Driver to use for token counting.
            prompt_stack: The Prompt Stack to add the Conversation Memory to.
            index: Optional index to insert the Conversation Memory runs at.
                   Defaults to appending to the end of the Prompt Stack.
        """
        num_runs_to_fit_in_prompt = len(self.runs)

        if self.autoprune:
            should_prune = True
            temp_stack = PromptStack()

            # Try to determine how many Conversation Memory runs we can
            # fit into the Prompt Stack without exceeding the token limit.
            while should_prune and num_runs_to_fit_in_prompt > 0:
                temp_stack.messages = prompt_stack.messages.copy()

                # Add n runs from Conversation Memory.
                # Where we insert into the Prompt Stack doesnt matter here
                # since we only care about the total token count.
                memory_inputs = self.to_prompt_stack(num_runs_to_fit_in_prompt).
↪messages
                temp_stack.messages.extend(memory_inputs)

                # Convert the Prompt Stack into tokens left.
                tokens_left = prompt_driver.tokenizer.count_input_tokens_left(
                    prompt_driver.prompt_stack_to_string(temp_stack),
                )
                if tokens_left > 0:
                    # There are still tokens left, no need to prune.
                    should_prune = False
                else:
                    # There were not any tokens left, prune one run and try again.
                    num_runs_to_fit_in_prompt -= 1

        if num_runs_to_fit_in_prompt:
            memory_inputs = self.to_prompt_stack(num_runs_to_fit_in_prompt).messages
            if index is None:
                prompt_stack.messages.extend(memory_inputs)
            else:
                prompt_stack.messages[index:index] = memory_inputs

        return prompt_stack
```

- `autoload = field(default=True, kw_only=True)` class-attribute instance-attribute
- `autoprune = field(default=True, kw_only=True)` class-attribute instance-attribute
- `conversation_memory_driver = field(default=Factory(lambda: Defaults.drivers_config.conversation_memory_driver), kw_only=True)` class-attribute instance-attribute

- `max_runs = field(default=None, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute
- `meta = field(factory=dict, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute
- `runs = field(factory=list, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/memory/structure/base_conversation_memory.py

```
def __attrs_post_init__(self) -> None:
    if self.autoload:
        self.load_runs()
```

add_run(run)

Source Code in griptape/memory/structure/base_conversation_memory.py

```
def add_run(self, run: Run) -> BaseConversationMemory:
    self.before_add_run()
    self.try_add_run(run)
    self.after_add_run()

    return self
```

add_to_prompt_stack(prompt_driver, prompt_stack, index=None)

Add the Conversation Memory runs to the Prompt Stack by modifying the messages in place.

If autoprune is enabled, this will fit as many Conversation Memory runs into the Prompt Stack as possible without exceeding the token limit.

Parameters

Name	Type	Description	Default
prompt_driver	<a href="../../drivers/prontape.drivers.prompt.BaPromptDriver" title="BasePrompt-Driver (griptape.drivers.prompt.BaPrompt-Driver)"><code class="language-pythc	The Prompt Driver to use for token counting. 	*required*

continues on next page

Table 332 – continued from previous page

Name	Type	Description	Default
prompt_stack	<a href="../../common#griptape.common.PromptStack" title="PromptStack (griptape.common.PromptStack)"><code class="language-pythc	The Prompt Stack to add the Conversation Memory to. 	*required*
index	*Optional[int]*	Optional index to insert the Conversation Memory runs at. Defaults to appending to the end of the Prompt Stack. 	*None*

Source Code in griptape/memory/structure/base_conversation_memory.py

```
def add_to_prompt_stack(
    self, prompt_driver: BasePromptDriver, prompt_stack: PromptStack, index:␣
↪Optional[int] = None
) -> PromptStack:
    """Add the Conversation Memory runs to the Prompt Stack by modifying the messages␣
↪in place.

    If autoprune is enabled, this will fit as many Conversation Memory runs into the␣
↪Prompt Stack
    as possible without exceeding the token limit.

    Args:
        prompt_driver: The Prompt Driver to use for token counting.
        prompt_stack: The Prompt Stack to add the Conversation Memory to.
        index: Optional index to insert the Conversation Memory runs at.
               Defaults to appending to the end of the Prompt Stack.
    """
    num_runs_to_fit_in_prompt = len(self.runs)

    if self.autoprune:
        should_prune = True
        temp_stack = PromptStack()

        # Try to determine how many Conversation Memory runs we can
        # fit into the Prompt Stack without exceeding the token limit.
        while should_prune and num_runs_to_fit_in_prompt > 0:
            temp_stack.messages = prompt_stack.messages.copy()

            # Add n runs from Conversation Memory.
            # Where we insert into the Prompt Stack doesnt matter here
            # since we only care about the total token count.
            memory_inputs = self.to_prompt_stack(num_runs_to_fit_in_prompt).messages
            temp_stack.messages.extend(memory_inputs)
```

(continues on next page)

(continued from previous page)

```
            # Convert the Prompt Stack into tokens left.
            tokens_left = prompt_driver.tokenizer.count_input_tokens_left(
                prompt_driver.prompt_stack_to_string(temp_stack),
            )
            if tokens_left > 0:
                # There are still tokens left, no need to prune.
                should_prune = False
            else:
                # There were not any tokens left, prune one run and try again.
                num_runs_to_fit_in_prompt -= 1

    if num_runs_to_fit_in_prompt:
        memory_inputs = self.to_prompt_stack(num_runs_to_fit_in_prompt).messages
        if index is None:
            prompt_stack.messages.extend(memory_inputs)
        else:
            prompt_stack.messages[index:index] = memory_inputs

    return prompt_stack
```

after_add_run()

Source Code in griptape/memory/structure/base_conversation_memory.py

```
def after_add_run(self) -> None:
    if self.max_runs:
        while len(self.runs) > self.max_runs:
            self.runs.pop(0)
    self.conversation_memory_driver.store(self.runs, self.meta)
```

before_add_run()

Source Code in griptape/memory/structure/base_conversation_memory.py

```
def before_add_run(self) -> None:
    pass
```

load_runs()

Source Code in griptape/memory/structure/base_conversation_memory.py

```
def load_runs(self) -> list[Run]:
    runs, meta = self.conversation_memory_driver.load()
    self.runs.extend(runs)
    self.meta = dict_merge(self.meta, meta)

    return self.runs
```

to_prompt_stack(last_n=None)abstractmethod

Source Code in griptape/memory/structure/base_conversation_memory.py

```
@abstractmethod
def to_prompt_stack(self, last_n: Optional[int] = None) -> PromptStack: ...
```

try_add_run(run)abstractmethod

Source Code in griptape/memory/structure/base_conversation_memory.py

```
@abstractmethod
def try_add_run(self, run: Run) -> None: ...
```

ConversationMemory

Bases: *BaseConversationMemoryDriver* “)

Source Code in griptape/memory/structure/conversation_memory.py

```
@define
class ConversationMemory(BaseConversationMemory):
    def try_add_run(self, run: Run) -> None:
        self.runs.append(run)

    def to_prompt_stack(self, last_n: Optional[int] = None) -> PromptStack:
        prompt_stack = PromptStack()
        runs = self.runs[-last_n:] if last_n else self.runs
        for run in runs:
            prompt_stack.add_user_message(run.input)
            prompt_stack.add_assistant_message(run.output)
        return prompt_stack
```

to_prompt_stack(last_n=None)

Source Code in griptape/memory/structure/conversation_memory.py

```
def to_prompt_stack(self, last_n: Optional[int] = None) -> PromptStack:
    prompt_stack = PromptStack()
    runs = self.runs[-last_n:] if last_n else self.runs
    for run in runs:
        prompt_stack.add_user_message(run.input)
        prompt_stack.add_assistant_message(run.output)
    return prompt_stack
```

try_add_run(run)

Source Code in griptape/memory/structure/conversation_memory.py

```
def try_add_run(self, run: Run) -> None:
    self.runs.append(run)
```

Run

Bases: *SerializableMixin* “)

Source Code in griptape/memory/structure/run.py

```
@define(kw_only=True)
class Run(SerializableMixin):
    id: str = field(default=Factory(lambda: uuid.uuid4().hex), metadata={"serializable
↪": True})
    meta: Optional[dict] = field(default=None, metadata={"serializable": True})
    input: BaseArtifact = field(metadata={"serializable": True})
    output: BaseArtifact = field(metadata={"serializable": True})
```

- `id = field(default=Factory(lambda: uuid.uuid4().hex), metadata={'serializable': True})` class-attribute instance-attribute
- `input = field(metadata={'serializable': True})` class-attribute instance-attribute
- `meta = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `output = field(metadata={'serializable': True})` class-attribute instance-attribute

SummaryConversationMemory

Bases: *BaseConversationMemoryDriver* “)

Source Code in griptape/memory/structure/summary_conversation_memory.py

```
@define
class SummaryConversationMemory(BaseConversationMemory):
    offset: int = field(default=1, kw_only=True, metadata={"serializable": True})
    prompt_driver: BasePromptDriver = field(
        kw_only=True, default=Factory(lambda: Defaults.drivers_config.prompt_driver)
    )
    summary: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    summary_index: int = field(default=0, kw_only=True, metadata={"serializable":␣
↪True})
    summary_get_template: J2 = field(default=Factory(lambda: J2("memory/conversation/
↪summary.j2")), kw_only=True)
    summarize_conversation_get_template: J2 = field(
        default=Factory(lambda: J2("memory/conversation/summarize_conversation.j2")),
        kw_only=True,
    )

    def to_prompt_stack(self, last_n: Optional[int] = None) -> PromptStack:
        stack = PromptStack()
        if self.summary:
            stack.add_user_message(self.summary_get_template.render(summary=self.
↪summary))

        for r in self.unsummarized_runs(last_n):
            stack.add_user_message(r.input)
            stack.add_assistant_message(r.output)
```

(continues on next page)

(continued from previous page)

```
        return stack

    def unsummarized_runs(self, last_n: Optional[int] = None) -> list[Run]:
        summary_index_runs = self.runs[self.summary_index :]

        if last_n:
            last_n_runs = self.runs[-last_n:]

            if len(summary_index_runs) > len(last_n_runs):
                return last_n_runs
            return summary_index_runs
        return summary_index_runs

    def try_add_run(self, run: Run) -> None:
        self.runs.append(run)
        unsummarized_runs = self.unsummarized_runs()
        runs_to_summarize = unsummarized_runs[: max(0, len(unsummarized_runs) - self.
↪offset)]

        if len(runs_to_summarize) > 0:
            self.summary = self.summarize_runs(self.summary, runs_to_summarize)
            self.summary_index = 1 + self.runs.index(runs_to_summarize[-1])

    def summarize_runs(self, previous_summary: str | None, runs: list[Run]) -> str |␣
↪None:
        try:
            if len(runs) > 0:
                summary = self.summarize_conversation_get_template.
↪render(summary=previous_summary, runs=runs)
                return self.prompt_driver.run(
                    PromptStack(messages=[Message(summary, role=Message.USER_ROLE)]),
                ).to_text()
            return previous_summary
        except Exception as e:
            logging.exception("Error summarizing memory: %s(%s)", type(e).__name__, e)

            return previous_summary
```

- `offset = field(default=1, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `prompt_driver = field(kw_only=True, default=Factory(lambda: Defaults.drivers_config.prompt_driver))` class-attribute instance-attribute
- `summarize_conversation_get_template = field(default=Factory(lambda: J2('memory/conversation/summarize_conversation.j2')), kw_only=True)` class-attribute instance-attribute
- `summary = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `summary_get_template = field(default=Factory(lambda: J2('memory/conversation/summary.j2')), kw_only=True)` class-attribute instance-attribute
- `summary_index = field(default=0, kw_only=True, metadata={'serializable': True})`

class-attribute instance-attribute

summarize_runs(previous_summary, runs)

Source Code in griptape/memory/structure/summary_conversation_memory.py

```
def summarize_runs(self, previous_summary: str | None, runs: list[Run]) -> str | None:
    try:
        if len(runs) > 0:
            summary = self.summarize_conversation_get_template.
↪render(summary=previous_summary, runs=runs)
            return self.prompt_driver.run(
                PromptStack(messages=[Message(summary, role=Message.USER_ROLE)]),
            ).to_text()
        return previous_summary
    except Exception as e:
        logging.exception("Error summarizing memory: %s(%s)", type(e).__name__, e)

        return previous_summary
```

to_prompt_stack(last_n=None)

Source Code in griptape/memory/structure/summary_conversation_memory.py

```
def to_prompt_stack(self, last_n: Optional[int] = None) -> PromptStack:
    stack = PromptStack()
    if self.summary:
        stack.add_user_message(self.summary_get_template.render(summary=self.summary))

    for r in self.unsummarized_runs(last_n):
        stack.add_user_message(r.input)
        stack.add_assistant_message(r.output)

    return stack
```

try_add_run(run)

Source Code in griptape/memory/structure/summary_conversation_memory.py

```
def try_add_run(self, run: Run) -> None:
    self.runs.append(run)
    unsummarized_runs = self.unsummarized_runs()
    runs_to_summarize = unsummarized_runs[: max(0, len(unsummarized_runs) - self.
↪offset)]

    if len(runs_to_summarize) > 0:
        self.summary = self.summarize_runs(self.summary, runs_to_summarize)
        self.summary_index = 1 + self.runs.index(runs_to_summarize[-1])
```

unsummarized_runs(last_n=None)

Source Code in griptape/memory/structure/summary_conversation_memory.py

```
def unsummarized_runs(self, last_n: Optional[int] = None) -> list[Run]:
    summary_index_runs = self.runs[self.summary_index :]

    if last_n:
        last_n_runs = self.runs[-last_n:]

        if len(summary_index_runs) > len(last_n_runs):
            return last_n_runs
        return summary_index_runs
    return summary_index_runs
```

base_conversation_memory

Bases: *SerializableMixin* “) , ABC

Source Code in griptape/memory/structure/base_conversation_memory.py

```
@define
class BaseConversationMemory(SerializableMixin, ABC):
    conversation_memory_driver: BaseConversationMemoryDriver = field(
        default=Factory(lambda: Defaults.drivers_config.conversation_memory_driver),␣
↪kw_only=True
    )
    runs: list[Run] = field(factory=list, kw_only=True, metadata={"serializable":␣
↪True})
    meta: dict[str, Any] = field(factory=dict, kw_only=True, metadata={"serializable
↪": True})
    autoload: bool = field(default=True, kw_only=True)
    autoprune: bool = field(default=True, kw_only=True)
    max_runs: Optional[int] = field(default=None, kw_only=True, metadata={
↪"serializable": True})

    def __attrs_post_init__(self) -> None:
        if self.autoload:
            self.load_runs()

    def before_add_run(self) -> None:
        pass

    def add_run(self, run: Run) -> BaseConversationMemory:
        self.before_add_run()
        self.try_add_run(run)
        self.after_add_run()

        return self

    def after_add_run(self) -> None:
        if self.max_runs:
            while len(self.runs) > self.max_runs:
                self.runs.pop(0)
```

(continues on next page)

(continued from previous page)

```
        self.conversation_memory_driver.store(self.runs, self.meta)

    @abstractmethod
    def try_add_run(self, run: Run) -> None: ...

    @abstractmethod
    def to_prompt_stack(self, last_n: Optional[int] = None) -> PromptStack: ...

    def load_runs(self) -> list[Run]:
        runs, meta = self.conversation_memory_driver.load()
        self.runs.extend(runs)
        self.meta = dict_merge(self.meta, meta)

        return self.runs

    def add_to_prompt_stack(
        self, prompt_driver: BasePromptDriver, prompt_stack: PromptStack, index:␣
↪Optional[int] = None
    ) -> PromptStack:
        """Add the Conversation Memory runs to the Prompt Stack by modifying the␣
↪messages in place.

        If autoprune is enabled, this will fit as many Conversation Memory runs into␣
↪the Prompt Stack
        as possible without exceeding the token limit.

        Args:
            prompt_driver: The Prompt Driver to use for token counting.
            prompt_stack: The Prompt Stack to add the Conversation Memory to.
            index: Optional index to insert the Conversation Memory runs at.
                   Defaults to appending to the end of the Prompt Stack.
        """
        num_runs_to_fit_in_prompt = len(self.runs)

        if self.autoprune:
            should_prune = True
            temp_stack = PromptStack()

            # Try to determine how many Conversation Memory runs we can
            # fit into the Prompt Stack without exceeding the token limit.
            while should_prune and num_runs_to_fit_in_prompt > 0:
                temp_stack.messages = prompt_stack.messages.copy()

                # Add n runs from Conversation Memory.
                # Where we insert into the Prompt Stack doesnt matter here
                # since we only care about the total token count.
                memory_inputs = self.to_prompt_stack(num_runs_to_fit_in_prompt).
↪messages
                temp_stack.messages.extend(memory_inputs)

                # Convert the Prompt Stack into tokens left.
                tokens_left = prompt_driver.tokenizer.count_input_tokens_left(
```

(continues on next page)

(continued from previous page)

```
                    prompt_driver.prompt_stack_to_string(temp_stack),
                )
                if tokens_left > 0:
                    # There are still tokens left, no need to prune.
                    should_prune = False
                else:
                    # There were not any tokens left, prune one run and try again.
                    num_runs_to_fit_in_prompt -= 1

        if num_runs_to_fit_in_prompt:
            memory_inputs = self.to_prompt_stack(num_runs_to_fit_in_prompt).messages
            if index is None:
                prompt_stack.messages.extend(memory_inputs)
            else:
                prompt_stack.messages[index:index] = memory_inputs

        return prompt_stack
```

- `autoload = field(default=True, kw_only=True)` class-attribute instance-attribute
- `autoprune = field(default=True, kw_only=True)` class-attribute instance-attribute
- `conversation_memory_driver = field(default=Factory(lambda: Defaults.drivers_config.conversation_memory_driver), kw_only=True)` class-attribute instance-attribute
- `max_runs = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `meta = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `runs = field(factory=list, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/memory/structure/base_conversation_memory.py

```
def __attrs_post_init__(self) -> None:
    if self.autoload:
        self.load_runs()
```

add_run(run)

Source Code in griptape/memory/structure/base_conversation_memory.py

```
def add_run(self, run: Run) -> BaseConversationMemory:
    self.before_add_run()
    self.try_add_run(run)
    self.after_add_run()

    return self
```

add_to_prompt_stack(prompt_driver, prompt_stack, index=None)

Add the Conversation Memory runs to the Prompt Stack by modifying the messages in place.

If autoprune is enabled, this will fit as many Conversation Memory runs into the Prompt Stack as possible without exceeding the token limit.

Parameters

Name	Type	Description	Default
prompt_driver	<a href="../../drivers/prontape.drivers.prompt.BaPromptDriver" title="BasePrompt-Driver (griptape.drivers.prompt.BaPrompt-Driver)"><code class="language-pythc	The Prompt Driver to use for token counting. 	*required*
prompt_stack	<a href="../../common#griptape.common.PromptStack" title="PromptStack (griptape.common.Prompt-Stack)"><code class="language-pythc	The Prompt Stack to add the Conversation Memory to. 	*required*
index	*Optional[int]*	Optional index to insert the Conversation Memory runs at. Defaults to appending to the end of the Prompt Stack. 	*None*

Source Code in griptape/memory/structure/base_conversation_memory.py

```
def add_to_prompt_stack(
    self, prompt_driver: BasePromptDriver, prompt_stack: PromptStack, index:␣
↪Optional[int] = None
) -> PromptStack:
    """Add the Conversation Memory runs to the Prompt Stack by modifying the messages␣
↪in place.

    If autoprune is enabled, this will fit as many Conversation Memory runs into the␣
↪Prompt Stack
    as possible without exceeding the token limit.

    Args:
        prompt_driver: The Prompt Driver to use for token counting.
        prompt_stack: The Prompt Stack to add the Conversation Memory to.
        index: Optional index to insert the Conversation Memory runs at.
```

(continues on next page)

(continued from previous page)

```
            Defaults to appending to the end of the Prompt Stack.
    """
    num_runs_to_fit_in_prompt = len(self.runs)

    if self.autoprune:
        should_prune = True
        temp_stack = PromptStack()

        # Try to determine how many Conversation Memory runs we can
        # fit into the Prompt Stack without exceeding the token limit.
        while should_prune and num_runs_to_fit_in_prompt > 0:
            temp_stack.messages = prompt_stack.messages.copy()

            # Add n runs from Conversation Memory.
            # Where we insert into the Prompt Stack doesnt matter here
            # since we only care about the total token count.
            memory_inputs = self.to_prompt_stack(num_runs_to_fit_in_prompt).messages
            temp_stack.messages.extend(memory_inputs)

            # Convert the Prompt Stack into tokens left.
            tokens_left = prompt_driver.tokenizer.count_input_tokens_left(
                prompt_driver.prompt_stack_to_string(temp_stack),
            )
            if tokens_left > 0:
                # There are still tokens left, no need to prune.
                should_prune = False
            else:
                # There were not any tokens left, prune one run and try again.
                num_runs_to_fit_in_prompt -= 1

    if num_runs_to_fit_in_prompt:
        memory_inputs = self.to_prompt_stack(num_runs_to_fit_in_prompt).messages
        if index is None:
            prompt_stack.messages.extend(memory_inputs)
        else:
            prompt_stack.messages[index:index] = memory_inputs

    return prompt_stack
```

after_add_run()

Source Code in griptape/memory/structure/base_conversation_memory.py

```
def after_add_run(self) -> None:
    if self.max_runs:
        while len(self.runs) > self.max_runs:
            self.runs.pop(0)
    self.conversation_memory_driver.store(self.runs, self.meta)
```

before_add_run()

Source Code in griptape/memory/structure/base_conversation_memory.py

```
def before_add_run(self) -> None:
    pass
```

load_runs()

Source Code in griptape/memory/structure/base_conversation_memory.py

```
def load_runs(self) -> list[Run]:
    runs, meta = self.conversation_memory_driver.load()
    self.runs.extend(runs)
    self.meta = dict_merge(self.meta, meta)

    return self.runs
```

to_prompt_stack(last_n=None)abstractmethod

Source Code in griptape/memory/structure/base_conversation_memory.py

```
@abstractmethod
def to_prompt_stack(self, last_n: Optional[int] = None) -> PromptStack: ...
```

try_add_run(run)abstractmethod

Source Code in griptape/memory/structure/base_conversation_memory.py

```
@abstractmethod
def try_add_run(self, run: Run) -> None: ...
```

conversation_memory

Bases: *BaseConversationMemoryDriver* “)

Source Code in griptape/memory/structure/conversation_memory.py

```
@define
class ConversationMemory(BaseConversationMemory):
    def try_add_run(self, run: Run) -> None:
        self.runs.append(run)

    def to_prompt_stack(self, last_n: Optional[int] = None) -> PromptStack:
        prompt_stack = PromptStack()
        runs = self.runs[-last_n:] if last_n else self.runs
        for run in runs:
            prompt_stack.add_user_message(run.input)
            prompt_stack.add_assistant_message(run.output)
        return prompt_stack
```

to_prompt_stack(last_n=None)

Source Code in griptape/memory/structure/conversation_memory.py

```
def to_prompt_stack(self, last_n: Optional[int] = None) -> PromptStack:
    prompt_stack = PromptStack()
    runs = self.runs[-last_n:] if last_n else self.runs
    for run in runs:
        prompt_stack.add_user_message(run.input)
        prompt_stack.add_assistant_message(run.output)
    return prompt_stack
```

try_add_run(run)

Source Code in griptape/memory/structure/conversation_memory.py

```
def try_add_run(self, run: Run) -> None:
    self.runs.append(run)
```

run

Bases: *SerializableMixin* “)

Source Code in griptape/memory/structure/run.py

```
@define(kw_only=True)
class Run(SerializableMixin):
    id: str = field(default=Factory(lambda: uuid.uuid4().hex), metadata={"serializable
↪": True})
    meta: Optional[dict] = field(default=None, metadata={"serializable": True})
    input: BaseArtifact = field(metadata={"serializable": True})
    output: BaseArtifact = field(metadata={"serializable": True})
```

- `id = field(default=Factory(lambda: uuid.uuid4().hex), metadata={'serializable': True})` class-attribute instance-attribute
- `input = field(metadata={'serializable': True})` class-attribute instance-attribute
- `meta = field(default=None, metadata={'serializable': True})` class-attribute instance-attribute
- `output = field(metadata={'serializable': True})` class-attribute instance-attribute

summary_conversation_memory

Bases: *BaseConversationMemoryDriver* “)

Source Code in griptape/memory/structure/summary_conversation_memory.py

```
@define
class SummaryConversationMemory(BaseConversationMemory):
    offset: int = field(default=1, kw_only=True, metadata={"serializable": True})
    prompt_driver: BasePromptDriver = field(
        kw_only=True, default=Factory(lambda: Defaults.drivers_config.prompt_driver)
    )
```

(continues on next page)

(continued from previous page)

```
    summary: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": True})
    summary_index: int = field(default=0, kw_only=True, metadata={"serializable":␣
↪True})
    summary_get_template: J2 = field(default=Factory(lambda: J2("memory/conversation/
↪summary.j2")), kw_only=True)
    summarize_conversation_get_template: J2 = field(
        default=Factory(lambda: J2("memory/conversation/summarize_conversation.j2")),
        kw_only=True,
    )

    def to_prompt_stack(self, last_n: Optional[int] = None) -> PromptStack:
        stack = PromptStack()
        if self.summary:
            stack.add_user_message(self.summary_get_template.render(summary=self.
↪summary))

        for r in self.unsummarized_runs(last_n):
            stack.add_user_message(r.input)
            stack.add_assistant_message(r.output)

        return stack

    def unsummarized_runs(self, last_n: Optional[int] = None) -> list[Run]:
        summary_index_runs = self.runs[self.summary_index :]

        if last_n:
            last_n_runs = self.runs[-last_n:]

            if len(summary_index_runs) > len(last_n_runs):
                return last_n_runs
            return summary_index_runs
        return summary_index_runs

    def try_add_run(self, run: Run) -> None:
        self.runs.append(run)
        unsummarized_runs = self.unsummarized_runs()
        runs_to_summarize = unsummarized_runs[: max(0, len(unsummarized_runs) - self.
↪offset)]

        if len(runs_to_summarize) > 0:
            self.summary = self.summarize_runs(self.summary, runs_to_summarize)
            self.summary_index = 1 + self.runs.index(runs_to_summarize[-1])

    def summarize_runs(self, previous_summary: str | None, runs: list[Run]) -> str |␣
↪None:
        try:
            if len(runs) > 0:
                summary = self.summarize_conversation_get_template.
↪render(summary=previous_summary, runs=runs)
                return self.prompt_driver.run(
                    PromptStack(messages=[Message(summary, role=Message.USER_ROLE)]),
```

(continues on next page)

(continued from previous page)

```
                ).to_text()
            return previous_summary
        except Exception as e:
            logging.exception("Error summarizing memory: %s(%s)", type(e).__name__, e)

            return previous_summary
```

- `offset = field(default=1, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `prompt_driver = field(kw_only=True, default=Factory(lambda: Defaults.drivers_config.prompt_driver))` class-attribute instance-attribute
- `summarize_conversation_get_template = field(default=Factory(lambda: J2('memory/conversation/summarize_conversation.j2')), kw_only=True)` class-attribute instance-attribute
- `summary = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `summary_get_template = field(default=Factory(lambda: J2('memory/conversation/summary.j2')), kw_only=True)` class-attribute instance-attribute
- `summary_index = field(default=0, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

summarize_runs(previous_summary, runs)

Source Code in griptape/memory/structure/summary_conversation_memory.py

```
def summarize_runs(self, previous_summary: str | None, runs: list[Run]) -> str | None:
    try:
        if len(runs) > 0:
            summary = self.summarize_conversation_get_template.
↪render(summary=previous_summary, runs=runs)
            return self.prompt_driver.run(
                PromptStack(messages=[Message(summary, role=Message.USER_ROLE)]),
            ).to_text()
        return previous_summary
    except Exception as e:
        logging.exception("Error summarizing memory: %s(%s)", type(e).__name__, e)

        return previous_summary
```

to_prompt_stack(last_n=None)

Source Code in griptape/memory/structure/summary_conversation_memory.py

```
def to_prompt_stack(self, last_n: Optional[int] = None) -> PromptStack:
    stack = PromptStack()
    if self.summary:
        stack.add_user_message(self.summary_get_template.render(summary=self.summary))

    for r in self.unsummarized_runs(last_n):
        stack.add_user_message(r.input)
```

(continues on next page)

(continued from previous page)

```
        stack.add_assistant_message(r.output)

    return stack
```

try_add_run(run)

Source Code in griptape/memory/structure/summary_conversation_memory.py

```
def try_add_run(self, run: Run) -> None:
    self.runs.append(run)
    unsummarized_runs = self.unsummarized_runs()
    runs_to_summarize = unsummarized_runs[: max(0, len(unsummarized_runs) - self.
↪offset)]

    if len(runs_to_summarize) > 0:
        self.summary = self.summarize_runs(self.summary, runs_to_summarize)
        self.summary_index = 1 + self.runs.index(runs_to_summarize[-1])
```

unsummarized_runs(last_n=None)

Source Code in griptape/memory/structure/summary_conversation_memory.py

```
def unsummarized_runs(self, last_n: Optional[int] = None) -> list[Run]:
    summary_index_runs = self.runs[self.summary_index :]

    if last_n:
        last_n_runs = self.runs[-last_n:]

        if len(summary_index_runs) > len(last_n_runs):
            return last_n_runs
        return summary_index_runs
    return summary_index_runs
```

task

storage

- __all__ = ['BaseArtifactStorage', 'BlobArtifactStorage', 'TextArtifactStorage'] module-attribute

Bases:

ABC

Source Code in griptape/memory/task/storage/base_artifact_storage.py

```
@define
class BaseArtifactStorage(ABC):
    @abstractmethod
    def store_artifact(self, namespace: str, artifact: BaseArtifact) -> None: ...

    @abstractmethod
    def load_artifacts(self, namespace: str) -> ListArtifact: ...
```

(continues on next page)

(continued from previous page)

```
    @abstractmethod
    def can_store(self, artifact: BaseArtifact) -> bool: ...
```

can_store(artifact)abstractmethod

Source Code in griptape/memory/task/storage/base_artifact_storage.py

```
@abstractmethod
def can_store(self, artifact: BaseArtifact) -> bool: ...
```

load_artifacts(namespace)abstractmethod

Source Code in griptape/memory/task/storage/base_artifact_storage.py

```
@abstractmethod
def load_artifacts(self, namespace: str) -> ListArtifact: ...
```

store_artifact(namespace, artifact)abstractmethod

Source Code in griptape/memory/task/storage/base_artifact_storage.py

```
@abstractmethod
def store_artifact(self, namespace: str, artifact: BaseArtifact) -> None: ...
```

BlobArtifactStorage

Bases: BaseArtifactStorage “)

Source Code in griptape/memory/task/storage/blob_artifact_storage.py

```
@define
class BlobArtifactStorage(BaseArtifactStorage):
    blobs: dict[str, list[BlobArtifact]] = field(factory=dict, kw_only=True)

    def can_store(self, artifact: BaseArtifact) -> bool:
        return isinstance(artifact, BlobArtifact)

    def store_artifact(self, namespace: str, artifact: BaseArtifact) -> None:
        if isinstance(artifact, BlobArtifact):
            if namespace not in self.blobs:
                self.blobs[namespace] = []

            self.blobs[namespace].append(artifact)
        else:
            raise ValueError("Artifact must be of instance BlobArtifact")

    def load_artifacts(self, namespace: str) -> ListArtifact:
        return ListArtifact(next((blobs for key, blobs in self.blobs.items() if key␣
↪== namespace), []))
```

- `blobs = field(factory=dict, kw_only=True)` class-attribute instance-attribute

can_store(artifact)

Source Code in griptape/memory/task/storage/blob_artifact_storage.py

```
def can_store(self, artifact: BaseArtifact) -> bool:
    return isinstance(artifact, BlobArtifact)
```

load_artifacts(namespace)

Source Code in griptape/memory/task/storage/blob_artifact_storage.py

```
def load_artifacts(self, namespace: str) -> ListArtifact:
    return ListArtifact(next((blobs for key, blobs in self.blobs.items() if key ==␣
↪namespace), []))
```

store_artifact(namespace, artifact)

Source Code in griptape/memory/task/storage/blob_artifact_storage.py

```
def store_artifact(self, namespace: str, artifact: BaseArtifact) -> None:
    if isinstance(artifact, BlobArtifact):
        if namespace not in self.blobs:
            self.blobs[namespace] = []

        self.blobs[namespace].append(artifact)
    else:
        raise ValueError("Artifact must be of instance BlobArtifact")
```

TextArtifactStorage

Bases: BaseArtifactStorage “)

Source Code in griptape/memory/task/storage/text_artifact_storage.py

```
@define(kw_only=True)
class TextArtifactStorage(BaseArtifactStorage):
    vector_store_driver: BaseVectorStoreDriver = field(
        default=Factory(lambda: Defaults.drivers_config.vector_store_driver)
    )

    def can_store(self, artifact: BaseArtifact) -> bool:
        return isinstance(artifact, TextArtifact)

    def store_artifact(self, namespace: str, artifact: BaseArtifact) -> None:
        if isinstance(artifact, TextArtifact):
            self.vector_store_driver.upsert(artifact, namespace=namespace)
        else:
            raise ValueError("Artifact must be of instance TextArtifact")

    def load_artifacts(self, namespace: str) -> ListArtifact:
        return self.vector_store_driver.load_artifacts(namespace=namespace)
```

- `vector_store_driver = field(default=Factory(lambda: Defaults.drivers_config.vector_store_driver))` class-attribute instance-attribute

can_store(artifact)

Source Code in griptape/memory/task/storage/text_artifact_storage.py

```
def can_store(self, artifact: BaseArtifact) -> bool:
    return isinstance(artifact, TextArtifact)
```

load_artifacts(namespace)

Source Code in griptape/memory/task/storage/text_artifact_storage.py

```
def load_artifacts(self, namespace: str) -> ListArtifact:
    return self.vector_store_driver.load_artifacts(namespace=namespace)
```

store_artifact(namespace, artifact)

Source Code in griptape/memory/task/storage/text_artifact_storage.py

```
def store_artifact(self, namespace: str, artifact: BaseArtifact) -> None:
    if isinstance(artifact, TextArtifact):
        self.vector_store_driver.upsert(artifact, namespace=namespace)
    else:
        raise ValueError("Artifact must be of instance TextArtifact")
```

base_artifact_storage

Bases:

ABC

Source Code in griptape/memory/task/storage/base_artifact_storage.py

```
@define
class BaseArtifactStorage(ABC):
    @abstractmethod
    def store_artifact(self, namespace: str, artifact: BaseArtifact) -> None: ...

    @abstractmethod
    def load_artifacts(self, namespace: str) -> ListArtifact: ...

    @abstractmethod
    def can_store(self, artifact: BaseArtifact) -> bool: ...
```

can_store(artifact)abstractmethod

Source Code in griptape/memory/task/storage/base_artifact_storage.py

```
@abstractmethod
def can_store(self, artifact: BaseArtifact) -> bool: ...
```

load_artifacts(namespace)abstractmethod

Source Code in griptape/memory/task/storage/base_artifact_storage.py

```
@abstractmethod
def load_artifacts(self, namespace: str) -> ListArtifact: ...
```

store_artifact(namespace, artifact)abstractmethod

Source Code in griptape/memory/task/storage/base_artifact_storage.py

```
@abstractmethod
def store_artifact(self, namespace: str, artifact: BaseArtifact) -> None: ...
```

blob_artifact_storage

Bases: BaseArtifactStorage “)

Source Code in griptape/memory/task/storage/blob_artifact_storage.py

```
@define
class BlobArtifactStorage(BaseArtifactStorage):
    blobs: dict[str, list[BlobArtifact]] = field(factory=dict, kw_only=True)

    def can_store(self, artifact: BaseArtifact) -> bool:
        return isinstance(artifact, BlobArtifact)

    def store_artifact(self, namespace: str, artifact: BaseArtifact) -> None:
        if isinstance(artifact, BlobArtifact):
            if namespace not in self.blobs:
                self.blobs[namespace] = []

            self.blobs[namespace].append(artifact)
        else:
            raise ValueError("Artifact must be of instance BlobArtifact")

    def load_artifacts(self, namespace: str) -> ListArtifact:
        return ListArtifact(next((blobs for key, blobs in self.blobs.items() if key␣
↪== namespace), []))
```

- `blobs = field(factory=dict, kw_only=True)` class-attribute instance-attribute

can_store(artifact)

Source Code in griptape/memory/task/storage/blob_artifact_storage.py

```
def can_store(self, artifact: BaseArtifact) -> bool:
    return isinstance(artifact, BlobArtifact)
```

load_artifacts(namespace)

Source Code in griptape/memory/task/storage/blob_artifact_storage.py

```
def load_artifacts(self, namespace: str) -> ListArtifact:
    return ListArtifact(next((blobs for key, blobs in self.blobs.items() if key ==␣
↪namespace), []))
```

store_artifact(namespace, artifact)

Source Code in griptape/memory/task/storage/blob_artifact_storage.py

```
def store_artifact(self, namespace: str, artifact: BaseArtifact) -> None:
    if isinstance(artifact, BlobArtifact):
        if namespace not in self.blobs:
            self.blobs[namespace] = []

        self.blobs[namespace].append(artifact)
    else:
        raise ValueError("Artifact must be of instance BlobArtifact")
```

text_artifact_storage

Bases: BaseArtifactStorage “)

Source Code in griptape/memory/task/storage/text_artifact_storage.py

```
@define(kw_only=True)
class TextArtifactStorage(BaseArtifactStorage):
    vector_store_driver: BaseVectorStoreDriver = field(
        default=Factory(lambda: Defaults.drivers_config.vector_store_driver)
    )

    def can_store(self, artifact: BaseArtifact) -> bool:
        return isinstance(artifact, TextArtifact)

    def store_artifact(self, namespace: str, artifact: BaseArtifact) -> None:
        if isinstance(artifact, TextArtifact):
            self.vector_store_driver.upsert(artifact, namespace=namespace)
        else:
            raise ValueError("Artifact must be of instance TextArtifact")

    def load_artifacts(self, namespace: str) -> ListArtifact:
        return self.vector_store_driver.load_artifacts(namespace=namespace)
```

- `vector_store_driver = field(default=Factory(lambda: Defaults.drivers_config.vector_store_driver))` class-attribute instance-attribute

can_store(artifact)

Source Code in griptape/memory/task/storage/text_artifact_storage.py

```
def can_store(self, artifact: BaseArtifact) -> bool:
    return isinstance(artifact, TextArtifact)
```

load_artifacts(namespace)

Source Code in griptape/memory/task/storage/text_artifact_storage.py

```
def load_artifacts(self, namespace: str) -> ListArtifact:
    return self.vector_store_driver.load_artifacts(namespace=namespace)
```

store_artifact(namespace, artifact)

Source Code in griptape/memory/task/storage/text_artifact_storage.py

```
def store_artifact(self, namespace: str, artifact: BaseArtifact) -> None:
    if isinstance(artifact, TextArtifact):
        self.vector_store_driver.upsert(artifact, namespace=namespace)
    else:
        raise ValueError("Artifact must be of instance TextArtifact")
```

task_memory

Bases: *ActivityMixin* “) , *SerializableMixin* ”)

Source Code in griptape/memory/task/task_memory.py

```
@define
class TaskMemory(ActivityMixin, SerializableMixin):
    name: str = field(
        default=Factory(lambda self: self.__class__.__name__, takes_self=True),
        kw_only=True,
        metadata={"serializable": True},
    )
    artifact_storages: dict[type, BaseArtifactStorage] = field(
        default=Factory(
            lambda: {
                TextArtifact: TextArtifactStorage(),
                BlobArtifact: BlobArtifactStorage(),
            }
        ),
        kw_only=True,
    )
    namespace_storage: dict[str, BaseArtifactStorage] = field(
        factory=dict, kw_only=True, metadata={"serializable": True}
    )
    namespace_metadata: dict[str, Any] = field(factory=dict, kw_only=True, metadata={
↪"serializable": True})

    @artifact_storages.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_artifact_storages(self, _: Attribute, artifact_storage: dict[type,␣
↪BaseArtifactStorage]) -> None:
        seen_types = []

        for storage in artifact_storage.values():
            if type(storage) in seen_types:
                raise ValueError("cant have more than memory storage of the same type
```

(continues on next page)

(continued from previous page)

```
↪")

            seen_types.append(type(storage))

    def get_storage_for(self, artifact: BaseArtifact) ->␣
↪Optional[BaseArtifactStorage]:
        def find_storage(a: BaseArtifact) -> Optional[BaseArtifactStorage]:
            return next((v for k, v in self.artifact_storages.items() if isinstance(a,
↪ k)), None)

        if isinstance(artifact, ListArtifact):
            if artifact.has_items():
                return find_storage(artifact.value[0])
            return None
        return find_storage(artifact)

    def process_output(
        self,
        tool_activity: Callable,
        subtask: ActionsSubtask,
        output_artifact: BaseArtifact,
    ) -> BaseArtifact:
        from griptape.utils import J2

        tool_name = getattr(getattr(tool_activity, "__self__"), "name")
        activity_name = getattr(tool_activity, "name")
        namespace = output_artifact.name

        if output_artifact:
            result = self.store_artifact(namespace, output_artifact)

            if result:
                return result
            self.namespace_metadata[namespace] = subtask.actions_to_json()

            output = J2("memory/tool.j2").render(
                memory_name=self.name,
                tool_name=tool_name,
                activity_name=activity_name,
                artifact_namespace=namespace,
            )

            if subtask.structure and subtask.structure.meta_memory:
                subtask.structure.meta_memory.add_entry(
                    ActionSubtaskMetaEntry(
                        thought=subtask.thought,
                        actions=subtask.actions_to_json(),
                        answer=output,
                    ),
                )

            return InfoArtifact(output, name=namespace)
```

(continues on next page)

(continued from previous page)

```
        return InfoArtifact("tool output is empty")

    def store_artifact(self, namespace: str, artifact: BaseArtifact) ->␣
↪Optional[BaseArtifact]:
        namespace_storage = self.namespace_storage.get(namespace)
        storage = self.get_storage_for(artifact)

        if not storage:
            return artifact
        if namespace_storage and namespace_storage != storage:
            return ErrorArtifact("error storing tool output in memory")
        if storage:
            if isinstance(artifact, ListArtifact):
                for a in artifact.value:
                    storage.store_artifact(namespace, a)

                self.namespace_storage[namespace] = storage

                return None
            if isinstance(artifact, BaseArtifact):
                storage.store_artifact(namespace, artifact)

                self.namespace_storage[namespace] = storage

                return None
            return ErrorArtifact("error storing tool output in memory")
        return ErrorArtifact("error storing tool output in memory")

    def load_artifacts(self, namespace: str) -> ListArtifact:
        storage = self.namespace_storage.get(namespace)

        if storage:
            return storage.load_artifacts(namespace)
        return ListArtifact()

    def find_input_memory(self, memory_name: str) -> Optional[TaskMemory]:
        if memory_name == self.name:
            return self
        return None
```

- `artifact_storages = field(default=Factory(lambda: {TextArtifact: TextArtifactStorage(), BlobArtifact: BlobArtifactStorage()}), kw_only=True)` class-attribute instance-attribute
- `name = field(default=Factory(lambda self: self.__class__.__name__, takes_self=True), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `namespace_metadata = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `namespace_storage = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

find_input_memory(memory_name)

Source Code in griptape/memory/task/task_memory.py

```
def find_input_memory(self, memory_name: str) -> Optional[TaskMemory]:
    if memory_name == self.name:
        return self
    return None
```

get_storage_for(artifact)

Source Code in griptape/memory/task/task_memory.py

```
def get_storage_for(self, artifact: BaseArtifact) -> Optional[BaseArtifactStorage]:
    def find_storage(a: BaseArtifact) -> Optional[BaseArtifactStorage]:
        return next((v for k, v in self.artifact_storages.items() if isinstance(a,␣
↪k)), None)

    if isinstance(artifact, ListArtifact):
        if artifact.has_items():
            return find_storage(artifact.value[0])
        return None
    return find_storage(artifact)
```

load_artifacts(namespace)

Source Code in griptape/memory/task/task_memory.py

```
def load_artifacts(self, namespace: str) -> ListArtifact:
    storage = self.namespace_storage.get(namespace)

    if storage:
        return storage.load_artifacts(namespace)
    return ListArtifact()
```

process_output(tool_activity, subtask, output_artifact)

Source Code in griptape/memory/task/task_memory.py

```
def process_output(
    self,
    tool_activity: Callable,
    subtask: ActionsSubtask,
    output_artifact: BaseArtifact,
) -> BaseArtifact:
    from griptape.utils import J2

    tool_name = getattr(getattr(tool_activity, "__self__"), "name")
    activity_name = getattr(tool_activity, "name")
    namespace = output_artifact.name

    if output_artifact:
        result = self.store_artifact(namespace, output_artifact)
```

(continues on next page)

(continued from previous page)

```
        if result:
            return result
        self.namespace_metadata[namespace] = subtask.actions_to_json()

        output = J2("memory/tool.j2").render(
            memory_name=self.name,
            tool_name=tool_name,
            activity_name=activity_name,
            artifact_namespace=namespace,
        )

        if subtask.structure and subtask.structure.meta_memory:
            subtask.structure.meta_memory.add_entry(
                ActionSubtaskMetaEntry(
                    thought=subtask.thought,
                    actions=subtask.actions_to_json(),
                    answer=output,
                ),
            )

        return InfoArtifact(output, name=namespace)
    return InfoArtifact("tool output is empty")
```

store_artifact(namespace, artifact)

Source Code in griptape/memory/task/task_memory.py

```
def store_artifact(self, namespace: str, artifact: BaseArtifact) ->␣
↪Optional[BaseArtifact]:
    namespace_storage = self.namespace_storage.get(namespace)
    storage = self.get_storage_for(artifact)

    if not storage:
        return artifact
    if namespace_storage and namespace_storage != storage:
        return ErrorArtifact("error storing tool output in memory")
    if storage:
        if isinstance(artifact, ListArtifact):
            for a in artifact.value:
                storage.store_artifact(namespace, a)

            self.namespace_storage[namespace] = storage

            return None
        if isinstance(artifact, BaseArtifact):
            storage.store_artifact(namespace, artifact)

            self.namespace_storage[namespace] = storage

            return None
        return ErrorArtifact("error storing tool output in memory")
```

(continues on next page)

(continued from previous page)

```
    return ErrorArtifact("error storing tool output in memory")
```

validate*artifact_storages(*, artifact_storage)

Source Code in griptape/memory/task/task_memory.py

```
@artifact_storages.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_artifact_storages(self, _: Attribute, artifact_storage: dict[type,␣
↪BaseArtifactStorage]) -> None:
    seen_types = []

    for storage in artifact_storage.values():
        if type(storage) in seen_types:
            raise ValueError("cant have more than memory storage of the same type")

        seen_types.append(type(storage))
```

11.5.13 mixins

actions_subtask_origin_mixin

Source Code in griptape/mixins/actions_subtask_origin_mixin.py

```
@define(slots=False)
class ActionsSubtaskOriginMixin:
    @abstractmethod
    def find_tool(self, tool_name: str) -> BaseTool: ...

    @abstractmethod
    def find_memory(self, memory_name: str) -> TaskMemory: ...

    @abstractmethod
    def find_subtask(self, subtask_id: str) -> BaseSubtask: ...

    @abstractmethod
    def add_subtask(self, subtask: BaseSubtask) -> BaseSubtask: ...

    @abstractmethod
    def actions_schema(self) -> Schema: ...

    def _actions_schema_for_tools(self, tools: list[BaseTool]) -> Schema:
        action_schemas = []

        for tool in tools:
            for activity_schema in tool.activity_schemas():
                action_schema = activity_schema.schema
                tag_key = Literal("tag", description="Unique tag name for action␣
↪execution.")

                action_schema[tag_key] = str

                action_schemas.append(action_schema)
```

(continues on next page)

(continued from previous page)

```
        return Schema(description="JSON schema for an array of actions.",␣
↪schema=action_schemas)
```

_actions_schema_for_tools(tools)

Source Code in griptape/mixins/actions_subtask_origin_mixin.py

```
def _actions_schema_for_tools(self, tools: list[BaseTool]) -> Schema:
    action_schemas = []

    for tool in tools:
        for activity_schema in tool.activity_schemas():
            action_schema = activity_schema.schema
            tag_key = Literal("tag", description="Unique tag name for action␣
↪execution.")

            action_schema[tag_key] = str

            action_schemas.append(action_schema)

    return Schema(description="JSON schema for an array of actions.", schema=action_
↪schemas)
```

actions_schema()abstractmethod

Source Code in griptape/mixins/actions_subtask_origin_mixin.py

```
@abstractmethod
def actions_schema(self) -> Schema: ...
```

add_subtask(subtask)abstractmethod

Source Code in griptape/mixins/actions_subtask_origin_mixin.py

```
@abstractmethod
def add_subtask(self, subtask: BaseSubtask) -> BaseSubtask: ...
```

find_memory(memory_name)abstractmethod

Source Code in griptape/mixins/actions_subtask_origin_mixin.py

```
@abstractmethod
def find_memory(self, memory_name: str) -> TaskMemory: ...
```

find_subtask(subtask_id)abstractmethod

Source Code in griptape/mixins/actions_subtask_origin_mixin.py

```
@abstractmethod
def find_subtask(self, subtask_id: str) -> BaseSubtask: ...
```

find_tool(tool_name)abstractmethod

Source Code in griptape/mixins/actions_subtask_origin_mixin.py

```
@abstractmethod
def find_tool(self, tool_name: str) -> BaseTool: ...
```

activity_mixin

Provides Tool Activity management functionality to Tools.

Attributes

Name	Type	Description
<a href="activity_mixin.mdx#griptape.mixins.activity_mixin.ActivityMixin.allowlist" title="allowlist = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.mixins.activity_mixin.ActivityMixin.allowlist)"><code class="language-python">	<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">Optional[list[str]]</code></a	List of Tool Activities to include in the Tool schema.
<a href="activity_mixin.mdx#griptape.mixins.activity_mixin.ActivityMixin.denylist" title="denylist = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.mixins.activity_mixin.ActivityMixin.denylist)"><code class="language-python">	<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-python">Optional[list[str]]</code></a	List of Tool Activities to remove from the Tool schema.

continues on next page

Table 334 – continued from previous page

Name	Type	Description
<a href="activity_mixin.mdx#griptape.mixins.activity_mixin.ActivityMixin.extra_schema_properties" title="extra_schema_properties = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.mixins.activity_mixin.ActivityMixin.extra_schema_properties)"><code class="language-python">	*Optional[dict[str, dict]]*	Mapping of Activity name and extra properties to include in the activity's schema.

Source Code in griptape/mixins/activity_mixin.py

```
@define(slots=False)
class ActivityMixin:
    """Provides Tool Activity management functionality to Tools.

    Attributes:
        allowlist: List of Tool Activities to include in the Tool schema.
        denylist: List of Tool Activities to remove from the Tool schema.
        extra_schema_properties: Mapping of Activity name and extra properties to␣
↪include in the activitys schema.
    """

    allowlist: Optional[list[str]] = field(default=None, kw_only=True)
    denylist: Optional[list[str]] = field(default=None, kw_only=True)
    extra_schema_properties: Optional[dict[str, dict]] = field(default=None, kw_
↪only=True)

    @allowlist.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_allowlist(self, _: Attribute, allowlist: Optional[list[str]]) ->␣
↪None:
        if allowlist is None:
            return

        for activity_name in allowlist:
            self._validate_tool_activity(activity_name)

    @denylist.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_denylist(self, _: Attribute, denylist: Optional[list[str]]) -> None:
```

(continues on next page)

(continued from previous page)

```
        if denylist is None:
            return

        for activity_name in denylist:
            self._validate_tool_activity(activity_name)

    def enable_activities(self) -> None:
        self.allowlist = None
        self.denylist = None

    def disable_activities(self) -> None:
        self.allowlist = []
        self.denylist = None

    # This method has to remain a method and cant be decorated with @property because
    # of the max depth recursion issue in `inspect.getmembers`.
    def activities(self) -> list[Callable]:
        methods = []

        for name, method in inspect.getmembers(self, predicate=inspect.ismethod):
            allowlist_condition = self.allowlist is None or name in self.allowlist
            denylist_condition = self.denylist is None or name not in self.denylist

            if getattr(method, "is_activity", False) and allowlist_condition and␣
↪denylist_condition:
                methods.append(method)

        return methods

    def find_activity(self, name: str) -> Optional[Callable]:
        for activity in self.activities():
            if getattr(activity, "is_activity", False) and getattr(activity, "name")␣
↪== name:
                return activity

        return None

    def activity_name(self, activity: Callable) -> str:
        if activity is None or not getattr(activity, "is_activity", False):
            raise Exception("This method is not an activity.")
        return getattr(activity, "name")

    def activity_description(self, activity: Callable) -> str:
        if activity is None or not getattr(activity, "is_activity", False):
            raise Exception("This method is not an activity.")
        return Template(getattr(activity, "config")["description"]).render({"_self":␣
↪self})

    def activity_schema(self, activity: Callable) -> Optional[Union[Schema,␣
↪type[BaseModel]]]:
        if activity is None or not getattr(activity, "is_activity", False):
            raise Exception("This method is not an activity.")
```

(continues on next page)

(continued from previous page)

```
        if getattr(activity, "config")["schema"] is not None:
            config_schema = getattr(activity, "config")["schema"]
            if isinstance(config_schema, Schema):
                # Need to deepcopy to avoid modifying the original schema
                config_schema = deepcopy(getattr(activity, "config")["schema"])
            elif isinstance(config_schema, Callable) and not isinstance(config_schema,
↪ type):
                config_schema = config_schema(self)
            activity_name = self.activity_name(activity)

            if isinstance(config_schema, Schema):
                if self.extra_schema_properties is not None and activity_name in self.
↪extra_schema_properties:
                    config_schema.schema.update(self.extra_schema_properties[activity_
↪name])

                return Schema({"values": config_schema})
            return create_model(config_schema.__name__, values=(config_schema,␣
↪Field(config_schema)))
        return None

    def to_activity_json_schema(self, activity: Callable, schema_id: str) -> dict:
        schema = self.activity_schema(activity)
        if schema is None:
            schema = Schema({})

        if isinstance(schema, Schema):
            json_schema = schema.json_schema(schema_id)
        else:
            json_schema = build_strict_schema(schema.model_json_schema(), schema_id)

        return json_schema

    def validate_activity_schema(self, activity_schema: Union[Schema,␣
↪type[BaseModel]], params: dict) -> None:
        try:
            if isinstance(activity_schema, Schema):
                activity_schema.validate(params)
            else:
                activity_schema.model_validate(params)
        except (SchemaError, ValidationError) as e:
            raise ValueError(e) from e

    def _validate_tool_activity(self, activity_name: str) -> None:
        tool = self.__class__

        activity = getattr(tool, activity_name, None)

        if not activity or not getattr(activity, "is_activity", False):
            raise ValueError(f"activity {activity_name} is not a valid activity for
↪{tool}")
```

- `allowlist = field(default=None, kw_only=True)` class-attribute instance-attribute

- `denylist = field(default=None, kw_only=True)` class-attribute instance-attribute
- `extra_schema_properties = field(default=None, kw_only=True)` class-attribute instance-attribute

_validate_tool_activity(activity_name)

Source Code in griptape/mixins/activity_mixin.py

```
def _validate_tool_activity(self, activity_name: str) -> None:
    tool = self.__class__

    activity = getattr(tool, activity_name, None)

    if not activity or not getattr(activity, "is_activity", False):
        raise ValueError(f"activity {activity_name} is not a valid activity for {tool}
↪")
```

activities()

Source Code in griptape/mixins/activity_mixin.py

```
def activities(self) -> list[Callable]:
    methods = []

    for name, method in inspect.getmembers(self, predicate=inspect.ismethod):
        allowlist_condition = self.allowlist is None or name in self.allowlist
        denylist_condition = self.denylist is None or name not in self.denylist

        if getattr(method, "is_activity", False) and allowlist_condition and denylist_
↪condition:
            methods.append(method)

    return methods
```

activity_description(activity)

Source Code in griptape/mixins/activity_mixin.py

```
def activity_description(self, activity: Callable) -> str:
    if activity is None or not getattr(activity, "is_activity", False):
        raise Exception("This method is not an activity.")
    return Template(getattr(activity, "config")["description"]).render({"_self": self}
↪)
```

activity_name(activity)

Source Code in griptape/mixins/activity_mixin.py

```
def activity_name(self, activity: Callable) -> str:
    if activity is None or not getattr(activity, "is_activity", False):
        raise Exception("This method is not an activity.")
    return getattr(activity, "name")
```

activity_schema(activity)

Source Code in griptape/mixins/activity_mixin.py

```
def activity_schema(self, activity: Callable) -> Optional[Union[Schema,␣
↪type[BaseModel]]]:
    if activity is None or not getattr(activity, "is_activity", False):
        raise Exception("This method is not an activity.")
    if getattr(activity, "config")["schema"] is not None:
        config_schema = getattr(activity, "config")["schema"]
        if isinstance(config_schema, Schema):
            # Need to deepcopy to avoid modifying the original schema
            config_schema = deepcopy(getattr(activity, "config")["schema"])
        elif isinstance(config_schema, Callable) and not isinstance(config_schema,␣
↪type):
            config_schema = config_schema(self)
        activity_name = self.activity_name(activity)

        if isinstance(config_schema, Schema):
            if self.extra_schema_properties is not None and activity_name in self.
↪extra_schema_properties:
                config_schema.schema.update(self.extra_schema_properties[activity_
↪name])

            return Schema({"values": config_schema})
        return create_model(config_schema.__name__, values=(config_schema,␣
↪Field(config_schema)))
    return None
```

disable_activities()

Source Code in griptape/mixins/activity_mixin.py

```
def disable_activities(self) -> None:
    self.allowlist = []
    self.denylist = None
```

enable_activities()

Source Code in griptape/mixins/activity_mixin.py

```
def enable_activities(self) -> None:
    self.allowlist = None
    self.denylist = None
```

find_activity(name)

Source Code in griptape/mixins/activity_mixin.py

```
def find_activity(self, name: str) -> Optional[Callable]:
    for activity in self.activities():
        if getattr(activity, "is_activity", False) and getattr(activity, "name") ==␣
↪name:
```

(continues on next page)

(continued from previous page)

```
            return activity

    return None
```

to_activity_json_schema(activity, schema_id)

Source Code in griptape/mixins/activity_mixin.py

```
def to_activity_json_schema(self, activity: Callable, schema_id: str) -> dict:
    schema = self.activity_schema(activity)
    if schema is None:
        schema = Schema({})

    if isinstance(schema, Schema):
        json_schema = schema.json_schema(schema_id)
    else:
        json_schema = build_strict_schema(schema.model_json_schema(), schema_id)

    return json_schema
```

validate_activity_schema(activity_schema, params)

Source Code in griptape/mixins/activity_mixin.py

```
def validate_activity_schema(self, activity_schema: Union[Schema, type[BaseModel]],␣
↪params: dict) -> None:
    try:
        if isinstance(activity_schema, Schema):
            activity_schema.validate(params)
        else:
            activity_schema.model_validate(params)
    except (SchemaError, ValidationError) as e:
        raise ValueError(e) from e
```

validate*allowlist(*, allowlist)

Source Code in griptape/mixins/activity_mixin.py

```
@allowlist.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_allowlist(self, _: Attribute, allowlist: Optional[list[str]]) -> None:
    if allowlist is None:
        return

    for activity_name in allowlist:
        self._validate_tool_activity(activity_name)
```

validate*denylist*(, denylist)

Source Code in griptape/mixins/activity_mixin.py

```
@denylist.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_denylist(self, _: Attribute, denylist: Optional[list[str]]) -> None:
    if denylist is None:
        return

    for activity_name in denylist:
        self._validate_tool_activity(activity_name)
```

artifact_file_output_mixin

Source Code in griptape/mixins/artifact_file_output_mixin.py

```
@define(slots=False)
class ArtifactFileOutputMixin:
    output_dir: Optional[str] = field(default=None, kw_only=True)
    output_file: Optional[str] = field(default=None, kw_only=True)

    @output_dir.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_output_dir(self, _: Attribute, output_dir: str) -> None:
        if not output_dir:
            return

        if self.output_file:
            raise ValueError("Cant have both output_dir and output_file specified.")

    @output_file.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_output_file(self, _: Attribute, output_file: str) -> None:
        if not output_file:
            return

        if self.output_dir:
            raise ValueError("Cant have both output_dir and output_file specified.")

    def _write_to_file(self, artifact: BaseArtifact) -> None:
        if self.output_file:
            outfile = self.output_file
        elif self.output_dir:
            outfile = os.path.join(self.output_dir, artifact.name)
        else:
            raise ValueError("No output_file or output_dir specified.")

        if os.path.dirname(outfile):
            os.makedirs(os.path.dirname(outfile), exist_ok=True)

        Path(outfile).write_bytes(artifact.to_bytes())
```

- `output_dir = field(default=None, kw_only=True)` class-attribute instance-attribute

- `output_file = field(default=None, kw_only=True)` class-attribute instance-attribute

_write_to_file(artifact)

Source Code in griptape/mixins/artifact_file_output_mixin.py

```
def _write_to_file(self, artifact: BaseArtifact) -> None:
    if self.output_file:
        outfile = self.output_file
    elif self.output_dir:
        outfile = os.path.join(self.output_dir, artifact.name)
    else:
        raise ValueError("No output_file or output_dir specified.")

    if os.path.dirname(outfile):
        os.makedirs(os.path.dirname(outfile), exist_ok=True)

    Path(outfile).write_bytes(artifact.to_bytes())
```

validate*output_dir*(, output_dir)

Source Code in griptape/mixins/artifact_file_output_mixin.py

```
@output_dir.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_output_dir(self, _: Attribute, output_dir: str) -> None:
    if not output_dir:
        return

    if self.output_file:
        raise ValueError("Cant have both output_dir and output_file specified.")
```

validate*output_file*(, output_file)

Source Code in griptape/mixins/artifact_file_output_mixin.py

```
@output_file.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_output_file(self, _: Attribute, output_file: str) -> None:
    if not output_file:
        return

    if self.output_dir:
        raise ValueError("Cant have both output_dir and output_file specified.")
```

exponential_backoff_mixin

Bases:

ABC

Source Code in griptape/mixins/exponential_backoff_mixin.py

```
@define(slots=False)
class ExponentialBackoffMixin(ABC):
    min_retry_delay: float = field(default=2, kw_only=True)
    max_retry_delay: float = field(default=10, kw_only=True)
    max_attempts: int = field(default=2, kw_only=True)
    after_hook: Callable = field(default=lambda s: logging.warning(s), kw_only=True)
    ignored_exception_types: tuple[type[Exception], ...] = field(factory=tuple, kw_
↪only=True)

    def retrying(self) -> Retrying:
        return Retrying(
            wait=wait_exponential(min=self.min_retry_delay, max=self.max_retry_delay),
            retry=retry_if_not_exception_type(self.ignored_exception_types),
            stop=stop_after_attempt(self.max_attempts),
            reraise=True,
            after=self.after_hook,
        )
```

- `after_hook = field(default=lambda s: logging.warning(s), kw_only=True)` class-attribute instance-attribute
- `ignored_exception_types = field(factory=tuple, kw_only=True)` class-attribute instance-attribute
- `max_attempts = field(default=2, kw_only=True)` class-attribute instance-attribute
- `max_retry_delay = field(default=10, kw_only=True)` class-attribute instance-attribute
- `min_retry_delay = field(default=2, kw_only=True)` class-attribute instance-attribute

retrying()

Source Code in griptape/mixins/exponential_backoff_mixin.py

```
def retrying(self) -> Retrying:
    return Retrying(
        wait=wait_exponential(min=self.min_retry_delay, max=self.max_retry_delay),
        retry=retry_if_not_exception_type(self.ignored_exception_types),
        stop=stop_after_attempt(self.max_attempts),
        reraise=True,
        after=self.after_hook,
    )
```

futures_executor_mixin

Bases:

ABC

Source Code in griptape/mixins/futures_executor_mixin.py

```
@define(slots=False, kw_only=True)
class FuturesExecutorMixin(ABC):
    create_futures_executor: Callable[[], futures.Executor] = field(
        default=Factory(lambda: lambda: futures.ThreadPoolExecutor()),
    )
```

(continues on next page)

(continued from previous page)

```
    _futures_executor: futures.Executor = field(
        default=Factory(
            lambda self: self.create_futures_executor(),
            takes_self=True,
        ),
        alias="futures_executor",
    )

    @property
    def futures_executor(self) -> futures.Executor:
        self.__raise_deprecation_warning()
        return self._futures_executor

    @futures_executor.setter
    def futures_executor(self, value: futures.Executor) -> None:
        self.__raise_deprecation_warning()
        self._futures_executor = value

    def __raise_deprecation_warning(self) -> None:
        warnings.warn(
            "`FuturesExecutorMixin.futures_executor` is deprecated and will be␣
↪removed in a future release. Use `FuturesExecutorMixin.create_futures_executor`␣
↪instead.",
            DeprecationWarning,
            stacklevel=2,
        )
```

- `_futures_executor = field(default=Factory(lambda self: self.create_futures_executor(), takes_self=True), alias='futures_executor')` class-attribute instance-attribute
- `create_futures_executor = field(default=Factory(lambda: lambda: futures.ThreadPoolExecutor()))` class-attribute instance-attribute
- `futures_executor` property writable

__raise_deprecation_warning()

Source Code in griptape/mixins/futures_executor_mixin.py

```
def __raise_deprecation_warning(self) -> None:
    warnings.warn(
        "`FuturesExecutorMixin.futures_executor` is deprecated and will be removed in␣
↪a future release. Use `FuturesExecutorMixin.create_futures_executor` instead.",
        DeprecationWarning,
        stacklevel=2,
    )
```

rule_mixin

Bases: *SerializableMixin* “)

Source Code in griptape/mixins/rule_mixin.py

```
@define(slots=False)
class RuleMixin(SerializableMixin):
    DEFAULT_RULESET_NAME = "Default Ruleset"

    _rulesets: list[Ruleset] = field(
        factory=list,
        kw_only=True,
        alias="rulesets",
        metadata={"serializable": True, "serialization_key": "_rulesets",
↪"deserialization_key": "rulesets"},
    )
    rules: list[BaseRule] = field(
        factory=list, kw_only=True, metadata={"serializable": True, "deserialization_
↪key": "rules"}
    )
    _default_ruleset_name: str = field(default=Factory(lambda: RuleMixin.DEFAULT_
↪RULESET_NAME), kw_only=True)
    _default_ruleset_id: str = field(default=Factory(lambda: uuid.uuid4().hex), kw_
↪only=True)

    @property
    def rulesets(self) -> list[Ruleset]:
        rulesets = self._rulesets.copy()

        if self.rules:
            rulesets.append(Ruleset(id=self._default_ruleset_id, name=self._default_
↪ruleset_name, rules=self.rules))

        return rulesets
```

- `DEFAULT_RULESET_NAME = 'Default Ruleset'` class-attribute instance-attribute
- `_default_ruleset_id = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True)` class-attribute instance-attribute
- `_default_ruleset_name = field(default=Factory(lambda: RuleMixin.DEFAULT_RULESET_NAME), kw_only=True)` class-attribute instance-attribute
- `_rulesets = field(factory=list, kw_only=True, alias='rulesets', metadata={'serializable': True, 'serialization_key': '_rulesets', 'deserialization_key': 'rulesets'})` class-attribute instance-attribute
- `rules = field(factory=list, kw_only=True, metadata={'serializable': True, 'deserialization_key': 'rules'})` class-attribute instance-attribute
- `rulesets` property

runnable_mixin

- `T = TypeVar('T', bound='RunnableMixin')` module-attribute

Bases:

ABC, *Generic[T*

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">Optional[Callable[[T], None]]</code>	Optional callback that is called at the very beginning of the *run* method.
<code class="language-python">	<code class="language-python">Optional[Callable[[T], None]]</code>	Optional callback that is called at the very end of the *run* method.

Source Code in griptape/mixins/runnable_mixin.py

```
@define()
class RunnableMixin(ABC, Generic[T]):
    """Mixin for classes that can be "run".

    Implementing classes should pass themselves as the generic type to ensure that␣
↪the correct type is used in the callbacks.

    Attributes:
        on_before_run: Optional callback that is called at the very beginning of the␣
↪`run` method.
        on_after_run: Optional callback that is called at the very end of the `run`␣
```

(continues on next page)

(continued from previous page)

```
↪method.
    """

    on_before_run: Optional[Callable[[T], None]] = field(kw_only=True, default=None)
    on_after_run: Optional[Callable[[T], None]] = field(kw_only=True, default=None)

    def before_run(self, *args, **kwargs) -> Any:
        if self.on_before_run is not None:
            self.on_before_run(cast("T", self))

    @abstractmethod
    def run(self, *args, **kwargs) -> Any: ...

    def after_run(self, *args, **kwargs) -> Any:
        if self.on_after_run is not None:
            self.on_after_run(cast("T", self))
```

- `on_after_run = field(kw_only=True, default=None)` class-attribute instance-attribute
- `on_before_run = field(kw_only=True, default=None)` class-attribute instance-attribute

after_run(*args, **kwargs)

Source Code in griptape/mixins/runnable_mixin.py

```
def after_run(self, *args, **kwargs) -> Any:
    if self.on_after_run is not None:
        self.on_after_run(cast("T", self))
```

before_run(*args, **kwargs)

Source Code in griptape/mixins/runnable_mixin.py

```
def before_run(self, *args, **kwargs) -> Any:
    if self.on_before_run is not None:
        self.on_before_run(cast("T", self))
```

run(*args, **kwargs)abstractmethod

Source Code in griptape/mixins/runnable_mixin.py

```
@abstractmethod
def run(self, *args, **kwargs) -> Any: ...
```

serializable_mixin

- `T = TypeVar('T', bound='SerializableMixin')` module-attribute

Bases: *Generic[T*

Source Code in griptape/mixins/serializable_mixin.py

```
@define(slots=False)
class SerializableMixin(Generic[T]):
    type: str = field(
        default=Factory(lambda self: self.__class__.__name__, takes_self=True),
        kw_only=True,
        metadata={"serializable": True},
    )
    module_name: str = field(
        default=Factory(lambda self: self.__class__.__module__, takes_self=True),
        kw_only=True,
        metadata={"serializable": False},
    )

    @classmethod
    def get_schema(
        cls: type[T],
        subclass_name: Optional[str] = None,
        *,
        module_name: Optional[str] = None,
        types_overrides: Optional[dict[str, type]] = None,
        serializable_overrides: Optional[dict[str, bool]] = None,
    ) -> Schema:
        """Generates a Marshmallow schema for the class.

        Args:
            subclass_name: An optional subclass name. Required if the class is␣
↪abstract.
            module_name: An optional module name. Defaults to the classs module.
            types_overrides: An optional dictionary of field names to override type.
            serializable_overrides: An optional dictionary of field names to override␣
↪serializable status.
        """
        if ABC in cls.__bases__:
            if subclass_name is None:
                raise ValueError(f"Type field is required for abstract class: {cls.__
↪name__}")

            module_name = module_name or cls.__module__
            subclass_cls = cls._import_cls_rec(module_name, subclass_name)

            schema_class = BaseSchema.from_attrs_cls(
                subclass_cls, types_overrides=types_overrides, serializable_
↪overrides=serializable_overrides
            )
        else:
            schema_class = BaseSchema.from_attrs_cls(
                cls, types_overrides=types_overrides, serializable_
↪overrides=serializable_overrides
            )

        return schema_class()

    @classmethod
```

(continues on next page)

(continued from previous page)

```
    def from_dict(
        cls: type[T],
        data: dict,
        *,
        types_overrides: Optional[dict[str, type]] = None,
        serializable_overrides: Optional[dict[str, bool]] = None,
    ) -> T:
        return cast(
            "T",
            cls.get_schema(
                subclass_name=data.get("type"),
                module_name=data.get("module_name"),
                types_overrides=types_overrides,
                serializable_overrides=serializable_overrides,
            ).load(data),
        )

    @classmethod
    def from_json(
        cls: type[T],
        data: str,
        *,
        types_overrides: Optional[dict[str, type]] = None,
        serializable_overrides: Optional[dict[str, bool]] = None,
    ) -> T:
        return cls.from_dict(
            json.loads(data), types_overrides=types_overrides, serializable_
↪overrides=serializable_overrides
        )

    def __str__(self) -> str:
        return json.dumps(self.to_dict())

    def to_json(
        self,
        *,
        types_overrides: Optional[dict[str, type]] = None,
        serializable_overrides: Optional[dict[str, bool]] = None,
    ) -> str:
        return json.dumps(self.to_dict(types_overrides=types_overrides, serializable_
↪overrides=serializable_overrides))

    def to_dict(
        self,
        *,
        types_overrides: Optional[dict[str, type]] = None,
        serializable_overrides: Optional[dict[str, bool]] = None,
    ) -> dict:
        schema = BaseSchema.from_attrs_cls(
            self.__class__, types_overrides=types_overrides, serializable_
↪overrides=serializable_overrides
        )
```

(continues on next page)

(continued from previous page)

```
        return dict(schema().dump(self))

    @classmethod
    def _import_cls_rec(cls, module_name: str, class_name: str) -> type:
        """Imports a class given a module name and class name.

        Will recursively traverse up the modules path until it finds a
        package that it can import `class_name` from.

        Args:
            module_name: The module name.
            class_name: The class name.

        Returns:
            The imported class if found. Raises `ValueError` if not found.
        """
        try:
            module = import_module(module_name)
            test = getattr(module, class_name, None)
        except ModuleNotFoundError:
            test = None

        if test is None:
            module_dirs = module_name.split(".")[:-1]
            module_name = ".".join(module_dirs)

            if not len(module_dirs):
                raise ValueError(f"Unable to import class: {class_name}")
            return cls._import_cls_rec(module_name, class_name)
        return test
```

- `module_name = field(default=Factory(lambda self: self.__class__.__module__, takes_self=True), kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `type = field(default=Factory(lambda self: self.__class__.__name__, takes_self=True), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

str()

Source Code in griptape/mixins/serializable_mixin.py

```
def __str__(self) -> str:
    return json.dumps(self.to_dict())
```

_import_cls_rec(module_name, class_name)classmethod

Imports a class given a module name and class name.

Will recursively traverse up the module's path until it finds a package that it can import `class_name` from.

Parameters

Name	Type	Description	Default
module_name	*str*	The module name. 	*required*
class_name	*str*	The class name. 	*required*

Returns

Type	Description
<a href="serializable_mixin.mdx#griptape.mixins.serializable_mixin.SerializableMixin.type" title="type = field(default=Factory(lambda self: self.__class__.__name__, takes_self=True), kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.mixins.serializable_mixin.SerializableMixin.type)"><code class="language-python">ty	The imported class if found. Raises *ValueError* if not found.

Source Code in griptape/mixins/serializable_mixin.py

```
@classmethod
def _import_cls_rec(cls, module_name: str, class_name: str) -> type:
    """Imports a class given a module name and class name.

    Will recursively traverse up the modules path until it finds a
    package that it can import `class_name` from.

    Args:
        module_name: The module name.
        class_name: The class name.

    Returns:
        The imported class if found. Raises `ValueError` if not found.
    """
    try:
        module = import_module(module_name)
        test = getattr(module, class_name, None)
    except ModuleNotFoundError:
```

(continues on next page)

(continued from previous page)

```
        test = None

    if test is None:
        module_dirs = module_name.split(".")[:-1]
        module_name = ".".join(module_dirs)

        if not len(module_dirs):
            raise ValueError(f"Unable to import class: {class_name}")
        return cls._import_cls_rec(module_name, class_name)
    return test
```

from_dict(data, *, types_overrides=None, serializable_overrides=None)classmethod

Source Code in griptape/mixins/serializable_mixin.py

```
@classmethod
def from_dict(
    cls: type[T],
    data: dict,
    *,
    types_overrides: Optional[dict[str, type]] = None,
    serializable_overrides: Optional[dict[str, bool]] = None,
) -> T:
    return cast(
        "T",
        cls.get_schema(
            subclass_name=data.get("type"),
            module_name=data.get("module_name"),
            types_overrides=types_overrides,
            serializable_overrides=serializable_overrides,
        ).load(data),
    )
```

from_json(data, *, types_overrides=None, serializable_overrides=None)classmethod

Source Code in griptape/mixins/serializable_mixin.py

```
@classmethod
def from_json(
    cls: type[T],
    data: str,
    *,
    types_overrides: Optional[dict[str, type]] = None,
    serializable_overrides: Optional[dict[str, bool]] = None,
) -> T:
    return cls.from_dict(
        json.loads(data), types_overrides=types_overrides, serializable_
↪overrides=serializable_overrides
    )
```

get_schema(subclass_name=None, *, module_name=None, types_overrides=None, serializable_overrides=None)classmethod

Generates a Marshmallow schema for the class.

Parameters

Name	Type	Description	Default
subclass_name	*Optional[str]*	An optional subclass name. Required if the class is abstract. 	*None*
module_name	*Optional[str]*	An optional module name. Defaults to the class's module. 	*None*
types_overrides	<co class="language-pythc type]]</code>	An optional dictionary of field names to override type. 	*None*
serializable_overrides	*Optional[dict[str, bool]]*	An optional dictionary of field names to override serializable status. 	*None*

Source Code in griptape/mixins/serializable_mixin.py

```
@classmethod
def get_schema(
    cls: type[T],
    subclass_name: Optional[str] = None,
    *,
    module_name: Optional[str] = None,
    types_overrides: Optional[dict[str, type]] = None,
    serializable_overrides: Optional[dict[str, bool]] = None,
```

(continues on next page)

(continued from previous page)

```
) -> Schema:
    """Generates a Marshmallow schema for the class.

    Args:
        subclass_name: An optional subclass name. Required if the class is abstract.
        module_name: An optional module name. Defaults to the classs module.
        types_overrides: An optional dictionary of field names to override type.
        serializable_overrides: An optional dictionary of field names to override␣
↪serializable status.
    """
    if ABC in cls.__bases__:
        if subclass_name is None:
            raise ValueError(f"Type field is required for abstract class: {cls.__name_
↪_}")

        module_name = module_name or cls.__module__
        subclass_cls = cls._import_cls_rec(module_name, subclass_name)

        schema_class = BaseSchema.from_attrs_cls(
            subclass_cls, types_overrides=types_overrides, serializable_
↪overrides=serializable_overrides
        )
    else:
        schema_class = BaseSchema.from_attrs_cls(
            cls, types_overrides=types_overrides, serializable_overrides=serializable_
↪overrides
        )

    return schema_class()
```

to_dict(*, types_overrides=None, serializable_overrides=None)

Source Code in griptape/mixins/serializable_mixin.py

```
def to_dict(
    self,
    *,
    types_overrides: Optional[dict[str, type]] = None,
    serializable_overrides: Optional[dict[str, bool]] = None,
) -> dict:
    schema = BaseSchema.from_attrs_cls(
        self.__class__, types_overrides=types_overrides, serializable_
↪overrides=serializable_overrides
    )

    return dict(schema().dump(self))
```

to_json(*, types_overrides=None, serializable_overrides=None)

Source Code in griptape/mixins/serializable_mixin.py

```
def to_json(
    self,
    *,
    types_overrides: Optional[dict[str, type]] = None,
    serializable_overrides: Optional[dict[str, bool]] = None,
) -> str:
    return json.dumps(self.to_dict(types_overrides=types_overrides, serializable_
↪overrides=serializable_overrides))
```

_default(_self, obj)

Source Code in griptape/mixins/serializable_mixin.py

```
def _default(_self: Any, obj: Any) -> Any:
    """Fallback method for JSONEncoder to handle custom serialization."""
    return getattr(obj.__class__, "to_dict", getattr(_default, "default"))(obj)
```

singleton_mixin

Source Code in griptape/mixins/singleton_mixin.py

```
class SingletonMixin:
    _instance = None

    def __new__(cls, *args, **kwargs) -> Self:
        if not cls._instance:
            cls._instance = super().__new__(cls, *args, **kwargs)
        return cls._instance
```

- `_instance = None` class-attribute instance-attribute

new(*args, **kwargs)

Source Code in griptape/mixins/singleton_mixin.py

```
def __new__(cls, *args, **kwargs) -> Self:
    if not cls._instance:
        cls._instance = super().__new__(cls, *args, **kwargs)
    return cls._instance
```

11.5.14 observability

- `__all__ = ['Observability']` module-attribute

Source Code in griptape/observability/observability.py

```
@define
class Observability:
    observability_driver: BaseObservabilityDriver = field(kw_only=True)
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def get_global_driver() -> Optional[BaseObservabilityDriver]:
        return _global_observability_driver

    @staticmethod
    def set_global_driver(driver: Optional[BaseObservabilityDriver]) -> None:
        global _global_observability_driver  # noqa: PLW0603
        _global_observability_driver = driver

    @staticmethod
    def observe(call: Observable.Call) -> Any:
        driver = Observability.get_global_driver() or _no_op_observability_driver
        return driver.observe(call)

    @staticmethod
    def get_span_id() -> Optional[str]:
        driver = Observability.get_global_driver() or _no_op_observability_driver
        return driver.get_span_id()

    def __enter__(self) -> None:
        if Observability.get_global_driver() is not None:
            raise ValueError("Observability driver already set.")
        Observability.set_global_driver(self.observability_driver)
        self.observability_driver.__enter__()

    def __exit__(
        self,
        exc_type: Optional[type[BaseException]],
        exc_value: Optional[BaseException],
        exc_traceback: Optional[TracebackType],
    ) -> bool:
        Observability.set_global_driver(None)
        self.observability_driver.__exit__(exc_type, exc_value, exc_traceback)
        return False
```

- `observability_driver = field(kw_only=True)` class-attribute instance-attribute

enter()

Source Code in griptape/observability/observability.py

```
def __enter__(self) -> None:
    if Observability.get_global_driver() is not None:
        raise ValueError("Observability driver already set.")
    Observability.set_global_driver(self.observability_driver)
    self.observability_driver.__enter__()
```

exit(exc_type, exc_value, exc_traceback)

Source Code in griptape/observability/observability.py

```
def __exit__(
    self,
    exc_type: Optional[type[BaseException]],
    exc_value: Optional[BaseException],
    exc_traceback: Optional[TracebackType],
) -> bool:
    Observability.set_global_driver(None)
    self.observability_driver.__exit__(exc_type, exc_value, exc_traceback)
    return False
```

get_global_driver()staticmethod

Source Code in griptape/observability/observability.py

```
@staticmethod
def get_global_driver() -> Optional[BaseObservabilityDriver]:
    return _global_observability_driver
```

get_span_id()staticmethod

Source Code in griptape/observability/observability.py

```
@staticmethod
def get_span_id() -> Optional[str]:
    driver = Observability.get_global_driver() or _no_op_observability_driver
    return driver.get_span_id()
```

observe(call)staticmethod

Source Code in griptape/observability/observability.py

```
@staticmethod
def observe(call: Observable.Call) -> Any:
    driver = Observability.get_global_driver() or _no_op_observability_driver
    return driver.observe(call)
```

set_global_driver(driver)staticmethod

Source Code in griptape/observability/observability.py

```
@staticmethod
def set_global_driver(driver: Optional[BaseObservabilityDriver]) -> None:
    global _global_observability_driver  # noqa: PLW0603
    _global_observability_driver = driver
```

observability

- `_global_observability_driver = None` module-attribute
- `_no_op_observability_driver = NoOpObservabilityDriver()` module-attribute

Source Code in griptape/observability/observability.py

```
@define
class Observability:
    observability_driver: BaseObservabilityDriver = field(kw_only=True)

    @staticmethod
    def get_global_driver() -> Optional[BaseObservabilityDriver]:
        return _global_observability_driver

    @staticmethod
    def set_global_driver(driver: Optional[BaseObservabilityDriver]) -> None:
        global _global_observability_driver  # noqa: PLW0603
        _global_observability_driver = driver

    @staticmethod
    def observe(call: Observable.Call) -> Any:
        driver = Observability.get_global_driver() or _no_op_observability_driver
        return driver.observe(call)

    @staticmethod
    def get_span_id() -> Optional[str]:
        driver = Observability.get_global_driver() or _no_op_observability_driver
        return driver.get_span_id()

    def __enter__(self) -> None:
        if Observability.get_global_driver() is not None:
            raise ValueError("Observability driver already set.")
        Observability.set_global_driver(self.observability_driver)
        self.observability_driver.__enter__()

    def __exit__(
        self,
        exc_type: Optional[type[BaseException]],
        exc_value: Optional[BaseException],
        exc_traceback: Optional[TracebackType],
    ) -> bool:
        Observability.set_global_driver(None)
        self.observability_driver.__exit__(exc_type, exc_value, exc_traceback)
        return False
```

- `observability_driver = field(kw_only=True)` class-attribute instance-attribute

enter()

Source Code in griptape/observability/observability.py

```
def __enter__(self) -> None:
    if Observability.get_global_driver() is not None:
        raise ValueError("Observability driver already set.")
    Observability.set_global_driver(self.observability_driver)
    self.observability_driver.__enter__()
```

exit(exc_type, exc_value, exc_traceback)

Source Code in griptape/observability/observability.py

```
def __exit__(
    self,
    exc_type: Optional[type[BaseException]],
    exc_value: Optional[BaseException],
    exc_traceback: Optional[TracebackType],
) -> bool:
    Observability.set_global_driver(None)
    self.observability_driver.__exit__(exc_type, exc_value, exc_traceback)
    return False
```

get_global_driver()staticmethod

Source Code in griptape/observability/observability.py

```
@staticmethod
def get_global_driver() -> Optional[BaseObservabilityDriver]:
    return _global_observability_driver
```

get_span_id()staticmethod

Source Code in griptape/observability/observability.py

```
@staticmethod
def get_span_id() -> Optional[str]:
    driver = Observability.get_global_driver() or _no_op_observability_driver
    return driver.get_span_id()
```

observe(call)staticmethod

Source Code in griptape/observability/observability.py

```
@staticmethod
def observe(call: Observable.Call) -> Any:
    driver = Observability.get_global_driver() or _no_op_observability_driver
    return driver.observe(call)
```

set_global_driver(driver)staticmethod

Source Code in griptape/observability/observability.py

```
@staticmethod
def set_global_driver(driver: Optional[BaseObservabilityDriver]) -> None:
    global _global_observability_driver  # noqa: PLW0603
    _global_observability_driver = driver
```

11.5.15 rules

- `__all__ = ['BaseRule', 'JsonSchemaRule', 'Rule', 'Ruleset']` module-attribute

Bases:

ABC, *SerializableMixin* “)

Source Code in griptape/rules/base_rule.py

```
@define()
class BaseRule(ABC, SerializableMixin):
    value: Any = field(metadata={"serializable": True})
    meta: dict[str, Any] = field(factory=dict, kw_only=True)

    def __str__(self) -> str:
        return self.to_text()

    @abstractmethod
    def to_text(self) -> str: ...
```

- `meta = field(factory=dict, kw_only=True)` class-attribute instance-attribute
- `value = field(metadata={'serializable': True})` class-attribute instance-attribute

str()

Source Code in griptape/rules/base_rule.py

```
def __str__(self) -> str:
    return self.to_text()
```

to_text()abstractmethod

Source Code in griptape/rules/base_rule.py

```
@abstractmethod
def to_text(self) -> str: ...
```

JsonSchemaRule

Bases: *BaseRulesetDriver* “)

Source Code in griptape/rules/json_schema_rule.py

```
@define()
class JsonSchemaRule(BaseRule):
    value: dict = field(metadata={"serializable": True})
    generate_template: J2 = field(default=Factory(lambda: J2("rules/json_schema.j2")))

    def to_text(self) -> str:
        return self.generate_template.render(json_schema=json.dumps(self.value))
```

- `generate_template = field(default=Factory(lambda: J2('rules/json_schema.j2')))` class-attribute instance-attribute
- `value = field(metadata={'serializable': True})` class-attribute instance-attribute

to_text()

Source Code in griptape/rules/json_schema_rule.py

```
def to_text(self) -> str:
    return self.generate_template.render(json_schema=json.dumps(self.value))
```

Rule

Bases: *BaseRulesetDriver* “)

Source Code in griptape/rules/rule.py

```
@define()
class Rule(BaseRule):
    value: str = field(metadata={"serializable": True})

    def to_text(self) -> str:
        return self.value
```

- `value = field(metadata={'serializable': True})` class-attribute instance-attribute

to_text()

Source Code in griptape/rules/rule.py

```
def to_text(self) -> str:
    return self.value
```

Ruleset

Bases: *SerializableMixin* “)

Source Code in griptape/rules/ruleset.py

```
@define
class Ruleset(SerializableMixin):
    id: str = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata=
↪{"serializable": True})
    name: str = field(
        default=Factory(lambda self: self.id, takes_self=True),
        metadata={"serializable": True},
    )
    ruleset_driver: BaseRulesetDriver = field(
        default=Factory(lambda: Defaults.drivers_config.ruleset_driver), kw_only=True
    )
    meta: dict[str, Any] = field(factory=dict, kw_only=True, metadata={"serializable
↪": True})
    rules: Sequence[BaseRule] = field(factory=list, metadata={"serializable": True})

    def __attrs_post_init__(self) -> None:
        rules, meta = self.ruleset_driver.load(self.name)
        self.rules = [*rules, *self.rules]
        self.meta = {**meta, **self.meta}
```

- `id = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `meta = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `name = field(default=Factory(lambda self: self.id, takes_self=True), metadata={'serializable': True})` class-attribute instance-attribute
- `rules = field(factory=list, metadata={'serializable': True})` class-attribute instance-attribute
- `ruleset_driver = field(default=Factory(lambda: Defaults.drivers_config.ruleset_driver), kw_only=True)` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/rules/ruleset.py

```
def __attrs_post_init__(self) -> None:
    rules, meta = self.ruleset_driver.load(self.name)
    self.rules = [*rules, *self.rules]
    self.meta = {**meta, **self.meta}
```

base_rule

Bases:

ABC, *SerializableMixin* “)

Source Code in griptape/rules/base_rule.py

```
@define()
class BaseRule(ABC, SerializableMixin):
    value: Any = field(metadata={"serializable": True})
    meta: dict[str, Any] = field(factory=dict, kw_only=True)

    def __str__(self) -> str:
        return self.to_text()

    @abstractmethod
    def to_text(self) -> str: ...
```

- `meta = field(factory=dict, kw_only=True)` class-attribute instance-attribute
- `value = field(metadata={'serializable': True})` class-attribute instance-attribute

str()

Source Code in griptape/rules/base_rule.py

```
def __str__(self) -> str:
    return self.to_text()
```

to_text()abstractmethod

Source Code in griptape/rules/base_rule.py

```
@abstractmethod
def to_text(self) -> str: ...
```

json_schema_rule

Bases: *BaseRulesetDriver* “)

Source Code in griptape/rules/json_schema_rule.py

```
@define()
class JsonSchemaRule(BaseRule):
    value: dict = field(metadata={"serializable": True})
    generate_template: J2 = field(default=Factory(lambda: J2("rules/json_schema.j2")))

    def to_text(self) -> str:
        return self.generate_template.render(json_schema=json.dumps(self.value))
```

- `generate_template = field(default=Factory(lambda: J2('rules/json_schema.j2')))` class-attribute instance-attribute
- `value = field(metadata={'serializable': True})` class-attribute instance-attribute

to_text()

Source Code in griptape/rules/json_schema_rule.py

```
def to_text(self) -> str:
    return self.generate_template.render(json_schema=json.dumps(self.value))
```

rule

Bases: *BaseRulesetDriver* “)

Source Code in griptape/rules/rule.py

```
@define()
class Rule(BaseRule):
    value: str = field(metadata={"serializable": True})

    def to_text(self) -> str:
        return self.value
```

- `value = field(metadata={'serializable': True})` class-attribute instance-attribute

to_text()

Source Code in griptape/rules/rule.py

```
def to_text(self) -> str:
    return self.value
```

ruleset

Bases: *SerializableMixin* “)

Source Code in griptape/rules/ruleset.py

```
@define
class Ruleset(SerializableMixin):
    id: str = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata=
↪{"serializable": True})
    name: str = field(
        default=Factory(lambda self: self.id, takes_self=True),
        metadata={"serializable": True},
    )
    ruleset_driver: BaseRulesetDriver = field(
        default=Factory(lambda: Defaults.drivers_config.ruleset_driver), kw_only=True
    )
    meta: dict[str, Any] = field(factory=dict, kw_only=True, metadata={"serializable
↪": True})
    rules: Sequence[BaseRule] = field(factory=list, metadata={"serializable": True})

    def __attrs_post_init__(self) -> None:
        rules, meta = self.ruleset_driver.load(self.name)
        self.rules = [*rules, *self.rules]
        self.meta = {**meta, **self.meta}
```

- `id = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `meta = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `name = field(default=Factory(lambda self: self.id, takes_self=True), metadata={'serializable': True})` class-attribute instance-attribute
- `rules = field(factory=list, metadata={'serializable': True})` class-attribute instance-attribute
- `ruleset_driver = field(default=Factory(lambda: Defaults.drivers_config.ruleset_driver), kw_only=True)` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/rules/ruleset.py

```
def __attrs_post_init__(self) -> None:
    rules, meta = self.ruleset_driver.load(self.name)
    self.rules = [*rules, *self.rules]
    self.meta = {**meta, **self.meta}
```

11.5.16 schemas

- `__all__ = ['BaseSchema', 'Bytes', 'PolymorphicSchema', 'PydanticModel', 'Union']` module-attribute

Bases:

Schema

Source Code in griptape/schemas/base_schema.py

```
class BaseSchema(Schema):
    class Meta:
        unknown = INCLUDE

    DATACLASS_TYPE_MAPPING = {
        **Schema.TYPE_MAPPING,
        dict: fields.Dict,
        bytes: Bytes,
        Any: fields.Raw,
        BaseModel: PydanticModel,
    }

    @classmethod
    def from_attrs_cls(
        cls,
        attrs_cls: type,
        *,
        types_overrides: Optional[dict[str, type]] = None,
        serializable_overrides: Optional[dict[str, bool]] = None,
    ) -> type:
        """Generate a Schema from an attrs class.

        Args:
            attrs_cls: An attrs class.
            types_overrides: A dictionary of types to override when resolving types.
            serializable_overrides: A dictionary of field names to whether they are␣
↪serializable.
        """
        from marshmallow import post_load

        if serializable_overrides is None:
            serializable_overrides = {}

        class SubSchema(cls):
            @post_load
            def make_obj(self, data: Any, **kwargs) -> Any:
                # Map the serialized keys to their correct deserialization keys
                fields = attrs.fields_dict(attrs_cls)
                for key in list(data):
                    if key in fields:
                        field = fields[key]
                        if field.metadata.get("deserialization_key"):
                            data[field.metadata["deserialization_key"]] = data.
↪pop(key)

                return attrs_cls(**data)

        cls._resolve_types(attrs_cls, types_override=types_overrides)
        fields = {}
        for field in attrs.fields(attrs_cls):
            field_key = field.alias or field.name
```

(continues on next page)

(continued from previous page)

```
            if serializable_overrides.get(field_key, field.metadata.get("serializable
↪", False)):
                fields[field_key] = cls._get_field_for_type(
                    field.type,
                    serialization_key=field.metadata.get("serialization_key"),
                    types_overrides=types_overrides,
                )
        return SubSchema.from_dict(fields, name=f"{attrs_cls.__name__}Schema")

    @classmethod
    def _get_field_for_type(
        cls,
        field_type: type,
        serialization_key: Optional[str] = None,
        types_overrides: Optional[dict[str, type]] = None,
    ) -> fields.Field | fields.Nested:
        """Generate a marshmallow Field instance from a Python type.

        Args:
            field_type: A field type.
            serialization_key: The key to pull the data from before serializing.
            types_overrides: A dictionary of types to override when resolving types.
        """
        from griptape.schemas.polymorphic_schema import PolymorphicSchema

        field_class, args, optional = cls._get_field_type_info(field_type)

        if field_class is None:
            return fields.Constant(None, allow_none=True)

        # Resolve TypeVars to their bound type
        if isinstance(field_class, TypeVar):
            field_class = field_class.__bound__
            if field_class is None:
                return fields.Raw(allow_none=optional, attribute=serialization_key)

        if cls._is_union(field_type):
            return cls._handle_union(
                field_type,
                optional=optional,
                serialization_key=serialization_key,
            )
        if attrs.has(field_class):
            if ABC in field_class.__bases__:
                schema = PolymorphicSchema(field_class, types_overrides=types_
↪overrides)
            else:
                schema = cls.from_attrs_cls(field_class, types_overrides=types_
↪overrides)
            return fields.Nested(schema, allow_none=optional, attribute=serialization_
↪key)
        if cls._is_enum(field_type):
```

(continues on next page)

(continued from previous page)

```
            return fields.String(allow_none=optional, attribute=serialization_key)
        if cls._is_list_sequence(field_class):
            if args:
                return cls._handle_list(
                    args[0],
                    optional=optional,
                    serialization_key=serialization_key,
                )
            raise ValueError(f"Missing type for list field: {field_type}")
        field_class = cls.DATACLASS_TYPE_MAPPING.get(field_class, fields.Raw)

        return field_class(allow_none=optional, attribute=serialization_key)

    @classmethod
    def _handle_list(
        cls,
        list_type: type,
        *,
        optional: bool,
        serialization_key: Optional[str] = None,
    ) -> fields.Field:
        """Handle List Fields, including Union Types.

        Args:
            list_type: The List type to handle.
            optional: Whether the List can be none.
            serialization_key: The key to pull the data from before serializing.

        Returns:
            A marshmallow List field.
        """
        if cls._is_union(list_type):
            instance = cls._handle_union(
                list_type,
                optional=optional,
                serialization_key=serialization_key,
            )
        else:
            instance = cls._get_field_for_type(list_type, serialization_
↪key=serialization_key)
        return fields.List(cls_or_instance=instance, allow_none=optional,␣
↪attribute=serialization_key)

    @classmethod
    def _handle_union(
        cls,
        union_type: type,
        *,
        optional: bool,
        serialization_key: Optional[str] = None,
    ) -> fields.Field:
        """Handle Union Fields, including Unions with List Types.
```

(continues on next page)

(continued from previous page)

```
        Args:
            union_type: The Union Type to handle.
            optional: Whether the Union can be None.
            serialization_key: The key to pull the data from before serializing.

        Returns:
            A marshmallow Union field.
        """
        candidate_fields = [cls._get_field_for_type(arg) for arg in get_args(union_
↪type) if arg is not type(None)]
        optional_args = [arg is None for arg in get_args(union_type)]
        if optional_args:
            optional = True
        if not candidate_fields:
            raise ValueError(f"Unsupported UnionType field: {union_type}")

        return UnionField(fields=candidate_fields, allow_none=optional,␣
↪attribute=serialization_key)

    @classmethod
    def _get_field_type_info(cls, field_type: type) -> tuple[type, tuple[type, ...],␣
↪bool]:
        """Get information about a field type.

        Args:
            field_type: A field type.
        """
        origin = get_origin(field_type) or field_type
        args = get_args(field_type)
        optional = False

        if origin is Union:
            origin = args[0]
            if len(args) > 1 and args[1] is type(None):
                optional = True

            origin, args, _ = cls._get_field_type_info(origin)
        elif origin is Literal:
            origin = type(args[0])
            args = ()

        return origin, args, optional

    @classmethod
    def _resolve_types(cls, attrs_cls: type, types_override: Optional[dict[str,␣
↪type]] = None) -> None:
        """Resolve types in an attrs class.

        Args:
            attrs_cls: An attrs class.
            types_override: A dictionary of types to override.
```

(continues on next page)

(continued from previous page)

```
        """
        from collections.abc import Sequence
        from typing import Any

        from pydantic import BaseModel
        from schema import Schema

        from griptape.artifacts import (
            ActionArtifact,
            AudioArtifact,
            BaseArtifact,
            BlobArtifact,
            BooleanArtifact,
            ErrorArtifact,
            GenericArtifact,
            ImageArtifact,
            ImageUrlArtifact,
            InfoArtifact,
            JsonArtifact,
            ListArtifact,
            TextArtifact,
        )
        from griptape.common import (
            BaseDeltaMessageContent,
            BaseMessageContent,
            Message,
            PromptStack,
            Reference,
            ToolAction,
        )
        from griptape.drivers.assistant import BaseAssistantDriver
        from griptape.drivers.audio_transcription import BaseAudioTranscriptionDriver
        from griptape.drivers.embedding import BaseEmbeddingDriver
        from griptape.drivers.file_manager import BaseFileManagerDriver
        from griptape.drivers.image_generation import BaseImageGenerationDriver,␣
↪BaseMultiModelImageGenerationDriver
        from griptape.drivers.image_generation_model import␣
↪BaseImageGenerationModelDriver
        from griptape.drivers.memory.conversation import BaseConversationMemoryDriver
        from griptape.drivers.observability import BaseObservabilityDriver
        from griptape.drivers.prompt import BasePromptDriver
        from griptape.drivers.prompt.base_prompt_driver import␣
↪StructuredOutputStrategy
        from griptape.drivers.ruleset import BaseRulesetDriver
        from griptape.drivers.sql import BaseSqlDriver
        from griptape.drivers.structure_run import BaseStructureRunDriver
        from griptape.drivers.text_to_speech import BaseTextToSpeechDriver
        from griptape.drivers.vector import BaseVectorStoreDriver
        from griptape.drivers.web_scraper import BaseWebScraperDriver
        from griptape.drivers.web_search import BaseWebSearchDriver
        from griptape.engines.rag import RagContext
        from griptape.events import EventListener
```

(continues on next page)

(continued from previous page)

```
        from griptape.memory import TaskMemory
        from griptape.memory.meta import BaseMetaEntry
        from griptape.memory.structure import BaseConversationMemory, Run
        from griptape.memory.task.storage import BaseArtifactStorage
        from griptape.rules.base_rule import BaseRule
        from griptape.rules.ruleset import Ruleset
        from griptape.structures import Structure
        from griptape.tasks import BaseTask
        from griptape.tokenizers import BaseTokenizer
        from griptape.tools import BaseTool
        from griptape.utils import import_optional_dependency, is_dependency_installed

        if types_override is None:
            types_override = {}

        attrs.resolve_types(
            attrs_cls,
            localns={
                "Any": Any,
                "BasePromptDriver": BasePromptDriver,
                "BaseEmbeddingDriver": BaseEmbeddingDriver,
                "BaseVectorStoreDriver": BaseVectorStoreDriver,
                "BaseTextToSpeechDriver": BaseTextToSpeechDriver,
                "BaseAudioTranscriptionDriver": BaseAudioTranscriptionDriver,
                "BaseConversationMemoryDriver": BaseConversationMemoryDriver,
                "BaseRulesetDriver": BaseRulesetDriver,
                "BaseImageGenerationDriver": BaseImageGenerationDriver,
                "BaseMultiModelImageGenerationDriver":␣
↪BaseMultiModelImageGenerationDriver,
                "BaseImageGenerationModelDriver": BaseImageGenerationModelDriver,
                "BaseWebSearchDriver": BaseWebSearchDriver,
                "BaseWebScraperDriver": BaseWebScraperDriver,
                "BaseFileManagerDriver": BaseFileManagerDriver,
                "BaseSqlDriver": BaseSqlDriver,
                "BaseObservabilityDriver": BaseObservabilityDriver,
                "BaseAssistantDriver": BaseAssistantDriver,
                "BaseStructureRunDriver": BaseStructureRunDriver,
                "BaseArtifact": BaseArtifact,
                "BaseMetaEntry": BaseMetaEntry,
                "PromptStack": PromptStack,
                "EventListener": EventListener,
                "BaseMessageContent": BaseMessageContent,
                "BaseDeltaMessageContent": BaseDeltaMessageContent,
                "BaseTool": BaseTool,
                "BaseTask": BaseTask,
                "TextArtifact": TextArtifact,
                "ImageArtifact": ImageArtifact,
                "ImageUrlArtifact": ImageUrlArtifact,
                "ErrorArtifact": ErrorArtifact,
                "InfoArtifact": InfoArtifact,
                "JsonArtifact": JsonArtifact,
                "BlobArtifact": BlobArtifact,
```

(continues on next page)

(continued from previous page)

```
                "BooleanArtifact": BooleanArtifact,
                "ListArtifact": ListArtifact,
                "AudioArtifact": AudioArtifact,
                "ActionArtifact": ActionArtifact,
                "GenericArtifact": GenericArtifact,
                "Usage": Message.Usage,
                "Structure": Structure,
                "BaseTokenizer": BaseTokenizer,
                "ToolAction": ToolAction,
                "Reference": Reference,
                "Run": Run,
                "Sequence": Sequence,
                "TaskMemory": TaskMemory,
                "State": BaseTask.State,
                "BaseConversationMemory": BaseConversationMemory,
                "BaseArtifactStorage": BaseArtifactStorage,
                "BaseRule": BaseRule,
                "Ruleset": Ruleset,
                "StructuredOutputStrategy": StructuredOutputStrategy,
                "RagContext": RagContext,
                # Third party modules
                "Client": import_optional_dependency("cohere").Client if is_
↪dependency_installed("cohere") else Any,
                "ClientV2": import_optional_dependency("cohere").ClientV2 if is_
↪dependency_installed("cohere") else Any,
                "GenerativeModel": import_optional_dependency("google.generativeai").
↪GenerativeModel
                if is_dependency_installed("google.generativeai")
                else Any,
                "boto3": import_optional_dependency("boto3") if is_dependency_
↪installed("boto3") else Any,
                "Anthropic": import_optional_dependency("anthropic").Anthropic
                if is_dependency_installed("anthropic")
                else Any,
                "BedrockRuntimeClient": import_optional_dependency("mypy_boto3_
↪bedrock_runtime").BedrockRuntimeClient
                if is_dependency_installed("mypy_boto3_bedrock_runtime")
                else Any,
                "voyageai": import_optional_dependency("voyageai") if is_dependency_
↪installed("voyageai") else Any,
                "Schema": Schema,
                "BaseModel": BaseModel,
                **types_override,
            },
        )

    @classmethod
    def _is_list_sequence(cls, field_type: type | _SpecialForm) -> bool:
        if isinstance(field_type, type):
            if issubclass(field_type, str) or issubclass(field_type, bytes) or␣
↪issubclass(field_type, tuple):
                return False
```

(continues on next page)

(continued from previous page)

```
            return issubclass(field_type, Sequence)
        return False

    @classmethod
    def _is_union(cls, field_type: type) -> bool:
        return field_type is Union or get_origin(field_type) is Union

    @classmethod
    def _is_enum(cls, field_type: type) -> bool:
        return isinstance(field_type, type) and issubclass(field_type, Enum)
```

- `DATACLASS_TYPE_MAPPING = {None: Schema.TYPE_MAPPING, dict: fields.Dict, bytes: Bytes, Any: fields.Raw, BaseModel: PydanticModel}` class-attribute instance-attribute

Meta

Source Code in griptape/schemas/base_schema.py

```
class Meta:
    unknown = INCLUDE
```

- `unknown = INCLUDE` class-attribute instance-attribute

_get_field_for_type(field_type, serialization_key=None, types_overrides=None)classmethod

Generate a marshmallow Field instance from a Python type.

Parameters

Name	Type	Description	Default
field_type	*type*	A field type. 	*required*
serialization_key	*Optional[str]*	The key to pull the data from before serializing. 	*None*
types_overrides	*Optional[dict[str, type]]*	A dictionary of types to override when resolving types. 	*None*

Source Code in griptape/schemas/base_schema.py

```
@classmethod
def _get_field_for_type(
    cls,
    field_type: type,
    serialization_key: Optional[str] = None,
    types_overrides: Optional[dict[str, type]] = None,
) -> fields.Field | fields.Nested:
    """Generate a marshmallow Field instance from a Python type.

    Args:
        field_type: A field type.
```

(continues on next page)

(continued from previous page)

```
        serialization_key: The key to pull the data from before serializing.
        types_overrides: A dictionary of types to override when resolving types.
    """
    from griptape.schemas.polymorphic_schema import PolymorphicSchema

    field_class, args, optional = cls._get_field_type_info(field_type)

    if field_class is None:
        return fields.Constant(None, allow_none=True)

    # Resolve TypeVars to their bound type
    if isinstance(field_class, TypeVar):
        field_class = field_class.__bound__
        if field_class is None:
            return fields.Raw(allow_none=optional, attribute=serialization_key)

    if cls._is_union(field_type):
        return cls._handle_union(
            field_type,
            optional=optional,
            serialization_key=serialization_key,
        )
    if attrs.has(field_class):
        if ABC in field_class.__bases__:
            schema = PolymorphicSchema(field_class, types_overrides=types_overrides)
        else:
            schema = cls.from_attrs_cls(field_class, types_overrides=types_overrides)
        return fields.Nested(schema, allow_none=optional, attribute=serialization_key)
    if cls._is_enum(field_type):
        return fields.String(allow_none=optional, attribute=serialization_key)
    if cls._is_list_sequence(field_class):
        if args:
            return cls._handle_list(
                args[0],
                optional=optional,
                serialization_key=serialization_key,
            )
        raise ValueError(f"Missing type for list field: {field_type}")
    field_class = cls.DATACLASS_TYPE_MAPPING.get(field_class, fields.Raw)

    return field_class(allow_none=optional, attribute=serialization_key)
```

_get_field_type_info(field_type)classmethod

Get information about a field type.

Parameters

Name	Type	Description	Default
field_type	*type*	A field type. 	*required*

Source Code in griptape/schemas/base_schema.py

```
@classmethod
def _get_field_type_info(cls, field_type: type) -> tuple[type, tuple[type, ...],␣
↪bool]:
    """Get information about a field type.

    Args:
        field_type: A field type.
    """
    origin = get_origin(field_type) or field_type
    args = get_args(field_type)
    optional = False

    if origin is Union:
        origin = args[0]
        if len(args) > 1 and args[1] is type(None):
            optional = True

        origin, args, _ = cls._get_field_type_info(origin)
    elif origin is Literal:
        origin = type(args[0])
        args = ()

    return origin, args, optional
```

_handle_list(list_type, *, optional, serialization_key=None)classmethod

Handle List Fields, including Union Types.

Parameters

Name	Type	Description	Default
list_type	*type*	The List type to handle. 	*required*
optional	*bool*	Whether the List can be none. 	*required*
serialization_key	*Optional[str]*	The key to pull the data from before serializing. 	*None*

Returns

Type	Description
Field	A marshmallow List field.

Source Code in griptape/schemas/base_schema.py

```
@classmethod
def _handle_list(
    cls,
    list_type: type,
    *,
    optional: bool,
    serialization_key: Optional[str] = None,
) -> fields.Field:
    """Handle List Fields, including Union Types.

    Args:
        list_type: The List type to handle.
        optional: Whether the List can be none.
        serialization_key: The key to pull the data from before serializing.

    Returns:
        A marshmallow List field.
    """
    if cls._is_union(list_type):
        instance = cls._handle_union(
            list_type,
            optional=optional,
            serialization_key=serialization_key,
        )
    else:
        instance = cls._get_field_for_type(list_type, serialization_key=serialization_
↪key)
    return fields.List(cls_or_instance=instance, allow_none=optional,␣
↪attribute=serialization_key)
```

_handle_union(union_type, *, optional, serialization_key=None)classmethod

Handle Union Fields, including Unions with List Types.

Parameters

Name	Type	Description	Default
union_type	*type*	The Union Type to handle. 	*required*
optional	*bool*	Whether the Union can be None. 	*required*
serialization_key	*Optional[str]*	The key to pull the data from before serializing. 	*None*

Returns

Type	Description
Field	A marshmallow Union field.

Source Code in griptape/schemas/base_schema.py

```
@classmethod
def _handle_union(
    cls,
    union_type: type,
    *,
    optional: bool,
    serialization_key: Optional[str] = None,
) -> fields.Field:
    """Handle Union Fields, including Unions with List Types.

    Args:
        union_type: The Union Type to handle.
        optional: Whether the Union can be None.
        serialization_key: The key to pull the data from before serializing.

    Returns:
        A marshmallow Union field.
    """
    candidate_fields = [cls._get_field_for_type(arg) for arg in get_args(union_type)␣
↪if arg is not type(None)]
    optional_args = [arg is None for arg in get_args(union_type)]
    if optional_args:
        optional = True
    if not candidate_fields:
        raise ValueError(f"Unsupported UnionType field: {union_type}")

    return UnionField(fields=candidate_fields, allow_none=optional,␣
↪attribute=serialization_key)
```

_is_enum(field_type)classmethod

Source Code in griptape/schemas/base_schema.py

```
@classmethod
def _is_enum(cls, field_type: type) -> bool:
    return isinstance(field_type, type) and issubclass(field_type, Enum)
```

_is_list_sequence(field_type)classmethod

Source Code in griptape/schemas/base_schema.py

```
@classmethod
def _is_list_sequence(cls, field_type: type | _SpecialForm) -> bool:
    if isinstance(field_type, type):
        if issubclass(field_type, str) or issubclass(field_type, bytes) or␣
↪issubclass(field_type, tuple):
            return False
```

(continues on next page)

(continued from previous page)

```
        return issubclass(field_type, Sequence)
    return False
```

_is_union(field_type)classmethod

Source Code in griptape/schemas/base_schema.py

```
@classmethod
def _is_union(cls, field_type: type) -> bool:
    return field_type is Union or get_origin(field_type) is Union
```

_resolve_types(attrs_cls, types_override=None)classmethod

Resolve types in an attrs class.

Parameters

Name	Type	Description	Default
attrs_cls	*type*	An attrs class. 	*required*
types_override	*Optional[dict[str, type]]*	A dictionary of types to override. 	*None*

Source Code in griptape/schemas/base_schema.py

```
@classmethod
def _resolve_types(cls, attrs_cls: type, types_override: Optional[dict[str, type]] =␣
↪None) -> None:
    """Resolve types in an attrs class.

    Args:
        attrs_cls: An attrs class.
        types_override: A dictionary of types to override.
    """
    from collections.abc import Sequence
    from typing import Any

    from pydantic import BaseModel
    from schema import Schema

    from griptape.artifacts import (
        ActionArtifact,
        AudioArtifact,
        BaseArtifact,
        BlobArtifact,
        BooleanArtifact,
        ErrorArtifact,
        GenericArtifact,
        ImageArtifact,
        ImageUrlArtifact,
        InfoArtifact,
```

(continues on next page)

(continued from previous page)

```
        JsonArtifact,
        ListArtifact,
        TextArtifact,
    )
    from griptape.common import (
        BaseDeltaMessageContent,
        BaseMessageContent,
        Message,
        PromptStack,
        Reference,
        ToolAction,
    )
    from griptape.drivers.assistant import BaseAssistantDriver
    from griptape.drivers.audio_transcription import BaseAudioTranscriptionDriver
    from griptape.drivers.embedding import BaseEmbeddingDriver
    from griptape.drivers.file_manager import BaseFileManagerDriver
    from griptape.drivers.image_generation import BaseImageGenerationDriver,␣
↪BaseMultiModelImageGenerationDriver
    from griptape.drivers.image_generation_model import BaseImageGenerationModelDriver
    from griptape.drivers.memory.conversation import BaseConversationMemoryDriver
    from griptape.drivers.observability import BaseObservabilityDriver
    from griptape.drivers.prompt import BasePromptDriver
    from griptape.drivers.prompt.base_prompt_driver import StructuredOutputStrategy
    from griptape.drivers.ruleset import BaseRulesetDriver
    from griptape.drivers.sql import BaseSqlDriver
    from griptape.drivers.structure_run import BaseStructureRunDriver
    from griptape.drivers.text_to_speech import BaseTextToSpeechDriver
    from griptape.drivers.vector import BaseVectorStoreDriver
    from griptape.drivers.web_scraper import BaseWebScraperDriver
    from griptape.drivers.web_search import BaseWebSearchDriver
    from griptape.engines.rag import RagContext
    from griptape.events import EventListener
    from griptape.memory import TaskMemory
    from griptape.memory.meta import BaseMetaEntry
    from griptape.memory.structure import BaseConversationMemory, Run
    from griptape.memory.task.storage import BaseArtifactStorage
    from griptape.rules.base_rule import BaseRule
    from griptape.rules.ruleset import Ruleset
    from griptape.structures import Structure
    from griptape.tasks import BaseTask
    from griptape.tokenizers import BaseTokenizer
    from griptape.tools import BaseTool
    from griptape.utils import import_optional_dependency, is_dependency_installed

    if types_override is None:
        types_override = {}

    attrs.resolve_types(
        attrs_cls,
        localns={
            "Any": Any,
            "BasePromptDriver": BasePromptDriver,
```

(continues on next page)

(continued from previous page)

```
            "BaseEmbeddingDriver": BaseEmbeddingDriver,
            "BaseVectorStoreDriver": BaseVectorStoreDriver,
            "BaseTextToSpeechDriver": BaseTextToSpeechDriver,
            "BaseAudioTranscriptionDriver": BaseAudioTranscriptionDriver,
            "BaseConversationMemoryDriver": BaseConversationMemoryDriver,
            "BaseRulesetDriver": BaseRulesetDriver,
            "BaseImageGenerationDriver": BaseImageGenerationDriver,
            "BaseMultiModelImageGenerationDriver":␣
↪BaseMultiModelImageGenerationDriver,
            "BaseImageGenerationModelDriver": BaseImageGenerationModelDriver,
            "BaseWebSearchDriver": BaseWebSearchDriver,
            "BaseWebScraperDriver": BaseWebScraperDriver,
            "BaseFileManagerDriver": BaseFileManagerDriver,
            "BaseSqlDriver": BaseSqlDriver,
            "BaseObservabilityDriver": BaseObservabilityDriver,
            "BaseAssistantDriver": BaseAssistantDriver,
            "BaseStructureRunDriver": BaseStructureRunDriver,
            "BaseArtifact": BaseArtifact,
            "BaseMetaEntry": BaseMetaEntry,
            "PromptStack": PromptStack,
            "EventListener": EventListener,
            "BaseMessageContent": BaseMessageContent,
            "BaseDeltaMessageContent": BaseDeltaMessageContent,
            "BaseTool": BaseTool,
            "BaseTask": BaseTask,
            "TextArtifact": TextArtifact,
            "ImageArtifact": ImageArtifact,
            "ImageUrlArtifact": ImageUrlArtifact,
            "ErrorArtifact": ErrorArtifact,
            "InfoArtifact": InfoArtifact,
            "JsonArtifact": JsonArtifact,
            "BlobArtifact": BlobArtifact,
            "BooleanArtifact": BooleanArtifact,
            "ListArtifact": ListArtifact,
            "AudioArtifact": AudioArtifact,
            "ActionArtifact": ActionArtifact,
            "GenericArtifact": GenericArtifact,
            "Usage": Message.Usage,
            "Structure": Structure,
            "BaseTokenizer": BaseTokenizer,
            "ToolAction": ToolAction,
            "Reference": Reference,
            "Run": Run,
            "Sequence": Sequence,
            "TaskMemory": TaskMemory,
            "State": BaseTask.State,
            "BaseConversationMemory": BaseConversationMemory,
            "BaseArtifactStorage": BaseArtifactStorage,
            "BaseRule": BaseRule,
            "Ruleset": Ruleset,
            "StructuredOutputStrategy": StructuredOutputStrategy,
            "RagContext": RagContext,
```

(continues on next page)

(continued from previous page)

```
            # Third party modules
            "Client": import_optional_dependency("cohere").Client if is_dependency_
↪installed("cohere") else Any,
            "ClientV2": import_optional_dependency("cohere").ClientV2 if is_
↪dependency_installed("cohere") else Any,
            "GenerativeModel": import_optional_dependency("google.generativeai").
↪GenerativeModel
            if is_dependency_installed("google.generativeai")
            else Any,
            "boto3": import_optional_dependency("boto3") if is_dependency_installed(
↪"boto3") else Any,
            "Anthropic": import_optional_dependency("anthropic").Anthropic
            if is_dependency_installed("anthropic")
            else Any,
            "BedrockRuntimeClient": import_optional_dependency("mypy_boto3_bedrock_
↪runtime").BedrockRuntimeClient
            if is_dependency_installed("mypy_boto3_bedrock_runtime")
            else Any,
            "voyageai": import_optional_dependency("voyageai") if is_dependency_
↪installed("voyageai") else Any,
            "Schema": Schema,
            "BaseModel": BaseModel,
            **types_override,
        },
    )
```

from_attrs_cls(attrs_cls, *, types_overrides=None, serializable_overrides=None)classmethod

Generate a Schema from an attrs class.

Parameters

Name	Type	Description	Default
attrs_cls	*type*	An attrs class. 	*required*
types_overrides	*Optional[dict[str, type]]*	A dictionary of types to override when resolving types. 	*None*
serializable_overrides	*Optional[dict[str, bool]]*	A dictionary of field names to whether they are serializable. 	*None*

Source Code in griptape/schemas/base_schema.py

```
@classmethod
def from_attrs_cls(
    cls,
    attrs_cls: type,
    *,
    types_overrides: Optional[dict[str, type]] = None,
```

(continues on next page)

(continued from previous page)

```
    serializable_overrides: Optional[dict[str, bool]] = None,
) -> type:
    """Generate a Schema from an attrs class.

    Args:
        attrs_cls: An attrs class.
        types_overrides: A dictionary of types to override when resolving types.
        serializable_overrides: A dictionary of field names to whether they are␣
↪serializable.
    """
    from marshmallow import post_load

    if serializable_overrides is None:
        serializable_overrides = {}

    class SubSchema(cls):
        @post_load
        def make_obj(self, data: Any, **kwargs) -> Any:
            # Map the serialized keys to their correct deserialization keys
            fields = attrs.fields_dict(attrs_cls)
            for key in list(data):
                if key in fields:
                    field = fields[key]
                    if field.metadata.get("deserialization_key"):
                        data[field.metadata["deserialization_key"]] = data.pop(key)

            return attrs_cls(**data)

    cls._resolve_types(attrs_cls, types_override=types_overrides)
    fields = {}
    for field in attrs.fields(attrs_cls):
        field_key = field.alias or field.name
        if serializable_overrides.get(field_key, field.metadata.get("serializable",␣
↪False)):
            fields[field_key] = cls._get_field_for_type(
                field.type,
                serialization_key=field.metadata.get("serialization_key"),
                types_overrides=types_overrides,
            )
    return SubSchema.from_dict(fields, name=f"{attrs_cls.__name__}Schema")
```

Bytes

Bases:

Field

Source Code in griptape/schemas/bytes_field.py

```
class Bytes(fields.Field):
    def _serialize(self, value: Any, attr: Any, obj: Any, **kwargs) -> str:
        return base64.b64encode(value).decode()
```

(continues on next page)

(continued from previous page)

```
    def _deserialize(self, value: Any, attr: Any, data: Any, **kwargs) -> bytes:
        return base64.b64decode(value)

    def _validate(self, value: Any) -> None:
        if not isinstance(value, bytes):
            raise ValidationError("Invalid input type.")
```

_deserialize(value, attr, data, **kwargs)

Source Code in griptape/schemas/bytes_field.py

```
def _deserialize(self, value: Any, attr: Any, data: Any, **kwargs) -> bytes:
    return base64.b64decode(value)
```

_serialize(value, attr, obj, **kwargs)

Source Code in griptape/schemas/bytes_field.py

```
def _serialize(self, value: Any, attr: Any, obj: Any, **kwargs) -> str:
    return base64.b64encode(value).decode()
```

_validate(value)

Source Code in griptape/schemas/bytes_field.py

```
def _validate(self, value: Any) -> None:
    if not isinstance(value, bytes):
        raise ValidationError("Invalid input type.")
```

PolymorphicSchema

Bases: BaseSchema “)

Source Code in griptape/schemas/polymorphic_schema.py

```
class PolymorphicSchema(BaseSchema):
    """PolymorphicSchema is based on https://github.com/marshmallow-code/marshmallow-
↪oneofschema."""

    def __init__(self, inner_class: Any, types_overrides: Optional[dict[str, type]] =␣
↪None, **kwargs) -> None:
        super().__init__(**kwargs)

        self.inner_class = inner_class
        self.types_overrides = types_overrides

    type_field = "type"
    type_field_remove = True

    def get_obj_type(self, obj: Any) -> Any:
        """Returns name of the schema during dump() calls, given the object being␣
↪dumped."""
```

(continues on next page)

(continued from previous page)

```
        return obj.__class__.__name__

    def get_data_type(self, data: Any) -> Any:
        """Returns name of the schema during load() calls, given the data being
↪loaded. Defaults to looking up `type_field` in the data."""
        data_type = data.get(self.type_field)
        if self.type_field in data and self.type_field_remove:
            data.pop(self.type_field)
        return data_type

    def dump(self, obj: Any, *, many: Any = None, **kwargs) -> Any:
        errors = {}
        result_data = []
        result_errors = {}
        many = self.many if many is None else bool(many)
        if not many:
            result = result_data = self._dump(obj, **kwargs)
        else:
            for idx, o in enumerate(obj):
                try:
                    result = self._dump(o, **kwargs)
                    result_data.append(result)
                except ValidationError as error:
                    result_errors[idx] = error.normalized_messages()
                    result_data.append(error.valid_data)

        result = result_data
        errors = result_errors

        if not errors:
            return result
        exc = ValidationError(errors, data=obj, valid_data=result)  # pyright:
↪ignore[reportArgumentType]
        raise exc

    def _dump(self, obj: Any, *, update_fields: bool = True, **kwargs) -> Any:
        obj_type = self.get_obj_type(obj)

        if not obj_type:
            return (None, {"_schema": f"Unknown object class: {obj.__class__.__name__}
↪"})

        type_schema = BaseSchema.from_attrs_cls(obj.__class__, types_overrides=self.
↪types_overrides)

        if not type_schema:
            return None, {"_schema": f"Unsupported object type: {obj_type}"}

        schema = type_schema if isinstance(type_schema, Schema) else type_schema()

        schema.context.update(getattr(self, "context", {}))
```

(continues on next page)

(continued from previous page)

```
        result = schema.dump(obj, many=False, **kwargs)

        if result is not None:
            result[self.type_field] = obj_type  # pyright: ignore[reportArgumentType,
↪reportCallIssue]

        return result

    def load(self, data: Any, *, many: Any = None, partial: Any = None, unknown: Any␣
↪= None, **kwargs) -> Any:
        errors = {}
        result_data = []
        result_errors = {}
        many = self.many if many is None else bool(many)
        if partial is None:
            partial = self.partial
        if not many:
            try:
                result = result_data = self._load(data, partial=partial,␣
↪unknown=unknown, **kwargs)
            except ValidationError as error:
                result_errors = error.normalized_messages()
                result_data.append(error.valid_data)
        else:
            for idx, item in enumerate(data):
                try:
                    result = self._load(item, partial=partial, **kwargs)
                    result_data.append(result)
                except ValidationError as error:
                    result_errors[idx] = error.normalized_messages()
                    result_data.append(error.valid_data)

        result = result_data
        errors = result_errors

        if not errors:
            return result
        exc = ValidationError(errors, data=data, valid_data=result)
        raise exc

    def _load(self, data: Any, *, partial: Any = None, unknown: Any = None, **kwargs)␣
↪-> Any:
        if not isinstance(data, dict):
            raise ValidationError({"_schema": f"Invalid data type: {data}"})

        data = dict(data)
        unknown = unknown or self.unknown
        data_type = self.get_data_type(data)

        if data_type is None:
            raise ValidationError({self.type_field: ["Missing data for required field.
↪"]})
```

(continues on next page)

(continued from previous page)

```
        type_schema = self.inner_class.get_schema(data_type, module_name=data.get(
↪"module_name"))
        if not type_schema:
            raise ValidationError({self.type_field: [f"Unsupported value: {data_type}
↪"]})

        schema = type_schema if isinstance(type_schema, Schema) else type_schema()

        schema.context.update(getattr(self, "context", {}))

        return schema.load(data, many=False, partial=partial, unknown=unknown,␣
↪**kwargs)

    def validate(self, data: Any, *, many: Any = None, partial: Any = None) -> Any:
↪# pyright: ignore[reportIncompatibleMethodOverride]
        try:
            self.load(data, many=many, partial=partial)
        except ValidationError as ve:
            return ve.messages
        return {}
```

- `inner_class = inner_class` instance-attribute
- `type_field = 'type'` class-attribute instance-attribute
- `type_field_remove = True` class-attribute instance-attribute
- `types_overrides = types_overrides` instance-attribute

init(inner_class, types_overrides=None, **kwargs)

Source Code in griptape/schemas/polymorphic_schema.py

```
def __init__(self, inner_class: Any, types_overrides: Optional[dict[str, type]] =␣
↪None, **kwargs) -> None:
    super().__init__(**kwargs)

    self.inner_class = inner_class
    self.types_overrides = types_overrides
```

_dump(obj, *, update_fields=True, **kwargs)

Source Code in griptape/schemas/polymorphic_schema.py

```
def _dump(self, obj: Any, *, update_fields: bool = True, **kwargs) -> Any:
    obj_type = self.get_obj_type(obj)

    if not obj_type:
        return (None, {"_schema": f"Unknown object class: {obj.__class__.__name__}"})

    type_schema = BaseSchema.from_attrs_cls(obj.__class__, types_overrides=self.types_
↪overrides)
```

(continues on next page)

(continued from previous page)

```
    if not type_schema:
        return None, {"_schema": f"Unsupported object type: {obj_type}"}

    schema = type_schema if isinstance(type_schema, Schema) else type_schema()

    schema.context.update(getattr(self, "context", {}))

    result = schema.dump(obj, many=False, **kwargs)

    if result is not None:
        result[self.type_field] = obj_type  # pyright: ignore[reportArgumentType,
↪reportCallIssue]

    return result
```

_load(data, *, partial=None, unknown=None, **kwargs)

Source Code in griptape/schemas/polymorphic_schema.py

```
def _load(self, data: Any, *, partial: Any = None, unknown: Any = None, **kwargs) ->␣
↪Any:
    if not isinstance(data, dict):
        raise ValidationError({"_schema": f"Invalid data type: {data}"})

    data = dict(data)
    unknown = unknown or self.unknown
    data_type = self.get_data_type(data)

    if data_type is None:
        raise ValidationError({self.type_field: ["Missing data for required field."]})

    type_schema = self.inner_class.get_schema(data_type, module_name=data.get("module_
↪name"))
    if not type_schema:
        raise ValidationError({self.type_field: [f"Unsupported value: {data_type}"]})

    schema = type_schema if isinstance(type_schema, Schema) else type_schema()

    schema.context.update(getattr(self, "context", {}))

    return schema.load(data, many=False, partial=partial, unknown=unknown, **kwargs)
```

dump(obj, *, many=None, **kwargs)

Source Code in griptape/schemas/polymorphic_schema.py

```
def dump(self, obj: Any, *, many: Any = None, **kwargs) -> Any:
    errors = {}
    result_data = []
    result_errors = {}
    many = self.many if many is None else bool(many)
    if not many:
```

(continues on next page)

(continued from previous page)

```
        result = result_data = self._dump(obj, **kwargs)
    else:
        for idx, o in enumerate(obj):
            try:
                result = self._dump(o, **kwargs)
                result_data.append(result)
            except ValidationError as error:
                result_errors[idx] = error.normalized_messages()
                result_data.append(error.valid_data)

    result = result_data
    errors = result_errors

    if not errors:
        return result
    exc = ValidationError(errors, data=obj, valid_data=result)  # pyright:␣
↪ignore[reportArgumentType]
    raise exc
```

get_data_type(data)

Source Code in griptape/schemas/polymorphic_schema.py

```
def get_data_type(self, data: Any) -> Any:
    """Returns name of the schema during load() calls, given the data being loaded.␣
↪Defaults to looking up `type_field` in the data."""
    data_type = data.get(self.type_field)
    if self.type_field in data and self.type_field_remove:
        data.pop(self.type_field)
    return data_type
```

get_obj_type(obj)

Source Code in griptape/schemas/polymorphic_schema.py

```
def get_obj_type(self, obj: Any) -> Any:
    """Returns name of the schema during dump() calls, given the object being dumped."
↪""
    return obj.__class__.__name__
```

load(data, *, many=None, partial=None, unknown=None, **kwargs)

Source Code in griptape/schemas/polymorphic_schema.py

```
def load(self, data: Any, *, many: Any = None, partial: Any = None, unknown: Any =␣
↪None, **kwargs) -> Any:
    errors = {}
    result_data = []
    result_errors = {}
    many = self.many if many is None else bool(many)
    if partial is None:
        partial = self.partial
```

(continues on next page)

(continued from previous page)

```
    if not many:
        try:
            result = result_data = self._load(data, partial=partial, unknown=unknown,␣
↪**kwargs)
        except ValidationError as error:
            result_errors = error.normalized_messages()
            result_data.append(error.valid_data)
    else:
        for idx, item in enumerate(data):
            try:
                result = self._load(item, partial=partial, **kwargs)
                result_data.append(result)
            except ValidationError as error:
                result_errors[idx] = error.normalized_messages()
                result_data.append(error.valid_data)

    result = result_data
    errors = result_errors

    if not errors:
        return result
    exc = ValidationError(errors, data=data, valid_data=result)
    raise exc
```

validate(data, *, many=None, partial=None)

Source Code in griptape/schemas/polymorphic_schema.py

```
def validate(self, data: Any, *, many: Any = None, partial: Any = None) -> Any:  #␣
↪pyright: ignore[reportIncompatibleMethodOverride]
    try:
        self.load(data, many=many, partial=partial)
    except ValidationError as ve:
        return ve.messages
    return {}
```

PydanticModel

Bases:

Field

Source Code in griptape/schemas/pydantic_model_field.py

```
class PydanticModel(fields.Field):
    def _serialize(self, value: Optional[BaseModel], attr: Any, obj: Any, **kwargs) ->
↪ Optional[dict]:
        if value is None:
            return None
        return value.model_dump()

    def _deserialize(self, value: dict, attr: Any, data: Any, **kwargs) -> BaseModel:
        return RootModel(value)
```

_deserialize(value, attr, data, **kwargs)

Source Code in griptape/schemas/pydantic_model_field.py

```
def _deserialize(self, value: dict, attr: Any, data: Any, **kwargs) -> BaseModel:
    return RootModel(value)
```

_serialize(value, attr, obj, **kwargs)

Source Code in griptape/schemas/pydantic_model_field.py

```
def _serialize(self, value: Optional[BaseModel], attr: Any, obj: Any, **kwargs) ->␣
↪Optional[dict]:
    if value is None:
        return None
    return value.model_dump()
```

Union

Bases:

Field

Parameters

Name	Type	Description	Default
fields	<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	The list of candidate fields to try. 	*required*
reverse_serialize_candidates	*bool*	Whether to try the candidates in reverse order when serializing. 	*False*

Source Code in griptape/schemas/union_field.py

```
class Union(marshmallow.fields.Field):
    """Field that accepts any one of multiple fields.

    Source: https://github.com/adamboche/python-marshmallow-union
    Each argument will be tried until one succeeds.

    Args:
        fields: The list of candidate fields to try.
        reverse_serialize_candidates: Whether to try the candidates in reverse order␣
↪when serializing.
    """

    def __init__(
```

(continues on next page)

(continued from previous page)

```
        self,
        fields: list[marshmallow.fields.Field],
        *,
        reverse_serialize_candidates: bool = False,
        **kwargs: Any,
    ) -> None:
        self._candidate_fields = fields
        self._reverse_serialize_candidates = reverse_serialize_candidates
        super().__init__(**kwargs)

    def _serialize(self, value: Any, attr: str | None, obj: str, **kwargs: Any) ->␣
↪Any:
        """Pulls the value for the given key from the object, applies the fields␣
↪formatting and returns the result.

        Args:
            value: The value to be serialized.
            attr: The attribute or key to get from the object.
            obj: The object to pull the key from.
            kwargs: Field-specific keyword arguments.

        Raises:
            marshmallow.exceptions.ValidationError: In case of formatting problem
        """
        error_store = kwargs.pop("error_store", marshmallow.error_store.ErrorStore())
        fields = (
            list(reversed(self._candidate_fields)) if self._reverse_serialize_
↪candidates else self._candidate_fields
        )

        for candidate_field in fields:
            try:
                # pylint: disable=protected-access
                return candidate_field._serialize(value, attr, obj, error_store=error_
↪store, **kwargs)
            except (TypeError, ValueError) as e:
                error_store.store_error({attr: str(e)})

        raise ExceptionGroupError("All serializers raised exceptions.", error_store.
↪errors)

    def _deserialize(self, value: Any, attr: str | None = None, data: Any = None,␣
↪**kwargs: Any) -> Any:
        """Deserialize ``value``.

        Args:
            value: The value to be deserialized.
            attr: The attribute/key in `data` to be deserialized.
            data: The raw input data passed to the `Schema.load`.
            kwargs: Field-specific keyword arguments.

        Raises:
```

(continues on next page)

(continued from previous page)

```
            ValidationError: If an invalid value is passed or if a required value is␣
↪missing.
        """
        errors = []
        for candidate_field in self._candidate_fields:
            try:
                return candidate_field.deserialize(value, attr, data, **kwargs)
            except marshmallow.exceptions.ValidationError as exc:
                errors.append(exc.messages)

        raise marshmallow.exceptions.ValidationError(message=errors, field_name=attr␣
↪or "")
```

- `_candidate_fields = fields` instance-attribute
- `_reverse_serialize_candidates = reverse_serialize_candidates` instance-attribute

init(fields, *, reverse_serialize_candidates=False, **kwargs)

Source Code in griptape/schemas/union_field.py

```
def __init__(
    self,
    fields: list[marshmallow.fields.Field],
    *,
    reverse_serialize_candidates: bool = False,
    **kwargs: Any,
) -> None:
    self._candidate_fields = fields
    self._reverse_serialize_candidates = reverse_serialize_candidates
    super().__init__(**kwargs)
```

_deserialize(value, attr=None, data=None, **kwargs)

Deserialize `value` .

Parameters

Name	Type	Description	Default
value	*Any*	The value to be deserialized. 	*required*
attr	*str I None*	The attribute/key in *data* to be deserialized. 	*None*
data	*Any*	The raw input data passed to the *Schema.load*. 	*None*
kwargs	*Any*	Field-specific keyword arguments. 	*{}*

Raises

Source Code in griptape/schemas/union_field.py

```
def _deserialize(self, value: Any, attr: str | None = None, data: Any = None,␣
↪**kwargs: Any) -> Any:
    """Deserialize ``value``.

    Args:
        value: The value to be deserialized.
        attr: The attribute/key in `data` to be deserialized.
        data: The raw input data passed to the `Schema.load`.
        kwargs: Field-specific keyword arguments.

    Raises:
        ValidationError: If an invalid value is passed or if a required value is␣
↪missing.
    """
    errors = []
    for candidate_field in self._candidate_fields:
        try:
            return candidate_field.deserialize(value, attr, data, **kwargs)
        except marshmallow.exceptions.ValidationError as exc:
            errors.append(exc.messages)

    raise marshmallow.exceptions.ValidationError(message=errors, field_name=attr or "
↪")
```

_serialize(value, attr, obj, **kwargs)

Pulls the value for the given key from the object, applies the field's formatting and returns the result.

Parameters

Name	Type	Description	Default
value	*Any*	The value to be serialized. 	*required*
attr	*str I None*	The attribute or key to get from the object. 	*required*
obj	*str*	The object to pull the key from. 	*required*
kwargs	*Any*	Field-specific keyword arguments. 	*{}*

Raises

Source Code in griptape/schemas/union_field.py

```
def _serialize(self, value: Any, attr: str | None, obj: str, **kwargs: Any) -> Any:
    """Pulls the value for the given key from the object, applies the fields␣
```

(continues on next page)

(continued from previous page)

```
↪formatting and returns the result.

    Args:
        value: The value to be serialized.
        attr: The attribute or key to get from the object.
        obj: The object to pull the key from.
        kwargs: Field-specific keyword arguments.

    Raises:
        marshmallow.exceptions.ValidationError: In case of formatting problem
    """
    error_store = kwargs.pop("error_store", marshmallow.error_store.ErrorStore())
    fields = (
        list(reversed(self._candidate_fields)) if self._reverse_serialize_candidates␣
↪else self._candidate_fields
    )

    for candidate_field in fields:
        try:
            # pylint: disable=protected-access
            return candidate_field._serialize(value, attr, obj, error_store=error_
↪store, **kwargs)
        except (TypeError, ValueError) as e:
            error_store.store_error({attr: str(e)})

    raise ExceptionGroupError("All serializers raised exceptions.", error_store.
↪errors)
```

base_schema

Bases:

Schema

Source Code in griptape/schemas/base_schema.py

```
class BaseSchema(Schema):
    class Meta:
        unknown = INCLUDE

    DATACLASS_TYPE_MAPPING = {
        **Schema.TYPE_MAPPING,
        dict: fields.Dict,
        bytes: Bytes,
        Any: fields.Raw,
        BaseModel: PydanticModel,
    }

    @classmethod
    def from_attrs_cls(
        cls,
        attrs_cls: type,
        *,
```

(continues on next page)

(continued from previous page)

```
        types_overrides: Optional[dict[str, type]] = None,
        serializable_overrides: Optional[dict[str, bool]] = None,
    ) -> type:
        """Generate a Schema from an attrs class.

        Args:
            attrs_cls: An attrs class.
            types_overrides: A dictionary of types to override when resolving types.
            serializable_overrides: A dictionary of field names to whether they are␣
↪serializable.
        """
        from marshmallow import post_load

        if serializable_overrides is None:
            serializable_overrides = {}

        class SubSchema(cls):
            @post_load
            def make_obj(self, data: Any, **kwargs) -> Any:
                # Map the serialized keys to their correct deserialization keys
                fields = attrs.fields_dict(attrs_cls)
                for key in list(data):
                    if key in fields:
                        field = fields[key]
                        if field.metadata.get("deserialization_key"):
                            data[field.metadata["deserialization_key"]] = data.
↪pop(key)

                return attrs_cls(**data)

        cls._resolve_types(attrs_cls, types_override=types_overrides)
        fields = {}
        for field in attrs.fields(attrs_cls):
            field_key = field.alias or field.name
            if serializable_overrides.get(field_key, field.metadata.get("serializable
↪", False)):
                fields[field_key] = cls._get_field_for_type(
                    field.type,
                    serialization_key=field.metadata.get("serialization_key"),
                    types_overrides=types_overrides,
                )
        return SubSchema.from_dict(fields, name=f"{attrs_cls.__name__}Schema")

    @classmethod
    def _get_field_for_type(
        cls,
        field_type: type,
        serialization_key: Optional[str] = None,
        types_overrides: Optional[dict[str, type]] = None,
    ) -> fields.Field | fields.Nested:
        """Generate a marshmallow Field instance from a Python type.
```

(continues on next page)

(continued from previous page)

```
        Args:
            field_type: A field type.
            serialization_key: The key to pull the data from before serializing.
            types_overrides: A dictionary of types to override when resolving types.
        """
        from griptape.schemas.polymorphic_schema import PolymorphicSchema

        field_class, args, optional = cls._get_field_type_info(field_type)

        if field_class is None:
            return fields.Constant(None, allow_none=True)

        # Resolve TypeVars to their bound type
        if isinstance(field_class, TypeVar):
            field_class = field_class.__bound__
            if field_class is None:
                return fields.Raw(allow_none=optional, attribute=serialization_key)

        if cls._is_union(field_type):
            return cls._handle_union(
                field_type,
                optional=optional,
                serialization_key=serialization_key,
            )
        if attrs.has(field_class):
            if ABC in field_class.__bases__:
                schema = PolymorphicSchema(field_class, types_overrides=types_
↪overrides)
            else:
                schema = cls.from_attrs_cls(field_class, types_overrides=types_
↪overrides)
            return fields.Nested(schema, allow_none=optional, attribute=serialization_
↪key)
        if cls._is_enum(field_type):
            return fields.String(allow_none=optional, attribute=serialization_key)
        if cls._is_list_sequence(field_class):
            if args:
                return cls._handle_list(
                    args[0],
                    optional=optional,
                    serialization_key=serialization_key,
                )
            raise ValueError(f"Missing type for list field: {field_type}")
        field_class = cls.DATACLASS_TYPE_MAPPING.get(field_class, fields.Raw)

        return field_class(allow_none=optional, attribute=serialization_key)

    @classmethod
    def _handle_list(
        cls,
        list_type: type,
        *,
```

(continues on next page)

(continued from previous page)

```
        optional: bool,
        serialization_key: Optional[str] = None,
    ) -> fields.Field:
        """Handle List Fields, including Union Types.

        Args:
            list_type: The List type to handle.
            optional: Whether the List can be none.
            serialization_key: The key to pull the data from before serializing.

        Returns:
            A marshmallow List field.
        """
        if cls._is_union(list_type):
            instance = cls._handle_union(
                list_type,
                optional=optional,
                serialization_key=serialization_key,
            )
        else:
            instance = cls._get_field_for_type(list_type, serialization_
↪key=serialization_key)
        return fields.List(cls_or_instance=instance, allow_none=optional,␣
↪attribute=serialization_key)

    @classmethod
    def _handle_union(
        cls,
        union_type: type,
        *,
        optional: bool,
        serialization_key: Optional[str] = None,
    ) -> fields.Field:
        """Handle Union Fields, including Unions with List Types.

        Args:
            union_type: The Union Type to handle.
            optional: Whether the Union can be None.
            serialization_key: The key to pull the data from before serializing.

        Returns:
            A marshmallow Union field.
        """
        candidate_fields = [cls._get_field_for_type(arg) for arg in get_args(union_
↪type) if arg is not type(None)]
        optional_args = [arg is None for arg in get_args(union_type)]
        if optional_args:
            optional = True
        if not candidate_fields:
            raise ValueError(f"Unsupported UnionType field: {union_type}")

        return UnionField(fields=candidate_fields, allow_none=optional,␣
```

(continues on next page)

(continued from previous page)

```
↪attribute=serialization_key)

    @classmethod
    def _get_field_type_info(cls, field_type: type) -> tuple[type, tuple[type, ...],␣
↪bool]:
        """Get information about a field type.

        Args:
            field_type: A field type.
        """
        origin = get_origin(field_type) or field_type
        args = get_args(field_type)
        optional = False

        if origin is Union:
            origin = args[0]
            if len(args) > 1 and args[1] is type(None):
                optional = True

            origin, args, _ = cls._get_field_type_info(origin)
        elif origin is Literal:
            origin = type(args[0])
            args = ()

        return origin, args, optional

    @classmethod
    def _resolve_types(cls, attrs_cls: type, types_override: Optional[dict[str,␣
↪type]] = None) -> None:
        """Resolve types in an attrs class.

        Args:
            attrs_cls: An attrs class.
            types_override: A dictionary of types to override.
        """
        from collections.abc import Sequence
        from typing import Any

        from pydantic import BaseModel
        from schema import Schema

        from griptape.artifacts import (
            ActionArtifact,
            AudioArtifact,
            BaseArtifact,
            BlobArtifact,
            BooleanArtifact,
            ErrorArtifact,
            GenericArtifact,
            ImageArtifact,
            ImageUrlArtifact,
            InfoArtifact,
```

(continues on next page)

(continued from previous page)

```
            JsonArtifact,
            ListArtifact,
            TextArtifact,
        )
        from griptape.common import (
            BaseDeltaMessageContent,
            BaseMessageContent,
            Message,
            PromptStack,
            Reference,
            ToolAction,
        )
        from griptape.drivers.assistant import BaseAssistantDriver
        from griptape.drivers.audio_transcription import BaseAudioTranscriptionDriver
        from griptape.drivers.embedding import BaseEmbeddingDriver
        from griptape.drivers.file_manager import BaseFileManagerDriver
        from griptape.drivers.image_generation import BaseImageGenerationDriver,␣
↪BaseMultiModelImageGenerationDriver
        from griptape.drivers.image_generation_model import␣
↪BaseImageGenerationModelDriver
        from griptape.drivers.memory.conversation import BaseConversationMemoryDriver
        from griptape.drivers.observability import BaseObservabilityDriver
        from griptape.drivers.prompt import BasePromptDriver
        from griptape.drivers.prompt.base_prompt_driver import␣
↪StructuredOutputStrategy
        from griptape.drivers.ruleset import BaseRulesetDriver
        from griptape.drivers.sql import BaseSqlDriver
        from griptape.drivers.structure_run import BaseStructureRunDriver
        from griptape.drivers.text_to_speech import BaseTextToSpeechDriver
        from griptape.drivers.vector import BaseVectorStoreDriver
        from griptape.drivers.web_scraper import BaseWebScraperDriver
        from griptape.drivers.web_search import BaseWebSearchDriver
        from griptape.engines.rag import RagContext
        from griptape.events import EventListener
        from griptape.memory import TaskMemory
        from griptape.memory.meta import BaseMetaEntry
        from griptape.memory.structure import BaseConversationMemory, Run
        from griptape.memory.task.storage import BaseArtifactStorage
        from griptape.rules.base_rule import BaseRule
        from griptape.rules.ruleset import Ruleset
        from griptape.structures import Structure
        from griptape.tasks import BaseTask
        from griptape.tokenizers import BaseTokenizer
        from griptape.tools import BaseTool
        from griptape.utils import import_optional_dependency, is_dependency_installed

        if types_override is None:
            types_override = {}

        attrs.resolve_types(
            attrs_cls,
            localns={
```

(continues on next page)

(continued from previous page)

```
                "Any": Any,
                "BasePromptDriver": BasePromptDriver,
                "BaseEmbeddingDriver": BaseEmbeddingDriver,
                "BaseVectorStoreDriver": BaseVectorStoreDriver,
                "BaseTextToSpeechDriver": BaseTextToSpeechDriver,
                "BaseAudioTranscriptionDriver": BaseAudioTranscriptionDriver,
                "BaseConversationMemoryDriver": BaseConversationMemoryDriver,
                "BaseRulesetDriver": BaseRulesetDriver,
                "BaseImageGenerationDriver": BaseImageGenerationDriver,
                "BaseMultiModelImageGenerationDriver":␣
↪BaseMultiModelImageGenerationDriver,
                "BaseImageGenerationModelDriver": BaseImageGenerationModelDriver,
                "BaseWebSearchDriver": BaseWebSearchDriver,
                "BaseWebScraperDriver": BaseWebScraperDriver,
                "BaseFileManagerDriver": BaseFileManagerDriver,
                "BaseSqlDriver": BaseSqlDriver,
                "BaseObservabilityDriver": BaseObservabilityDriver,
                "BaseAssistantDriver": BaseAssistantDriver,
                "BaseStructureRunDriver": BaseStructureRunDriver,
                "BaseArtifact": BaseArtifact,
                "BaseMetaEntry": BaseMetaEntry,
                "PromptStack": PromptStack,
                "EventListener": EventListener,
                "BaseMessageContent": BaseMessageContent,
                "BaseDeltaMessageContent": BaseDeltaMessageContent,
                "BaseTool": BaseTool,
                "BaseTask": BaseTask,
                "TextArtifact": TextArtifact,
                "ImageArtifact": ImageArtifact,
                "ImageUrlArtifact": ImageUrlArtifact,
                "ErrorArtifact": ErrorArtifact,
                "InfoArtifact": InfoArtifact,
                "JsonArtifact": JsonArtifact,
                "BlobArtifact": BlobArtifact,
                "BooleanArtifact": BooleanArtifact,
                "ListArtifact": ListArtifact,
                "AudioArtifact": AudioArtifact,
                "ActionArtifact": ActionArtifact,
                "GenericArtifact": GenericArtifact,
                "Usage": Message.Usage,
                "Structure": Structure,
                "BaseTokenizer": BaseTokenizer,
                "ToolAction": ToolAction,
                "Reference": Reference,
                "Run": Run,
                "Sequence": Sequence,
                "TaskMemory": TaskMemory,
                "State": BaseTask.State,
                "BaseConversationMemory": BaseConversationMemory,
                "BaseArtifactStorage": BaseArtifactStorage,
                "BaseRule": BaseRule,
                "Ruleset": Ruleset,
```

(continues on next page)

(continued from previous page)

```
                "StructuredOutputStrategy": StructuredOutputStrategy,
                "RagContext": RagContext,
                # Third party modules
                "Client": import_optional_dependency("cohere").Client if is_
↪dependency_installed("cohere") else Any,
                "ClientV2": import_optional_dependency("cohere").ClientV2 if is_
↪dependency_installed("cohere") else Any,
                "GenerativeModel": import_optional_dependency("google.generativeai").
↪GenerativeModel
                if is_dependency_installed("google.generativeai")
                else Any,
                "boto3": import_optional_dependency("boto3") if is_dependency_
↪installed("boto3") else Any,
                "Anthropic": import_optional_dependency("anthropic").Anthropic
                if is_dependency_installed("anthropic")
                else Any,
                "BedrockRuntimeClient": import_optional_dependency("mypy_boto3_
↪bedrock_runtime").BedrockRuntimeClient
                if is_dependency_installed("mypy_boto3_bedrock_runtime")
                else Any,
                "voyageai": import_optional_dependency("voyageai") if is_dependency_
↪installed("voyageai") else Any,
                "Schema": Schema,
                "BaseModel": BaseModel,
                **types_override,
            },
        )

    @classmethod
    def _is_list_sequence(cls, field_type: type | _SpecialForm) -> bool:
        if isinstance(field_type, type):
            if issubclass(field_type, str) or issubclass(field_type, bytes) or␣
↪issubclass(field_type, tuple):
                return False
            return issubclass(field_type, Sequence)
        return False

    @classmethod
    def _is_union(cls, field_type: type) -> bool:
        return field_type is Union or get_origin(field_type) is Union

    @classmethod
    def _is_enum(cls, field_type: type) -> bool:
        return isinstance(field_type, type) and issubclass(field_type, Enum)
```

- `DATACLASS_TYPE_MAPPING = {None: Schema.TYPE_MAPPING, dict: fields.Dict, bytes: Bytes, Any: fields.Raw, BaseModel: PydanticModel}` class-attribute instance-attribute

Meta

Source Code in griptape/schemas/base_schema.py

```
class Meta:
    unknown = INCLUDE
```

- `unknown = INCLUDE` class-attribute instance-attribute

_get_field_for_type(field_type, serialization_key=None, types_overrides=None)classmethod

Generate a marshmallow Field instance from a Python type.

Parameters

Name	Type	Description	Default
field_type	*type*	A field type. 	*required*
serialization_key	*Optional[str]*	The key to pull the data from before serializing. 	*None*
types_overrides	*Optional[dict[str, type]]*	A dictionary of types to override when resolving types. 	*None*

Source Code in griptape/schemas/base_schema.py

```
@classmethod
def _get_field_for_type(
    cls,
    field_type: type,
    serialization_key: Optional[str] = None,
    types_overrides: Optional[dict[str, type]] = None,
) -> fields.Field | fields.Nested:
    """Generate a marshmallow Field instance from a Python type.

    Args:
        field_type: A field type.
        serialization_key: The key to pull the data from before serializing.
        types_overrides: A dictionary of types to override when resolving types.
    """
    from griptape.schemas.polymorphic_schema import PolymorphicSchema

    field_class, args, optional = cls._get_field_type_info(field_type)

    if field_class is None:
        return fields.Constant(None, allow_none=True)

    # Resolve TypeVars to their bound type
    if isinstance(field_class, TypeVar):
        field_class = field_class.__bound__
        if field_class is None:
            return fields.Raw(allow_none=optional, attribute=serialization_key)
```

(continues on next page)

(continued from previous page)

```
    if cls._is_union(field_type):
        return cls._handle_union(
            field_type,
            optional=optional,
            serialization_key=serialization_key,
        )
    if attrs.has(field_class):
        if ABC in field_class.__bases__:
            schema = PolymorphicSchema(field_class, types_overrides=types_overrides)
        else:
            schema = cls.from_attrs_cls(field_class, types_overrides=types_overrides)
        return fields.Nested(schema, allow_none=optional, attribute=serialization_key)
    if cls._is_enum(field_type):
        return fields.String(allow_none=optional, attribute=serialization_key)
    if cls._is_list_sequence(field_class):
        if args:
            return cls._handle_list(
                args[0],
                optional=optional,
                serialization_key=serialization_key,
            )
        raise ValueError(f"Missing type for list field: {field_type}")
    field_class = cls.DATACLASS_TYPE_MAPPING.get(field_class, fields.Raw)

    return field_class(allow_none=optional, attribute=serialization_key)
```

_get_field_type_info(field_type)classmethod

Get information about a field type.

Parameters

Name	Type	Description	Default
field_type	*type*	A field type. 	*required*

Source Code in griptape/schemas/base_schema.py

```
@classmethod
def _get_field_type_info(cls, field_type: type) -> tuple[type, tuple[type, ...],␣
↪bool]:
    """Get information about a field type.

    Args:
        field_type: A field type.
    """
    origin = get_origin(field_type) or field_type
    args = get_args(field_type)
    optional = False
```

(continues on next page)

(continued from previous page)

```
    if origin is Union:
        origin = args[0]
        if len(args) > 1 and args[1] is type(None):
            optional = True

        origin, args, _ = cls._get_field_type_info(origin)
    elif origin is Literal:
        origin = type(args[0])
        args = ()

    return origin, args, optional
```

_handle_list(list_type, *, optional, serialization_key=None)classmethod

Handle List Fields, including Union Types.

Parameters

Name	Type	Description	Default
list_type	*type*	The List type to handle. 	*required*
optional	*bool*	Whether the List can be none. 	*required*
serialization_key	*Optional[str]*	The key to pull the data from before serializing. 	*None*

Returns

Type	Description
Field	A marshmallow List field.

Source Code in griptape/schemas/base_schema.py

```
@classmethod
def _handle_list(
    cls,
    list_type: type,
    *,
    optional: bool,
    serialization_key: Optional[str] = None,
) -> fields.Field:
    """Handle List Fields, including Union Types.

    Args:
        list_type: The List type to handle.
        optional: Whether the List can be none.
        serialization_key: The key to pull the data from before serializing.
```

(continues on next page)

(continued from previous page)

```
    Returns:
        A marshmallow List field.
    """
    if cls._is_union(list_type):
        instance = cls._handle_union(
            list_type,
            optional=optional,
            serialization_key=serialization_key,
        )
    else:
        instance = cls._get_field_for_type(list_type, serialization_key=serialization_
↪key)
    return fields.List(cls_or_instance=instance, allow_none=optional,␣
↪attribute=serialization_key)
```

_handle_union(union_type, *, optional, serialization_key=None)classmethod

Handle Union Fields, including Unions with List Types.

Parameters

Name	Type	Description	Default
union_type	*type*	The Union Type to handle. 	*required*
optional	*bool*	Whether the Union can be None. 	*required*
serialization_key	*Optional[str]*	The key to pull the data from before serializing. 	*None*

Returns

Type	Description
Field	A marshmallow Union field.

Source Code in griptape/schemas/base_schema.py

```
@classmethod
def _handle_union(
    cls,
    union_type: type,
    *,
    optional: bool,
    serialization_key: Optional[str] = None,
) -> fields.Field:
    """Handle Union Fields, including Unions with List Types.
```

(continues on next page)

(continued from previous page)

```
    Args:
        union_type: The Union Type to handle.
        optional: Whether the Union can be None.
        serialization_key: The key to pull the data from before serializing.

    Returns:
        A marshmallow Union field.
    """
    candidate_fields = [cls._get_field_for_type(arg) for arg in get_args(union_type)␣
↪if arg is not type(None)]
    optional_args = [arg is None for arg in get_args(union_type)]
    if optional_args:
        optional = True
    if not candidate_fields:
        raise ValueError(f"Unsupported UnionType field: {union_type}")

    return UnionField(fields=candidate_fields, allow_none=optional,␣
↪attribute=serialization_key)
```

_is_enum(field_type)classmethod

Source Code in griptape/schemas/base_schema.py

```
@classmethod
def _is_enum(cls, field_type: type) -> bool:
    return isinstance(field_type, type) and issubclass(field_type, Enum)
```

_is_list_sequence(field_type)classmethod

Source Code in griptape/schemas/base_schema.py

```
@classmethod
def _is_list_sequence(cls, field_type: type | _SpecialForm) -> bool:
    if isinstance(field_type, type):
        if issubclass(field_type, str) or issubclass(field_type, bytes) or␣
↪issubclass(field_type, tuple):
            return False
        return issubclass(field_type, Sequence)
    return False
```

_is_union(field_type)classmethod

Source Code in griptape/schemas/base_schema.py

```
@classmethod
def _is_union(cls, field_type: type) -> bool:
    return field_type is Union or get_origin(field_type) is Union
```

_resolve_types(attrs_cls, types_override=None)classmethod

Resolve types in an attrs class.

Parameters

Name	Type	Description	Default
attrs_cls	*type*	An attrs class. 	*required*
types_override	*Optional[dict[str, type]]*	A dictionary of types to override. 	*None*

Source Code in griptape/schemas/base_schema.py

```
@classmethod
def _resolve_types(cls, attrs_cls: type, types_override: Optional[dict[str, type]] =␣
↪None) -> None:
    """Resolve types in an attrs class.

    Args:
        attrs_cls: An attrs class.
        types_override: A dictionary of types to override.
    """
    from collections.abc import Sequence
    from typing import Any

    from pydantic import BaseModel
    from schema import Schema

    from griptape.artifacts import (
        ActionArtifact,
        AudioArtifact,
        BaseArtifact,
        BlobArtifact,
        BooleanArtifact,
        ErrorArtifact,
        GenericArtifact,
        ImageArtifact,
        ImageUrlArtifact,
        InfoArtifact,
        JsonArtifact,
        ListArtifact,
        TextArtifact,
    )
    from griptape.common import (
        BaseDeltaMessageContent,
        BaseMessageContent,
        Message,
        PromptStack,
        Reference,
        ToolAction,
    )
    from griptape.drivers.assistant import BaseAssistantDriver
```

(continues on next page)

(continued from previous page)

```
    from griptape.drivers.audio_transcription import BaseAudioTranscriptionDriver
    from griptape.drivers.embedding import BaseEmbeddingDriver
    from griptape.drivers.file_manager import BaseFileManagerDriver
    from griptape.drivers.image_generation import BaseImageGenerationDriver,␣
↪BaseMultiModelImageGenerationDriver
    from griptape.drivers.image_generation_model import BaseImageGenerationModelDriver
    from griptape.drivers.memory.conversation import BaseConversationMemoryDriver
    from griptape.drivers.observability import BaseObservabilityDriver
    from griptape.drivers.prompt import BasePromptDriver
    from griptape.drivers.prompt.base_prompt_driver import StructuredOutputStrategy
    from griptape.drivers.ruleset import BaseRulesetDriver
    from griptape.drivers.sql import BaseSqlDriver
    from griptape.drivers.structure_run import BaseStructureRunDriver
    from griptape.drivers.text_to_speech import BaseTextToSpeechDriver
    from griptape.drivers.vector import BaseVectorStoreDriver
    from griptape.drivers.web_scraper import BaseWebScraperDriver
    from griptape.drivers.web_search import BaseWebSearchDriver
    from griptape.engines.rag import RagContext
    from griptape.events import EventListener
    from griptape.memory import TaskMemory
    from griptape.memory.meta import BaseMetaEntry
    from griptape.memory.structure import BaseConversationMemory, Run
    from griptape.memory.task.storage import BaseArtifactStorage
    from griptape.rules.base_rule import BaseRule
    from griptape.rules.ruleset import Ruleset
    from griptape.structures import Structure
    from griptape.tasks import BaseTask
    from griptape.tokenizers import BaseTokenizer
    from griptape.tools import BaseTool
    from griptape.utils import import_optional_dependency, is_dependency_installed

    if types_override is None:
        types_override = {}

    attrs.resolve_types(
        attrs_cls,
        localns={
            "Any": Any,
            "BasePromptDriver": BasePromptDriver,
            "BaseEmbeddingDriver": BaseEmbeddingDriver,
            "BaseVectorStoreDriver": BaseVectorStoreDriver,
            "BaseTextToSpeechDriver": BaseTextToSpeechDriver,
            "BaseAudioTranscriptionDriver": BaseAudioTranscriptionDriver,
            "BaseConversationMemoryDriver": BaseConversationMemoryDriver,
            "BaseRulesetDriver": BaseRulesetDriver,
            "BaseImageGenerationDriver": BaseImageGenerationDriver,
            "BaseMultiModelImageGenerationDriver":␣
↪BaseMultiModelImageGenerationDriver,
            "BaseImageGenerationModelDriver": BaseImageGenerationModelDriver,
            "BaseWebSearchDriver": BaseWebSearchDriver,
            "BaseWebScraperDriver": BaseWebScraperDriver,
            "BaseFileManagerDriver": BaseFileManagerDriver,
```

(continues on next page)

(continued from previous page)

```
            "BaseSqlDriver": BaseSqlDriver,
            "BaseObservabilityDriver": BaseObservabilityDriver,
            "BaseAssistantDriver": BaseAssistantDriver,
            "BaseStructureRunDriver": BaseStructureRunDriver,
            "BaseArtifact": BaseArtifact,
            "BaseMetaEntry": BaseMetaEntry,
            "PromptStack": PromptStack,
            "EventListener": EventListener,
            "BaseMessageContent": BaseMessageContent,
            "BaseDeltaMessageContent": BaseDeltaMessageContent,
            "BaseTool": BaseTool,
            "BaseTask": BaseTask,
            "TextArtifact": TextArtifact,
            "ImageArtifact": ImageArtifact,
            "ImageUrlArtifact": ImageUrlArtifact,
            "ErrorArtifact": ErrorArtifact,
            "InfoArtifact": InfoArtifact,
            "JsonArtifact": JsonArtifact,
            "BlobArtifact": BlobArtifact,
            "BooleanArtifact": BooleanArtifact,
            "ListArtifact": ListArtifact,
            "AudioArtifact": AudioArtifact,
            "ActionArtifact": ActionArtifact,
            "GenericArtifact": GenericArtifact,
            "Usage": Message.Usage,
            "Structure": Structure,
            "BaseTokenizer": BaseTokenizer,
            "ToolAction": ToolAction,
            "Reference": Reference,
            "Run": Run,
            "Sequence": Sequence,
            "TaskMemory": TaskMemory,
            "State": BaseTask.State,
            "BaseConversationMemory": BaseConversationMemory,
            "BaseArtifactStorage": BaseArtifactStorage,
            "BaseRule": BaseRule,
            "Ruleset": Ruleset,
            "StructuredOutputStrategy": StructuredOutputStrategy,
            "RagContext": RagContext,
            # Third party modules
            "Client": import_optional_dependency("cohere").Client if is_dependency_
↪installed("cohere") else Any,
            "ClientV2": import_optional_dependency("cohere").ClientV2 if is_
↪dependency_installed("cohere") else Any,
            "GenerativeModel": import_optional_dependency("google.generativeai").
↪GenerativeModel
            if is_dependency_installed("google.generativeai")
            else Any,
            "boto3": import_optional_dependency("boto3") if is_dependency_installed(
↪"boto3") else Any,
            "Anthropic": import_optional_dependency("anthropic").Anthropic
            if is_dependency_installed("anthropic")
```

(continues on next page)

(continued from previous page)

```
            else Any,
            "BedrockRuntimeClient": import_optional_dependency("mypy_boto3_bedrock_
↪runtime").BedrockRuntimeClient
            if is_dependency_installed("mypy_boto3_bedrock_runtime")
            else Any,
            "voyageai": import_optional_dependency("voyageai") if is_dependency_
↪installed("voyageai") else Any,
            "Schema": Schema,
            "BaseModel": BaseModel,
            **types_override,
        },
    )
```

from_attrs_cls(attrs_cls, *, types_overrides=None, serializable_overrides=None)classmethod

Generate a Schema from an attrs class.

Parameters

Name	Type	Description	Default
attrs_cls	*type*	An attrs class. 	*required*
types_overrides	*Optional[dict[str, type]]*	A dictionary of types to override when resolving types. 	*None*
serializable_overrides	*Optional[dict[str, bool]]*	A dictionary of field names to whether they are serializable. 	*None*

Source Code in griptape/schemas/base_schema.py

```
@classmethod
def from_attrs_cls(
    cls,
    attrs_cls: type,
    *,
    types_overrides: Optional[dict[str, type]] = None,
    serializable_overrides: Optional[dict[str, bool]] = None,
) -> type:
    """Generate a Schema from an attrs class.

    Args:
        attrs_cls: An attrs class.
        types_overrides: A dictionary of types to override when resolving types.
        serializable_overrides: A dictionary of field names to whether they are␣
↪serializable.
    """
    from marshmallow import post_load

    if serializable_overrides is None:
```

(continues on next page)

(continued from previous page)

```
        serializable_overrides = {}

    class SubSchema(cls):
        @post_load
        def make_obj(self, data: Any, **kwargs) -> Any:
            # Map the serialized keys to their correct deserialization keys
            fields = attrs.fields_dict(attrs_cls)
            for key in list(data):
                if key in fields:
                    field = fields[key]
                    if field.metadata.get("deserialization_key"):
                        data[field.metadata["deserialization_key"]] = data.pop(key)

            return attrs_cls(**data)

    cls._resolve_types(attrs_cls, types_override=types_overrides)
    fields = {}
    for field in attrs.fields(attrs_cls):
        field_key = field.alias or field.name
        if serializable_overrides.get(field_key, field.metadata.get("serializable",␣
↪False)):
            fields[field_key] = cls._get_field_for_type(
                field.type,
                serialization_key=field.metadata.get("serialization_key"),
                types_overrides=types_overrides,
            )
    return SubSchema.from_dict(fields, name=f"{attrs_cls.__name__}Schema")
```

bytes_field

Bases:

Field

Source Code in griptape/schemas/bytes_field.py

```
class Bytes(fields.Field):
    def _serialize(self, value: Any, attr: Any, obj: Any, **kwargs) -> str:
        return base64.b64encode(value).decode()

    def _deserialize(self, value: Any, attr: Any, data: Any, **kwargs) -> bytes:
        return base64.b64decode(value)

    def _validate(self, value: Any) -> None:
        if not isinstance(value, bytes):
            raise ValidationError("Invalid input type.")
```

_deserialize(value, attr, data, **kwargs)

Source Code in griptape/schemas/bytes_field.py

```
def _deserialize(self, value: Any, attr: Any, data: Any, **kwargs) -> bytes:
    return base64.b64decode(value)
```

_serialize(value, attr, obj, **kwargs)

Source Code in griptape/schemas/bytes_field.py

```
def _serialize(self, value: Any, attr: Any, obj: Any, **kwargs) -> str:
    return base64.b64encode(value).decode()
```

_validate(value)

Source Code in griptape/schemas/bytes_field.py

```
def _validate(self, value: Any) -> None:
    if not isinstance(value, bytes):
        raise ValidationError("Invalid input type.")
```

polymorphic_schema

Bases: BaseSchema “)

Source Code in griptape/schemas/polymorphic_schema.py

```
class PolymorphicSchema(BaseSchema):
    """PolymorphicSchema is based on https://github.com/marshmallow-code/marshmallow-
↪oneofschema."""

    def __init__(self, inner_class: Any, types_overrides: Optional[dict[str, type]] =␣
↪None, **kwargs) -> None:
        super().__init__(**kwargs)

        self.inner_class = inner_class
        self.types_overrides = types_overrides

    type_field = "type"
    type_field_remove = True

    def get_obj_type(self, obj: Any) -> Any:
        """Returns name of the schema during dump() calls, given the object being␣
↪dumped."""
        return obj.__class__.__name__

    def get_data_type(self, data: Any) -> Any:
        """Returns name of the schema during load() calls, given the data being␣
↪loaded. Defaults to looking up `type_field` in the data."""
        data_type = data.get(self.type_field)
        if self.type_field in data and self.type_field_remove:
            data.pop(self.type_field)
        return data_type

    def dump(self, obj: Any, *, many: Any = None, **kwargs) -> Any:
        errors = {}
        result_data = []
        result_errors = {}
        many = self.many if many is None else bool(many)
```

(continues on next page)

(continued from previous page)

```
        if not many:
            result = result_data = self._dump(obj, **kwargs)
        else:
            for idx, o in enumerate(obj):
                try:
                    result = self._dump(o, **kwargs)
                    result_data.append(result)
                except ValidationError as error:
                    result_errors[idx] = error.normalized_messages()
                    result_data.append(error.valid_data)

        result = result_data
        errors = result_errors

        if not errors:
            return result
        exc = ValidationError(errors, data=obj, valid_data=result)  # pyright:␣
↪ignore[reportArgumentType]
        raise exc

    def _dump(self, obj: Any, *, update_fields: bool = True, **kwargs) -> Any:
        obj_type = self.get_obj_type(obj)

        if not obj_type:
            return (None, {"_schema": f"Unknown object class: {obj.__class__.__name__}
↪"})

        type_schema = BaseSchema.from_attrs_cls(obj.__class__, types_overrides=self.
↪types_overrides)

        if not type_schema:
            return None, {"_schema": f"Unsupported object type: {obj_type}"}

        schema = type_schema if isinstance(type_schema, Schema) else type_schema()

        schema.context.update(getattr(self, "context", {}))

        result = schema.dump(obj, many=False, **kwargs)

        if result is not None:
            result[self.type_field] = obj_type  # pyright: ignore[reportArgumentType,
↪reportCallIssue]

        return result

    def load(self, data: Any, *, many: Any = None, partial: Any = None, unknown: Any␣
↪= None, **kwargs) -> Any:
        errors = {}
        result_data = []
        result_errors = {}
        many = self.many if many is None else bool(many)
        if partial is None:
```

(continues on next page)

(continued from previous page)

```
            partial = self.partial
        if not many:
            try:
                result = result_data = self._load(data, partial=partial,␣
↪unknown=unknown, **kwargs)
            except ValidationError as error:
                result_errors = error.normalized_messages()
                result_data.append(error.valid_data)
        else:
            for idx, item in enumerate(data):
                try:
                    result = self._load(item, partial=partial, **kwargs)
                    result_data.append(result)
                except ValidationError as error:
                    result_errors[idx] = error.normalized_messages()
                    result_data.append(error.valid_data)

        result = result_data
        errors = result_errors

        if not errors:
            return result
        exc = ValidationError(errors, data=data, valid_data=result)
        raise exc

    def _load(self, data: Any, *, partial: Any = None, unknown: Any = None, **kwargs)␣
↪-> Any:
        if not isinstance(data, dict):
            raise ValidationError({"_schema": f"Invalid data type: {data}"})

        data = dict(data)
        unknown = unknown or self.unknown
        data_type = self.get_data_type(data)

        if data_type is None:
            raise ValidationError({self.type_field: ["Missing data for required field.
↪"]})

        type_schema = self.inner_class.get_schema(data_type, module_name=data.get(
↪"module_name"))
        if not type_schema:
            raise ValidationError({self.type_field: [f"Unsupported value: {data_type}
↪"]})

        schema = type_schema if isinstance(type_schema, Schema) else type_schema()

        schema.context.update(getattr(self, "context", {}))

        return schema.load(data, many=False, partial=partial, unknown=unknown,␣
↪**kwargs)

    def validate(self, data: Any, *, many: Any = None, partial: Any = None) -> Any:
```

(continues on next page)

(continued from previous page)

```
↪# pyright: ignore[reportIncompatibleMethodOverride]
        try:
            self.load(data, many=many, partial=partial)
        except ValidationError as ve:
            return ve.messages
        return {}
```

- `inner_class = inner_class` instance-attribute
- `type_field = 'type'` class-attribute instance-attribute
- `type_field_remove = True` class-attribute instance-attribute
- `types_overrides = types_overrides` instance-attribute

init(inner_class, types_overrides=None, **kwargs)

Source Code in griptape/schemas/polymorphic_schema.py

```
def __init__(self, inner_class: Any, types_overrides: Optional[dict[str, type]] =␣
↪None, **kwargs) -> None:
    super().__init__(**kwargs)

    self.inner_class = inner_class
    self.types_overrides = types_overrides
```

_dump(obj, *, update_fields=True, **kwargs)

Source Code in griptape/schemas/polymorphic_schema.py

```
def _dump(self, obj: Any, *, update_fields: bool = True, **kwargs) -> Any:
    obj_type = self.get_obj_type(obj)

    if not obj_type:
        return (None, {"_schema": f"Unknown object class: {obj.__class__.__name__}"})

    type_schema = BaseSchema.from_attrs_cls(obj.__class__, types_overrides=self.types_
↪overrides)

    if not type_schema:
        return None, {"_schema": f"Unsupported object type: {obj_type}"}

    schema = type_schema if isinstance(type_schema, Schema) else type_schema()

    schema.context.update(getattr(self, "context", {}))

    result = schema.dump(obj, many=False, **kwargs)

    if result is not None:
        result[self.type_field] = obj_type  # pyright: ignore[reportArgumentType,
↪reportCallIssue]

    return result
```

_load(data, *, partial=None, unknown=None, **kwargs)

Source Code in griptape/schemas/polymorphic_schema.py

```
def _load(self, data: Any, *, partial: Any = None, unknown: Any = None, **kwargs) ->␣
↪Any:
    if not isinstance(data, dict):
        raise ValidationError({"_schema": f"Invalid data type: {data}"})

    data = dict(data)
    unknown = unknown or self.unknown
    data_type = self.get_data_type(data)

    if data_type is None:
        raise ValidationError({self.type_field: ["Missing data for required field."]})

    type_schema = self.inner_class.get_schema(data_type, module_name=data.get("module_
↪name"))
    if not type_schema:
        raise ValidationError({self.type_field: [f"Unsupported value: {data_type}"]})

    schema = type_schema if isinstance(type_schema, Schema) else type_schema()

    schema.context.update(getattr(self, "context", {}))

    return schema.load(data, many=False, partial=partial, unknown=unknown, **kwargs)
```

dump(obj, *, many=None, **kwargs)

Source Code in griptape/schemas/polymorphic_schema.py

```
def dump(self, obj: Any, *, many: Any = None, **kwargs) -> Any:
    errors = {}
    result_data = []
    result_errors = {}
    many = self.many if many is None else bool(many)
    if not many:
        result = result_data = self._dump(obj, **kwargs)
    else:
        for idx, o in enumerate(obj):
            try:
                result = self._dump(o, **kwargs)
                result_data.append(result)
            except ValidationError as error:
                result_errors[idx] = error.normalized_messages()
                result_data.append(error.valid_data)

    result = result_data
    errors = result_errors

    if not errors:
        return result
    exc = ValidationError(errors, data=obj, valid_data=result)  # pyright:␣
```

(continues on next page)

(continued from previous page)

```
↪ignore[reportArgumentType]
    raise exc
```

get_data_type(data)

Source Code in griptape/schemas/polymorphic_schema.py

```
def get_data_type(self, data: Any) -> Any:
    """Returns name of the schema during load() calls, given the data being loaded.␣
↪Defaults to looking up `type_field` in the data."""
    data_type = data.get(self.type_field)
    if self.type_field in data and self.type_field_remove:
        data.pop(self.type_field)
    return data_type
```

get_obj_type(obj)

Source Code in griptape/schemas/polymorphic_schema.py

```
def get_obj_type(self, obj: Any) -> Any:
    """Returns name of the schema during dump() calls, given the object being dumped."
↪""
    return obj.__class__.__name__
```

load(data, *, many=None, partial=None, unknown=None, **kwargs)

Source Code in griptape/schemas/polymorphic_schema.py

```
def load(self, data: Any, *, many: Any = None, partial: Any = None, unknown: Any =␣
↪None, **kwargs) -> Any:
    errors = {}
    result_data = []
    result_errors = {}
    many = self.many if many is None else bool(many)
    if partial is None:
        partial = self.partial
    if not many:
        try:
            result = result_data = self._load(data, partial=partial, unknown=unknown,␣
↪**kwargs)
        except ValidationError as error:
            result_errors = error.normalized_messages()
            result_data.append(error.valid_data)
    else:
        for idx, item in enumerate(data):
            try:
                result = self._load(item, partial=partial, **kwargs)
                result_data.append(result)
            except ValidationError as error:
                result_errors[idx] = error.normalized_messages()
                result_data.append(error.valid_data)
```

(continues on next page)

(continued from previous page)

```
    result = result_data
    errors = result_errors

    if not errors:
        return result
    exc = ValidationError(errors, data=data, valid_data=result)
    raise exc
```

validate(data, *, many=None, partial=None)

Source Code in griptape/schemas/polymorphic_schema.py

```
def validate(self, data: Any, *, many: Any = None, partial: Any = None) -> Any:  #␣
↪pyright: ignore[reportIncompatibleMethodOverride]
    try:
        self.load(data, many=many, partial=partial)
    except ValidationError as ve:
        return ve.messages
    return {}
```

pydantic_model_field

Bases:

Field

Source Code in griptape/schemas/pydantic_model_field.py

```
class PydanticModel(fields.Field):
    def _serialize(self, value: Optional[BaseModel], attr: Any, obj: Any, **kwargs) ->
↪ Optional[dict]:
        if value is None:
            return None
        return value.model_dump()

    def _deserialize(self, value: dict, attr: Any, data: Any, **kwargs) -> BaseModel:
        return RootModel(value)
```

_deserialize(value, attr, data, **kwargs)

Source Code in griptape/schemas/pydantic_model_field.py

```
def _deserialize(self, value: dict, attr: Any, data: Any, **kwargs) -> BaseModel:
    return RootModel(value)
```

_serialize(value, attr, obj, **kwargs)

Source Code in griptape/schemas/pydantic_model_field.py

```
def _serialize(self, value: Optional[BaseModel], attr: Any, obj: Any, **kwargs) ->␣
↪Optional[dict]:
```

(continues on next page)

(continued from previous page)

```
    if value is None:
        return None
    return value.model_dump()
```

union_field

Bases: *MarshmallowUnionError* “)

Source Code in griptape/schemas/union_field.py

```
class ExceptionGroupError(MarshmallowUnionError):
    """Collection of possibly multiple exceptions."""

    def __init__(self, msg: str, errors: Any) -> None:
        self.msg = msg
        self.errors = errors
        super().__init__(msg, errors)
```

- `errors = errors` instance-attribute
- `msg = msg` instance-attribute

init(msg, errors)

Source Code in griptape/schemas/union_field.py

```
def __init__(self, msg: str, errors: Any) -> None:
    self.msg = msg
    self.errors = errors
    super().__init__(msg, errors)
```

MarshmallowUnionError

Bases:

Exception

Source Code in griptape/schemas/union_field.py

```
class MarshmallowUnionError(Exception):
    """Base error for marshmallow_union."""
```

Union

Bases:

Field

Parameters

Name	Type	Description	Default
fields	<a href="../../data/artifacts.mdx#list" title="List (list)"><code class="language-pythc	The list of candidate fields to try. 	*required*
reverse_serialize_candidates	*bool*	Whether to try the candidates in reverse order when serializing. 	*False*

Source Code in griptape/schemas/union_field.py

```
class Union(marshmallow.fields.Field):
    """Field that accepts any one of multiple fields.

    Source: https://github.com/adamboche/python-marshmallow-union
    Each argument will be tried until one succeeds.

    Args:
        fields: The list of candidate fields to try.
        reverse_serialize_candidates: Whether to try the candidates in reverse order␣
↪when serializing.
    """

    def __init__(
        self,
        fields: list[marshmallow.fields.Field],
        *,
        reverse_serialize_candidates: bool = False,
        **kwargs: Any,
    ) -> None:
        self._candidate_fields = fields
        self._reverse_serialize_candidates = reverse_serialize_candidates
        super().__init__(**kwargs)

    def _serialize(self, value: Any, attr: str | None, obj: str, **kwargs: Any) ->␣
↪Any:
        """Pulls the value for the given key from the object, applies the fields␣
↪formatting and returns the result.

        Args:
            value: The value to be serialized.
            attr: The attribute or key to get from the object.
            obj: The object to pull the key from.
            kwargs: Field-specific keyword arguments.

        Raises:
            marshmallow.exceptions.ValidationError: In case of formatting problem
        """
        error_store = kwargs.pop("error_store", marshmallow.error_store.ErrorStore())
```

(continues on next page)

(continued from previous page)

```
        fields = (
            list(reversed(self._candidate_fields)) if self._reverse_serialize_
↪candidates else self._candidate_fields
        )

        for candidate_field in fields:
            try:
                # pylint: disable=protected-access
                return candidate_field._serialize(value, attr, obj, error_store=error_
↪store, **kwargs)
            except (TypeError, ValueError) as e:
                error_store.store_error({attr: str(e)})

        raise ExceptionGroupError("All serializers raised exceptions.", error_store.
↪errors)

    def _deserialize(self, value: Any, attr: str | None = None, data: Any = None,␣
↪**kwargs: Any) -> Any:
        """Deserialize ``value``.

        Args:
            value: The value to be deserialized.
            attr: The attribute/key in `data` to be deserialized.
            data: The raw input data passed to the `Schema.load`.
            kwargs: Field-specific keyword arguments.

        Raises:
            ValidationError: If an invalid value is passed or if a required value is␣
↪missing.
        """
        errors = []
        for candidate_field in self._candidate_fields:
            try:
                return candidate_field.deserialize(value, attr, data, **kwargs)
            except marshmallow.exceptions.ValidationError as exc:
                errors.append(exc.messages)

        raise marshmallow.exceptions.ValidationError(message=errors, field_name=attr␣
↪or "")
```

- `_candidate_fields = fields` instance-attribute
- `_reverse_serialize_candidates = reverse_serialize_candidates` instance-attribute

init(fields, *, reverse_serialize_candidates=False, **kwargs)

Source Code in griptape/schemas/union_field.py

```
def __init__(
    self,
    fields: list[marshmallow.fields.Field],
    *,
    reverse_serialize_candidates: bool = False,
```

(continues on next page)

(continued from previous page)

```
    **kwargs: Any,
) -> None:
    self._candidate_fields = fields
    self._reverse_serialize_candidates = reverse_serialize_candidates
    super().__init__(**kwargs)
```

_deserialize(value, attr=None, data=None, **kwargs)

Deserialize `value` .

Parameters

Name	Type	Description	Default
value	*Any*	The value to be deserialized. 	*required*
attr	*str I None*	The attribute/key in *data* to be deserialized. 	*None*
data	*Any*	The raw input data passed to the *Schema.load*. 	*None*
kwargs	*Any*	Field-specific keyword arguments. 	*{}*

Raises

Source Code in griptape/schemas/union_field.py

```
def _deserialize(self, value: Any, attr: str | None = None, data: Any = None,␣
↪**kwargs: Any) -> Any:
    """Deserialize ``value``.

    Args:
        value: The value to be deserialized.
        attr: The attribute/key in `data` to be deserialized.
        data: The raw input data passed to the `Schema.load`.
        kwargs: Field-specific keyword arguments.

    Raises:
        ValidationError: If an invalid value is passed or if a required value is␣
↪missing.
    """
    errors = []
    for candidate_field in self._candidate_fields:
        try:
            return candidate_field.deserialize(value, attr, data, **kwargs)
        except marshmallow.exceptions.ValidationError as exc:
            errors.append(exc.messages)
```

(continues on next page)

(continued from previous page)

```
    raise marshmallow.exceptions.ValidationError(message=errors, field_name=attr or "
↪")
```

_serialize(value, attr, obj, **kwargs)

Pulls the value for the given key from the object, applies the field's formatting and returns the result.

Parameters

Name	Type	Description	Default
value	*Any*	The value to be serialized. 	*required*
attr	*str I None*	The attribute or key to get from the object. 	*required*
obj	*str*	The object to pull the key from. 	*required*
kwargs	*Any*	Field-specific keyword arguments. 	*{}*

Raises

Source Code in griptape/schemas/union_field.py

```
def _serialize(self, value: Any, attr: str | None, obj: str, **kwargs: Any) -> Any:
    """Pulls the value for the given key from the object, applies the fields␣
↪formatting and returns the result.

    Args:
        value: The value to be serialized.
        attr: The attribute or key to get from the object.
        obj: The object to pull the key from.
        kwargs: Field-specific keyword arguments.

    Raises:
        marshmallow.exceptions.ValidationError: In case of formatting problem
    """
    error_store = kwargs.pop("error_store", marshmallow.error_store.ErrorStore())
    fields = (
        list(reversed(self._candidate_fields)) if self._reverse_serialize_candidates␣
↪else self._candidate_fields
    )

    for candidate_field in fields:
        try:
            # pylint: disable=protected-access
            return candidate_field._serialize(value, attr, obj, error_store=error_
↪store, **kwargs)
        except (TypeError, ValueError) as e:
```

(continues on next page)

(continued from previous page)

```
            error_store.store_error({attr: str(e)})

    raise ExceptionGroupError("All serializers raised exceptions.", error_store.
↪errors)
```

11.5.17 structures

- `__all__ = ['Agent', 'Pipeline', 'Structure', 'Workflow']` module-attribute

Bases: *Override Default Structure Embedding Driver* “)

Source Code in griptape/structures/agent.py

```
@define
class Agent(Structure):
    input: Union[str, list, tuple, BaseArtifact, Callable[[BaseTask], BaseArtifact]]␣
↪= field(
        default=lambda task: task.full_context["args"][0] if task.full_context["args
↪"] else TextArtifact(value=""),
    )
    stream: Optional[bool] = field(default=None, kw_only=True)
    prompt_driver: Optional[BasePromptDriver] = field(default=None, kw_only=True)
    output_schema: Optional[Union[Schema, type[BaseModel]]] = field(default=None, kw_
↪only=True)
    tools: list[BaseTool] = field(factory=list, kw_only=True)
    max_meta_memory_entries: Optional[int] = field(default=20, kw_only=True)
    fail_fast: bool = field(default=False, kw_only=True)
    _tasks: list[Union[BaseTask, list[BaseTask]]] = field(
        factory=list, kw_only=True, alias="tasks", metadata={"serializable": True}
    )

    @fail_fast.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_fail_fast(self, _: Attribute, fail_fast: bool) -> None:  # noqa:␣
↪FBT001
        if fail_fast:
            raise ValueError("Agents cannot fail fast, as they can only have 1 task.")

    @prompt_driver.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_prompt_driver(self, _: Attribute, prompt_driver:␣
↪Optional[BasePromptDriver]) -> None:
        if prompt_driver is not None and self.stream is not None:
            warnings.warn(
                "`Agent.prompt_driver` is set, but `Agent.stream` was provided.␣
↪`Agent.stream` will be ignored. This will be an error in the future.",
                UserWarning,
                stacklevel=2,
            )

    @_tasks.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_tasks(self, _: Attribute, tasks: list) -> None:
        if tasks and self.prompt_driver is not None:
            warnings.warn(
```

(continues on next page)

(continued from previous page)

```
                "`Agent.tasks` is set, but `Agent.prompt_driver` was provided. `Agent.
↪prompt_driver` will be ignored. This will be an error in the future.",
                UserWarning,
                stacklevel=2,
            )

    def __attrs_post_init__(self) -> None:
        super().__attrs_post_init__()

        if len(self.tasks) == 0:
            self._init_task()

    @property
    def task(self) -> BaseTask:
        return self.tasks[0]

    def add_task(self, task: BaseTask) -> BaseTask:
        self._tasks.clear()

        task.preprocess(self)

        self._tasks.append(task)

        return task

    def add_tasks(self, *tasks: BaseTask | list[BaseTask]) -> list[BaseTask]:
        if len(tasks) > 1:
            raise ValueError("Agents can only have one task.")
        return super().add_tasks(*tasks)

    @observable
    def try_run(self, *args) -> Agent:
        self.task.run()

        return self

    def _init_task(self) -> None:
        if self.stream is None:
            with validators.disabled():
                self.stream = Defaults.drivers_config.prompt_driver.stream

        if self.prompt_driver is None:
            with validators.disabled():
                prompt_driver = evolve(Defaults.drivers_config.prompt_driver,␣
↪stream=self.stream)
                self.prompt_driver = prompt_driver
        else:
            prompt_driver = self.prompt_driver

        task = PromptTask(
            self.input,
            prompt_driver=prompt_driver,
```

(continues on next page)

(continued from previous page)

```
            tools=self.tools,
            output_schema=self.output_schema,
            max_meta_memory_entries=self.max_meta_memory_entries,
        )

        self.add_task(task)
```

- `_tasks = field(factory=list, kw_only=True, alias='tasks', metadata={'serializable': True})` class-attribute instance-attribute
- `fail_fast = field(default=False, kw_only=True)` class-attribute instance-attribute
- `input = field(default=lambda task: task.full_context['args'][0] if task.full_context['args'] else TextArtifact(value=''))` class-attribute instance-attribute
- `max_meta_memory_entries = field(default=20, kw_only=True)` class-attribute instance-attribute
- `output_schema = field(default=None, kw_only=True)` class-attribute instance-attribute
- `prompt_driver = field(default=None, kw_only=True)` class-attribute instance-attribute
- `stream = field(default=None, kw_only=True)` class-attribute instance-attribute
- `task` property
- `tools = field(factory=list, kw_only=True)` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/structures/agent.py

```
def __attrs_post_init__(self) -> None:
    super().__attrs_post_init__()

    if len(self.tasks) == 0:
        self._init_task()
```

_init_task()

Source Code in griptape/structures/agent.py

```
def _init_task(self) -> None:
    if self.stream is None:
        with validators.disabled():
            self.stream = Defaults.drivers_config.prompt_driver.stream

    if self.prompt_driver is None:
        with validators.disabled():
            prompt_driver = evolve(Defaults.drivers_config.prompt_driver, stream=self.
↪stream)
            self.prompt_driver = prompt_driver
    else:
        prompt_driver = self.prompt_driver

    task = PromptTask(
        self.input,
```

(continues on next page)

(continued from previous page)

```
        prompt_driver=prompt_driver,
        tools=self.tools,
        output_schema=self.output_schema,
        max_meta_memory_entries=self.max_meta_memory_entries,
    )

    self.add_task(task)
```

add_task(task)

Source Code in griptape/structures/agent.py

```
def add_task(self, task: BaseTask) -> BaseTask:
    self._tasks.clear()

    task.preprocess(self)

    self._tasks.append(task)

    return task
```

add_tasks(*tasks)

Source Code in griptape/structures/agent.py

```
def add_tasks(self, *tasks: BaseTask | list[BaseTask]) -> list[BaseTask]:
    if len(tasks) > 1:
        raise ValueError("Agents can only have one task.")
    return super().add_tasks(*tasks)
```

try_run(*args)

Source Code in griptape/structures/agent.py

```
@observable
def try_run(self, *args) -> Agent:
    self.task.run()

    return self
```

validate*fail_fast*(, fail_fast)

Source Code in griptape/structures/agent.py

```
@fail_fast.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_fail_fast(self, _: Attribute, fail_fast: bool) -> None:  # noqa: FBT001
    if fail_fast:
        raise ValueError("Agents cannot fail fast, as they can only have 1 task.")
```

validate*prompt_driver(*, prompt_driver)

Source Code in griptape/structures/agent.py

```
@prompt_driver.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_prompt_driver(self, _: Attribute, prompt_driver:␣
↪Optional[BasePromptDriver]) -> None:
    if prompt_driver is not None and self.stream is not None:
        warnings.warn(
            "`Agent.prompt_driver` is set, but `Agent.stream` was provided. `Agent.
↪stream` will be ignored. This will be an error in the future.",
            UserWarning,
            stacklevel=2,
        )
```

validate*tasks(*, tasks)

Source Code in griptape/structures/agent.py

```
@_tasks.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_tasks(self, _: Attribute, tasks: list) -> None:
    if tasks and self.prompt_driver is not None:
        warnings.warn(
            "`Agent.tasks` is set, but `Agent.prompt_driver` was provided. `Agent.
↪prompt_driver` will be ignored. This will be an error in the future.",
            UserWarning,
            stacklevel=2,
        )
```

Pipeline

Bases: *Override Default Structure Embedding Driver* “)

Source Code in griptape/structures/pipeline.py

```
@define
class Pipeline(Structure):
    def add_task(self, task: BaseTask) -> BaseTask:
        if (existing_task := self.try_find_task(task.id)) is not None:
            return existing_task

        task.preprocess(self)

        if self.output_task:
            self.output_task.child_ids.append(task.id)
            task.parent_ids.append(self.output_task.id)

        self._tasks.append(task)

        return task

    def insert_task(self, parent_task: BaseTask, task: BaseTask) -> BaseTask:
        task.preprocess(self)
```

(continues on next page)

(continued from previous page)

```
        if parent_task.children:
            child_task = parent_task.children[0]

            task.child_ids.append(child_task.id)
            child_task.parent_ids.append(task.id)

            child_task.parent_ids.remove(parent_task.id)
            parent_task.child_ids.remove(child_task.id)

        task.parent_ids.append(parent_task.id)
        parent_task.child_ids.append(task.id)

        parent_index = self.tasks.index(parent_task)
        self._tasks.insert(parent_index + 1, task)

        return task

    @observable
    def try_run(self, *args) -> Pipeline:
        self.__run_from_task(self.input_task)

        return self

    def context(self, task: BaseTask) -> dict[str, Any]:
        context = super().context(task)

        context.update(
            {
                "parent_output": task.parents[0].output if task.parents else None,
                "task_outputs": self.task_outputs,
                "parent": task.parents[0] if task.parents else None,
                "child": task.children[0] if task.children else None,
            },
        )

        return context

    def __run_from_task(self, task: Optional[BaseTask]) -> None:
        if task is None or isinstance(task.run(), ErrorArtifact) and self.fail_fast:
            return
        self.__run_from_task(next(iter(task.children), None))
```

__run_from_task(task)

Source Code in griptape/structures/pipeline.py

```
def __run_from_task(self, task: Optional[BaseTask]) -> None:
    if task is None or isinstance(task.run(), ErrorArtifact) and self.fail_fast:
        return
    self.__run_from_task(next(iter(task.children), None))
```

add_task(task)

Source Code in griptape/structures/pipeline.py

```
def add_task(self, task: BaseTask) -> BaseTask:
    if (existing_task := self.try_find_task(task.id)) is not None:
        return existing_task

    task.preprocess(self)

    if self.output_task:
        self.output_task.child_ids.append(task.id)
        task.parent_ids.append(self.output_task.id)

    self._tasks.append(task)

    return task
```

context(task)

Source Code in griptape/structures/pipeline.py

```
def context(self, task: BaseTask) -> dict[str, Any]:
    context = super().context(task)

    context.update(
        {
            "parent_output": task.parents[0].output if task.parents else None,
            "task_outputs": self.task_outputs,
            "parent": task.parents[0] if task.parents else None,
            "child": task.children[0] if task.children else None,
        },
    )

    return context
```

insert_task(parent_task, task)

Source Code in griptape/structures/pipeline.py

```
def insert_task(self, parent_task: BaseTask, task: BaseTask) -> BaseTask:
    task.preprocess(self)

    if parent_task.children:
        child_task = parent_task.children[0]

        task.child_ids.append(child_task.id)
        child_task.parent_ids.append(task.id)

        child_task.parent_ids.remove(parent_task.id)
        parent_task.child_ids.remove(child_task.id)

    task.parent_ids.append(parent_task.id)
    parent_task.child_ids.append(task.id)
```

(continues on next page)

(continued from previous page)

```
    parent_index = self.tasks.index(parent_task)
    self._tasks.insert(parent_index + 1, task)

    return task
```

try_run(*args)

Source Code in griptape/structures/pipeline.py

```
@observable
def try_run(self, *args) -> Pipeline:
    self.__run_from_task(self.input_task)

    return self
```

Structure

Bases: *RuleMixin* “) , *SerializableMixin* ”) , *RunnableMixin['Structure'* “) , ABC

Source Code in griptape/structures/structure.py

```
@define
class Structure(RuleMixin, SerializableMixin, RunnableMixin["Structure"], ABC):
    id: str = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata=
↪{"serializable": True})
    _tasks: list[Union[BaseTask, list[BaseTask]]] = field(
        factory=list, kw_only=True, alias="tasks", metadata={"serializable": True}
    )
    conversation_memory: Optional[BaseConversationMemory] = field(
        default=Factory(lambda: ConversationMemory()),
        kw_only=True,
        metadata={"serializable": True},
    )
    conversation_memory_strategy: Literal["per_structure", "per_task"] = field(
        default="per_structure", kw_only=True, metadata={"serializable": True}
    )
    task_memory: TaskMemory = field(
        default=Factory(lambda self: TaskMemory(), takes_self=True),
        kw_only=True,
    )
    meta_memory: MetaMemory = field(default=Factory(lambda: MetaMemory()), kw_
↪only=True)
    fail_fast: bool = field(default=True, kw_only=True, metadata={"serializable":␣
↪True})
    _execution_args: tuple = ()
    _event_queue: Queue[BaseEvent] = field(default=Factory(lambda: Queue()),␣
↪init=False)

    def __attrs_post_init__(self) -> None:
        tasks = self._tasks.copy()
        self._tasks.clear()
```

(continues on next page)

(continued from previous page)

```
        self.add_tasks(*tasks)

    def __add__(self, other: BaseTask | list[BaseTask | list[BaseTask]]) ->␣
↪list[BaseTask]:
        return self.add_tasks(*other) if isinstance(other, list) else self.add_
↪tasks(other)

    @property
    def tasks(self) -> list[BaseTask]:
        tasks = []

        for task in self._tasks:
            if isinstance(task, list):
                tasks.extend(task)
            else:
                tasks.append(task)
        return tasks

    @property
    def execution_args(self) -> tuple:
        return self._execution_args

    @property
    def input_task(self) -> Optional[BaseTask]:
        return self.tasks[0] if self.tasks else None

    @property
    def output_task(self) -> Optional[BaseTask]:
        return self.tasks[-1] if self.tasks else None

    @property
    def output(self) -> BaseArtifact:
        if self.output_task is None:
            raise ValueError("Structure has no output Task. Add a Task to the␣
↪Structure to generate output.")
        if self.output_task.output is None:
            raise ValueError("Structures output Task has no output. Run the Structure␣
↪to generate output.")
        return self.output_task.output

    @property
    def task_outputs(self) -> dict[str, Optional[BaseArtifact]]:
        return {task.id: task.output for task in self.tasks}

    @property
    def finished_tasks(self) -> list[BaseTask]:
        return [s for s in self.tasks if s.is_finished()]

    def is_finished(self) -> bool:
        return all(not s.can_run() for s in self.tasks)

    def is_running(self) -> bool:
```

(continues on next page)

(continued from previous page)

```
        return any(s for s in self.tasks if s.is_running())

    def find_task(self, task_id: str) -> BaseTask:
        if (task := self.try_find_task(task_id)) is not None:
            return task
        raise ValueError(f"Task with id {task_id} doesnt exist.")

    def try_find_task(self, task_id: str) -> Optional[BaseTask]:
        for task in self.tasks:
            if task.id == task_id:
                return task
        return None

    def add_tasks(self, *tasks: BaseTask | list[BaseTask]) -> list[BaseTask]:
        added_tasks = []
        for task in tasks:
            if isinstance(task, list):
                added_tasks.extend(self.add_tasks(*task))
            else:
                added_tasks.append(self.add_task(task))
        return added_tasks

    def context(self, task: BaseTask) -> dict[str, Any]:
        return {"args": self.execution_args, "structure": self}

    def resolve_relationships(self) -> None:
        task_by_id = {}
        for task in self.tasks:
            if task.id in task_by_id:
                raise ValueError(f"Duplicate task with id {task.id} found.")
            task_by_id[task.id] = task

        for task in self.tasks:
            # Ensure parents include this task as a child
            for parent_id in task.parent_ids:
                if parent_id not in task_by_id:
                    raise ValueError(f"Task with id {parent_id} doesnt exist.")
                parent = task_by_id[parent_id]
                if task.id not in parent.child_ids:
                    parent.child_ids.append(task.id)

            # Ensure children include this task as a parent
            for child_id in task.child_ids:
                if child_id not in task_by_id:
                    raise ValueError(f"Task with id {child_id} doesnt exist.")
                child = task_by_id[child_id]
                if task.id not in child.parent_ids:
                    child.parent_ids.append(task.id)

    @observable
    def before_run(self, args: Any) -> None:
        super().before_run(args)
```

(continues on next page)

(continued from previous page)

```
        self._execution_args = args

        [task.reset() for task in self.tasks]

        if self.input_task is not None:
            EventBus.publish_event(
                StartStructureRunEvent(
                    structure_id=self.id,
                    input_task_input=self.input_task.input,
                    input_task_output=self.input_task.output,
                ),
            )

        self.resolve_relationships()

    @observable
    def after_run(self) -> None:
        super().after_run()

        if self.output_task is not None:
            if (
                self.conversation_memory_strategy == "per_structure"
                and self.conversation_memory is not None
                and self.input_task is not None
                and self.output_task.output is not None
            ):
                run = Run(input=self.input_task.input, output=self.output_task.output)

                self.conversation_memory.add_run(run)

            EventBus.publish_event(
                FinishStructureRunEvent(
                    structure_id=self.id,
                    output_task_input=self.output_task.input,
                    output_task_output=self.output_task.output,
                ),
                flush=True,
            )

    @abstractmethod
    def add_task(self, task: BaseTask) -> BaseTask: ...

    @observable
    def run(self, *args) -> Structure:
        self.before_run(args)

        result = self.try_run(*args)

        self.after_run()

        return result
```

(continues on next page)

(continued from previous page)

```
    @observable
    def run_stream(self, *args, event_types: Optional[list[type[BaseEvent]]] = None) -
↪> Iterator[BaseEvent]:
        if event_types is None:
            event_types = [BaseEvent]
        elif FinishStructureRunEvent not in event_types:
            event_types = [*event_types, FinishStructureRunEvent]

        with EventListener(self._event_queue.put, event_types=event_types):
            t = Thread(target=with_contextvars(self.run), args=args)
            t.start()

            while True:
                event = self._event_queue.get()
                if isinstance(event, FinishStructureRunEvent) and event.structure_id␣
↪== self.id:
                    break
                else:
                    yield event
            t.join()

    @abstractmethod
    def try_run(self, *args) -> Structure: ...
```

- `_event_queue = field(default=Factory(lambda: Queue()), init=False)` class-attribute instance-attribute
- `_execution_args = ()` class-attribute instance-attribute
- `_tasks = field(factory=list, kw_only=True, alias='tasks', metadata={'serializable': True})` class-attribute instance-attribute
- `conversation_memory = field(default=Factory(lambda: ConversationMemory()), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `conversation_memory_strategy = field(default='per_structure', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `execution_args` property
- `fail_fast = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `finished_tasks` property
- `id = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `input_task` property
- `meta_memory = field(default=Factory(lambda: MetaMemory()), kw_only=True)` class-attribute instance-attribute
- `output` property
- `output_task` property
- `task_memory = field(default=Factory(lambda self: TaskMemory(), takes_self=True), kw_only=True)` class-attribute instance-attribute

- `task_outputs` property
- `tasks` property

add(other)

Source Code in griptape/structures/structure.py

```
def __add__(self, other: BaseTask | list[BaseTask | list[BaseTask]]) ->␣
↪list[BaseTask]:
    return self.add_tasks(*other) if isinstance(other, list) else self.add_
↪tasks(other)
```

attrs_post_init()

Source Code in griptape/structures/structure.py

```
def __attrs_post_init__(self) -> None:
    tasks = self._tasks.copy()
    self._tasks.clear()
    self.add_tasks(*tasks)
```

add_task(task)abstractmethod

Source Code in griptape/structures/structure.py

```
@abstractmethod
def add_task(self, task: BaseTask) -> BaseTask: ...
```

add_tasks(*tasks)

Source Code in griptape/structures/structure.py

```
def add_tasks(self, *tasks: BaseTask | list[BaseTask]) -> list[BaseTask]:
    added_tasks = []
    for task in tasks:
        if isinstance(task, list):
            added_tasks.extend(self.add_tasks(*task))
        else:
            added_tasks.append(self.add_task(task))
    return added_tasks
```

after_run()

Source Code in griptape/structures/structure.py

```
@observable
def after_run(self) -> None:
    super().after_run()

    if self.output_task is not None:
        if (
            self.conversation_memory_strategy == "per_structure"
            and self.conversation_memory is not None
```

(continues on next page)

(continued from previous page)

```
            and self.input_task is not None
            and self.output_task.output is not None
        ):
            run = Run(input=self.input_task.input, output=self.output_task.output)

            self.conversation_memory.add_run(run)

        EventBus.publish_event(
            FinishStructureRunEvent(
                structure_id=self.id,
                output_task_input=self.output_task.input,
                output_task_output=self.output_task.output,
            ),
            flush=True,
        )
```

before_run(args)

Source Code in griptape/structures/structure.py

```
@observable
def before_run(self, args: Any) -> None:
    super().before_run(args)
    self._execution_args = args

    [task.reset() for task in self.tasks]

    if self.input_task is not None:
        EventBus.publish_event(
            StartStructureRunEvent(
                structure_id=self.id,
                input_task_input=self.input_task.input,
                input_task_output=self.input_task.output,
            ),
        )

    self.resolve_relationships()
```

context(task)

Source Code in griptape/structures/structure.py

```
def context(self, task: BaseTask) -> dict[str, Any]:
    return {"args": self.execution_args, "structure": self}
```

find_task(task_id)

Source Code in griptape/structures/structure.py

```
def find_task(self, task_id: str) -> BaseTask:
    if (task := self.try_find_task(task_id)) is not None:
```

(continues on next page)

(continued from previous page)

```
            return task
        raise ValueError(f"Task with id {task_id} doesnt exist.")
```

is_finished()

Source Code in griptape/structures/structure.py

```
def is_finished(self) -> bool:
    return all(not s.can_run() for s in self.tasks)
```

is_running()

Source Code in griptape/structures/structure.py

```
def is_running(self) -> bool:
    return any(s for s in self.tasks if s.is_running())
```

resolve_relationships()

Source Code in griptape/structures/structure.py

```
def resolve_relationships(self) -> None:
    task_by_id = {}
    for task in self.tasks:
        if task.id in task_by_id:
            raise ValueError(f"Duplicate task with id {task.id} found.")
        task_by_id[task.id] = task

    for task in self.tasks:
        # Ensure parents include this task as a child
        for parent_id in task.parent_ids:
            if parent_id not in task_by_id:
                raise ValueError(f"Task with id {parent_id} doesnt exist.")
            parent = task_by_id[parent_id]
            if task.id not in parent.child_ids:
                parent.child_ids.append(task.id)

        # Ensure children include this task as a parent
        for child_id in task.child_ids:
            if child_id not in task_by_id:
                raise ValueError(f"Task with id {child_id} doesnt exist.")
            child = task_by_id[child_id]
            if task.id not in child.parent_ids:
                child.parent_ids.append(task.id)
```

run(*args)

Source Code in griptape/structures/structure.py

```
@observable
def run(self, *args) -> Structure:
    self.before_run(args)
```

(continues on next page)

(continued from previous page)

```
    result = self.try_run(*args)

    self.after_run()

    return result
```

run_stream(*args, event_types=None)

Source Code in griptape/structures/structure.py

```
@observable
def run_stream(self, *args, event_types: Optional[list[type[BaseEvent]]] = None) ->␣
↪Iterator[BaseEvent]:
    if event_types is None:
        event_types = [BaseEvent]
    elif FinishStructureRunEvent not in event_types:
        event_types = [*event_types, FinishStructureRunEvent]

    with EventListener(self._event_queue.put, event_types=event_types):
        t = Thread(target=with_contextvars(self.run), args=args)
        t.start()

        while True:
            event = self._event_queue.get()
            if isinstance(event, FinishStructureRunEvent) and event.structure_id ==␣
↪self.id:
                break
            else:
                yield event
        t.join()
```

try_find_task(task_id)

Source Code in griptape/structures/structure.py

```
def try_find_task(self, task_id: str) -> Optional[BaseTask]:
    for task in self.tasks:
        if task.id == task_id:
            return task
    return None
```

try_run(*args)abstractmethod

Source Code in griptape/structures/structure.py

```
@abstractmethod
def try_run(self, *args) -> Structure: ...
```

Workflow

Bases: *Override Default Structure Embedding Driver* “) , *FuturesExecutorMixin* ”)

Source Code in griptape/structures/workflow.py

```
@define
class Workflow(Structure, FuturesExecutorMixin):
    @property
    def input_task(self) -> Optional[BaseTask]:
        return self.order_tasks()[0] if self.tasks else None

    @property
    def output_task(self) -> Optional[BaseTask]:
        return self.order_tasks()[-1] if self.tasks else None

    @property
    def input_tasks(self) -> list[BaseTask]:
        return [task for task in self.tasks if not task.parents]

    @property
    def output_tasks(self) -> list[BaseTask]:
        return [task for task in self.tasks if not task.children]

    @property
    def outputs(self) -> list[BaseArtifact]:
        return [task.output for task in self.output_tasks if task.output is not None]

    def add_task(self, task: BaseTask) -> BaseTask:
        if (existing_task := self.try_find_task(task.id)) is not None:
            return existing_task

        task.preprocess(self)

        self._tasks.append(task)

        return task

    def insert_tasks(
        self,
        parent_tasks: BaseTask | list[BaseTask],
        tasks: BaseTask | list[BaseTask],
        child_tasks: BaseTask | list[BaseTask],
        *,
        preserve_relationship: bool = False,
    ) -> list[BaseTask]:
        """Insert tasks between parent and child tasks in the workflow.

        Args:
            parent_tasks: The tasks that will be the parents of the new tasks.
            tasks: The tasks to insert between the parent and child tasks.
            child_tasks: The tasks that will be the children of the new tasks.
            preserve_relationship: Whether to preserve the parent/child relationship␣
↪when inserting between parent and child tasks.
```

(continues on next page)

(continued from previous page)

```
        """
        if not isinstance(parent_tasks, list):
            parent_tasks = [parent_tasks]
        if not isinstance(tasks, list):
            tasks = [tasks]
        if not isinstance(child_tasks, list):
            child_tasks = [child_tasks]

        for task in tasks:
            self.insert_task(parent_tasks, task, child_tasks, preserve_
↪relationship=preserve_relationship)

        return tasks

    def insert_task(
        self,
        parent_tasks: list[BaseTask],
        task: BaseTask,
        child_tasks: list[BaseTask],
        *,
        preserve_relationship: bool = False,
    ) -> BaseTask:
        task.preprocess(self)

        self.__link_task_to_children(task, child_tasks)

        if not preserve_relationship:
            self.__remove_old_parent_child_relationships(parent_tasks, child_tasks)

        last_parent_index = self.__link_task_to_parents(task, parent_tasks)

        # Insert the new task once, just after the last parent task
        self._tasks.insert(last_parent_index + 1, task)

        return task

    @observable
    def try_run(self, *args) -> Workflow:
        exit_loop = False

        with self.create_futures_executor() as futures_executor:
            while not self.is_finished() and not exit_loop:
                futures_list = {}
                ordered_tasks = self.order_tasks()

                for task in ordered_tasks:
                    if task.can_run():
                        future = futures_executor.submit(with_contextvars(task.run))
                        futures_list[future] = task

                # Wait for all tasks to complete
                for future in futures.as_completed(futures_list):
```

(continues on next page)

(continued from previous page)

```
                    if isinstance(future.result(), ErrorArtifact) and self.fail_fast:
                        exit_loop = True

                        break

            return self

    def context(self, task: BaseTask) -> dict[str, Any]:
        context = super().context(task)

        context.update(
            {
                "task_outputs": self.task_outputs,
                "parent_outputs": task.parent_outputs,
                "parents_output_text": task.parents_output_text,
                "parents": {parent.id: parent for parent in task.parents},
                "children": {child.id: child for child in task.children},
            },
        )

        return context

    def to_graph(self) -> dict[str, set[str]]:
        graph: dict[str, set[str]] = {}

        for key_task in self.tasks:
            graph[key_task.id] = set()

            for value_task in self.tasks:
                if key_task.id in value_task.child_ids:
                    graph[key_task.id].add(value_task.id)

        return graph

    def order_tasks(self) -> list[BaseTask]:
        return [self.find_task(task_id) for task_id in TopologicalSorter(self.to_
↪graph()).static_order()]

    def __link_task_to_children(self, task: BaseTask, child_tasks: list[BaseTask]) ->␣
↪None:
        for child_task in child_tasks:
            # Link the new task to the child task
            if child_task.id not in task.child_ids:
                task.child_ids.append(child_task.id)
            if task.id not in child_task.parent_ids:
                child_task.parent_ids.append(task.id)

    def __remove_old_parent_child_relationships(
        self,
        parent_tasks: list[BaseTask],
        child_tasks: list[BaseTask],
    ) -> None:
```

(continues on next page)

(continued from previous page)

```
        for parent_task in parent_tasks:
            for child_task in child_tasks:
                # Remove the old parent/child relationship
                if child_task.id in parent_task.child_ids:
                    parent_task.child_ids.remove(child_task.id)
                if parent_task.id in child_task.parent_ids:
                    child_task.parent_ids.remove(parent_task.id)

    def __link_task_to_parents(self, task: BaseTask, parent_tasks: list[BaseTask]) ->␣
↪int:
        last_parent_index = -1
        for parent_task in parent_tasks:
            # Link the new task to the parent task
            if parent_task.id not in task.parent_ids:
                task.parent_ids.append(parent_task.id)
            if task.id not in parent_task.child_ids:
                parent_task.child_ids.append(task.id)

            try:
                parent_index = self.tasks.index(parent_task)
            except ValueError as exc:
                raise ValueError(f"Parent task {parent_task.id} not found in workflow.
↪") from exc
            else:
                last_parent_index = max(last_parent_index, parent_index)

        return last_parent_index
```

- `input_task` property
- `input_tasks` property
- `output_task` property
- `output_tasks` property
- `outputs` property

__link_task_to_children(task, child_tasks)

Source Code in griptape/structures/workflow.py

```
def __link_task_to_children(self, task: BaseTask, child_tasks: list[BaseTask]) ->␣
↪None:
    for child_task in child_tasks:
        # Link the new task to the child task
        if child_task.id not in task.child_ids:
            task.child_ids.append(child_task.id)
        if task.id not in child_task.parent_ids:
            child_task.parent_ids.append(task.id)
```

__link_task_to_parents(task, parent_tasks)

Source Code in griptape/structures/workflow.py

```
def __link_task_to_parents(self, task: BaseTask, parent_tasks: list[BaseTask]) -> int:
    last_parent_index = -1
    for parent_task in parent_tasks:
        # Link the new task to the parent task
        if parent_task.id not in task.parent_ids:
            task.parent_ids.append(parent_task.id)
        if task.id not in parent_task.child_ids:
            parent_task.child_ids.append(task.id)

        try:
            parent_index = self.tasks.index(parent_task)
        except ValueError as exc:
            raise ValueError(f"Parent task {parent_task.id} not found in workflow.")␣
↪from exc
        else:
            last_parent_index = max(last_parent_index, parent_index)

    return last_parent_index
```

__remove_old_parent_child_relationships(parent_tasks, child_tasks)

Source Code in griptape/structures/workflow.py

```
def __remove_old_parent_child_relationships(
    self,
    parent_tasks: list[BaseTask],
    child_tasks: list[BaseTask],
) -> None:
    for parent_task in parent_tasks:
        for child_task in child_tasks:
            # Remove the old parent/child relationship
            if child_task.id in parent_task.child_ids:
                parent_task.child_ids.remove(child_task.id)
            if parent_task.id in child_task.parent_ids:
                child_task.parent_ids.remove(parent_task.id)
```

add_task(task)

Source Code in griptape/structures/workflow.py

```
def add_task(self, task: BaseTask) -> BaseTask:
    if (existing_task := self.try_find_task(task.id)) is not None:
        return existing_task

    task.preprocess(self)

    self._tasks.append(task)

    return task
```

context(task)

Source Code in griptape/structures/workflow.py

```
def context(self, task: BaseTask) -> dict[str, Any]:
    context = super().context(task)

    context.update(
        {
            "task_outputs": self.task_outputs,
            "parent_outputs": task.parent_outputs,
            "parents_output_text": task.parents_output_text,
            "parents": {parent.id: parent for parent in task.parents},
            "children": {child.id: child for child in task.children},
        },
    )

    return context
```

insert_task(parent_tasks, task, child_tasks, *, preserve_relationship=False)

Source Code in griptape/structures/workflow.py

```
def insert_task(
    self,
    parent_tasks: list[BaseTask],
    task: BaseTask,
    child_tasks: list[BaseTask],
    *,
    preserve_relationship: bool = False,
) -> BaseTask:
    task.preprocess(self)

    self.__link_task_to_children(task, child_tasks)

    if not preserve_relationship:
        self.__remove_old_parent_child_relationships(parent_tasks, child_tasks)

    last_parent_index = self.__link_task_to_parents(task, parent_tasks)

    # Insert the new task once, just after the last parent task
    self._tasks.insert(last_parent_index + 1, task)

    return task
```

insert_tasks(parent_tasks, tasks, child_tasks, *, preserve_relationship=False)

Insert tasks between parent and child tasks in the workflow.

Parameters

Name	Type	Description	Default
parent_tasks	<code class="language-pythc I list[BaseTask]</code>	The tasks that will be the parents of the new tasks. 	*required*
tasks	<code class="language-pythc I list[BaseTask]</code>	The tasks to insert between the parent and child tasks. 	*required*
child_tasks	<code class="language-pythc I list[BaseTask]</code>	The tasks that will be the children of the new tasks. 	*required*
preserve_relationship	*bool*	Whether to preserve the parent/child relationship when inserting between parent and child tasks. 	*False*

Source Code in griptape/structures/workflow.py

```
def insert_tasks(
    self,
    parent_tasks: BaseTask | list[BaseTask],
    tasks: BaseTask | list[BaseTask],
    child_tasks: BaseTask | list[BaseTask],
    *,
    preserve_relationship: bool = False,
) -> list[BaseTask]:
    """Insert tasks between parent and child tasks in the workflow.

    Args:
        parent_tasks: The tasks that will be the parents of the new tasks.
        tasks: The tasks to insert between the parent and child tasks.
```

(continues on next page)

(continued from previous page)

```
        child_tasks: The tasks that will be the children of the new tasks.
        preserve_relationship: Whether to preserve the parent/child relationship when␣
↪inserting between parent and child tasks.
    """
    if not isinstance(parent_tasks, list):
        parent_tasks = [parent_tasks]
    if not isinstance(tasks, list):
        tasks = [tasks]
    if not isinstance(child_tasks, list):
        child_tasks = [child_tasks]

    for task in tasks:
        self.insert_task(parent_tasks, task, child_tasks, preserve_
↪relationship=preserve_relationship)

    return tasks
```

order_tasks()

Source Code in griptape/structures/workflow.py

```
def order_tasks(self) -> list[BaseTask]:
    return [self.find_task(task_id) for task_id in TopologicalSorter(self.to_graph()).
↪static_order()]
```

to_graph()

Source Code in griptape/structures/workflow.py

```
def to_graph(self) -> dict[str, set[str]]:
    graph: dict[str, set[str]] = {}

    for key_task in self.tasks:
        graph[key_task.id] = set()

        for value_task in self.tasks:
            if key_task.id in value_task.child_ids:
                graph[key_task.id].add(value_task.id)

    return graph
```

try_run(*args)

Source Code in griptape/structures/workflow.py

```
@observable
def try_run(self, *args) -> Workflow:
    exit_loop = False

    with self.create_futures_executor() as futures_executor:
        while not self.is_finished() and not exit_loop:
            futures_list = {}
```

(continues on next page)

(continued from previous page)

```
            ordered_tasks = self.order_tasks()

            for task in ordered_tasks:
                if task.can_run():
                    future = futures_executor.submit(with_contextvars(task.run))
                    futures_list[future] = task

            # Wait for all tasks to complete
            for future in futures.as_completed(futures_list):
                if isinstance(future.result(), ErrorArtifact) and self.fail_fast:
                    exit_loop = True

                    break

        return self
```

agent

Bases: *Override Default Structure Embedding Driver* “)

Source Code in griptape/structures/agent.py

```
@define
class Agent(Structure):
    input: Union[str, list, tuple, BaseArtifact, Callable[[BaseTask], BaseArtifact]]␣
↪= field(
        default=lambda task: task.full_context["args"][0] if task.full_context["args
↪"] else TextArtifact(value=""),
    )
    stream: Optional[bool] = field(default=None, kw_only=True)
    prompt_driver: Optional[BasePromptDriver] = field(default=None, kw_only=True)
    output_schema: Optional[Union[Schema, type[BaseModel]]] = field(default=None, kw_
↪only=True)
    tools: list[BaseTool] = field(factory=list, kw_only=True)
    max_meta_memory_entries: Optional[int] = field(default=20, kw_only=True)
    fail_fast: bool = field(default=False, kw_only=True)
    _tasks: list[Union[BaseTask, list[BaseTask]]] = field(
        factory=list, kw_only=True, alias="tasks", metadata={"serializable": True}
    )

    @fail_fast.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_fail_fast(self, _: Attribute, fail_fast: bool) -> None:  # noqa:␣
↪FBT001
        if fail_fast:
            raise ValueError("Agents cannot fail fast, as they can only have 1 task.")

    @prompt_driver.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_prompt_driver(self, _: Attribute, prompt_driver:␣
↪Optional[BasePromptDriver]) -> None:
        if prompt_driver is not None and self.stream is not None:
            warnings.warn(
```

(continues on next page)

(continued from previous page)

```
                "`Agent.prompt_driver` is set, but `Agent.stream` was provided.␣
↪`Agent.stream` will be ignored. This will be an error in the future.",
                UserWarning,
                stacklevel=2,
            )

    @_tasks.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_tasks(self, _: Attribute, tasks: list) -> None:
        if tasks and self.prompt_driver is not None:
            warnings.warn(
                "`Agent.tasks` is set, but `Agent.prompt_driver` was provided. `Agent.
↪prompt_driver` will be ignored. This will be an error in the future.",
                UserWarning,
                stacklevel=2,
            )

    def __attrs_post_init__(self) -> None:
        super().__attrs_post_init__()

        if len(self.tasks) == 0:
            self._init_task()

    @property
    def task(self) -> BaseTask:
        return self.tasks[0]

    def add_task(self, task: BaseTask) -> BaseTask:
        self._tasks.clear()

        task.preprocess(self)

        self._tasks.append(task)

        return task

    def add_tasks(self, *tasks: BaseTask | list[BaseTask]) -> list[BaseTask]:
        if len(tasks) > 1:
            raise ValueError("Agents can only have one task.")
        return super().add_tasks(*tasks)

    @observable
    def try_run(self, *args) -> Agent:
        self.task.run()

        return self

    def _init_task(self) -> None:
        if self.stream is None:
            with validators.disabled():
                self.stream = Defaults.drivers_config.prompt_driver.stream

        if self.prompt_driver is None:
```

(continues on next page)

(continued from previous page)

```
            with validators.disabled():
                prompt_driver = evolve(Defaults.drivers_config.prompt_driver,␣
↪stream=self.stream)
                self.prompt_driver = prompt_driver
        else:
            prompt_driver = self.prompt_driver

        task = PromptTask(
            self.input,
            prompt_driver=prompt_driver,
            tools=self.tools,
            output_schema=self.output_schema,
            max_meta_memory_entries=self.max_meta_memory_entries,
        )

        self.add_task(task)
```

- `_tasks = field(factory=list, kw_only=True, alias='tasks', metadata={'serializable': True})` class-attribute instance-attribute
- `fail_fast = field(default=False, kw_only=True)` class-attribute instance-attribute
- `input = field(default=lambda task: task.full_context['args'][0] if task.full_context['args'] else TextArtifact(value=''))` class-attribute instance-attribute
- `max_meta_memory_entries = field(default=20, kw_only=True)` class-attribute instance-attribute
- `output_schema = field(default=None, kw_only=True)` class-attribute instance-attribute
- `prompt_driver = field(default=None, kw_only=True)` class-attribute instance-attribute
- `stream = field(default=None, kw_only=True)` class-attribute instance-attribute
- `task` property
- `tools = field(factory=list, kw_only=True)` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/structures/agent.py

```
def __attrs_post_init__(self) -> None:
    super().__attrs_post_init__()

    if len(self.tasks) == 0:
        self._init_task()
```

_init_task()

Source Code in griptape/structures/agent.py

```
def _init_task(self) -> None:
    if self.stream is None:
        with validators.disabled():
            self.stream = Defaults.drivers_config.prompt_driver.stream
```

(continues on next page)

(continued from previous page)

```
    if self.prompt_driver is None:
        with validators.disabled():
            prompt_driver = evolve(Defaults.drivers_config.prompt_driver, stream=self.
↪stream)
            self.prompt_driver = prompt_driver
    else:
        prompt_driver = self.prompt_driver

    task = PromptTask(
        self.input,
        prompt_driver=prompt_driver,
        tools=self.tools,
        output_schema=self.output_schema,
        max_meta_memory_entries=self.max_meta_memory_entries,
    )

    self.add_task(task)
```

add_task(task)

Source Code in griptape/structures/agent.py

```
def add_task(self, task: BaseTask) -> BaseTask:
    self._tasks.clear()

    task.preprocess(self)

    self._tasks.append(task)

    return task
```

add_tasks(*tasks)

Source Code in griptape/structures/agent.py

```
def add_tasks(self, *tasks: BaseTask | list[BaseTask]) -> list[BaseTask]:
    if len(tasks) > 1:
        raise ValueError("Agents can only have one task.")
    return super().add_tasks(*tasks)
```

try_run(*args)

Source Code in griptape/structures/agent.py

```
@observable
def try_run(self, *args) -> Agent:
    self.task.run()

    return self
```

validate*fail_fast*(, fail_fast)

Source Code in griptape/structures/agent.py

```
@fail_fast.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_fail_fast(self, _: Attribute, fail_fast: bool) -> None:  # noqa: FBT001
    if fail_fast:
        raise ValueError("Agents cannot fail fast, as they can only have 1 task.")
```

validate*prompt_driver*(, prompt_driver)

Source Code in griptape/structures/agent.py

```
@prompt_driver.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_prompt_driver(self, _: Attribute, prompt_driver:␣
↪Optional[BasePromptDriver]) -> None:
    if prompt_driver is not None and self.stream is not None:
        warnings.warn(
            "`Agent.prompt_driver` is set, but `Agent.stream` was provided. `Agent.
↪stream` will be ignored. This will be an error in the future.",
            UserWarning,
            stacklevel=2,
        )
```

validate*tasks*(, tasks)

Source Code in griptape/structures/agent.py

```
@_tasks.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_tasks(self, _: Attribute, tasks: list) -> None:
    if tasks and self.prompt_driver is not None:
        warnings.warn(
            "`Agent.tasks` is set, but `Agent.prompt_driver` was provided. `Agent.
↪prompt_driver` will be ignored. This will be an error in the future.",
            UserWarning,
            stacklevel=2,
        )
```

pipeline

Bases: *Override Default Structure Embedding Driver* “)

Source Code in griptape/structures/pipeline.py

```
@define
class Pipeline(Structure):
    def add_task(self, task: BaseTask) -> BaseTask:
        if (existing_task := self.try_find_task(task.id)) is not None:
            return existing_task

        task.preprocess(self)
```

(continues on next page)

(continued from previous page)

```
        if self.output_task:
            self.output_task.child_ids.append(task.id)
            task.parent_ids.append(self.output_task.id)

        self._tasks.append(task)

        return task

    def insert_task(self, parent_task: BaseTask, task: BaseTask) -> BaseTask:
        task.preprocess(self)

        if parent_task.children:
            child_task = parent_task.children[0]

            task.child_ids.append(child_task.id)
            child_task.parent_ids.append(task.id)

            child_task.parent_ids.remove(parent_task.id)
            parent_task.child_ids.remove(child_task.id)

        task.parent_ids.append(parent_task.id)
        parent_task.child_ids.append(task.id)

        parent_index = self.tasks.index(parent_task)
        self._tasks.insert(parent_index + 1, task)

        return task

    @observable
    def try_run(self, *args) -> Pipeline:
        self.__run_from_task(self.input_task)

        return self

    def context(self, task: BaseTask) -> dict[str, Any]:
        context = super().context(task)

        context.update(
            {
                "parent_output": task.parents[0].output if task.parents else None,
                "task_outputs": self.task_outputs,
                "parent": task.parents[0] if task.parents else None,
                "child": task.children[0] if task.children else None,
            },
        )

        return context

    def __run_from_task(self, task: Optional[BaseTask]) -> None:
        if task is None or isinstance(task.run(), ErrorArtifact) and self.fail_fast:
            return
        self.__run_from_task(next(iter(task.children), None))
```

__run_from_task(task)

Source Code in griptape/structures/pipeline.py

```
def __run_from_task(self, task: Optional[BaseTask]) -> None:
    if task is None or isinstance(task.run(), ErrorArtifact) and self.fail_fast:
        return
    self.__run_from_task(next(iter(task.children), None))
```

add_task(task)

Source Code in griptape/structures/pipeline.py

```
def add_task(self, task: BaseTask) -> BaseTask:
    if (existing_task := self.try_find_task(task.id)) is not None:
        return existing_task

    task.preprocess(self)

    if self.output_task:
        self.output_task.child_ids.append(task.id)
        task.parent_ids.append(self.output_task.id)

    self._tasks.append(task)

    return task
```

context(task)

Source Code in griptape/structures/pipeline.py

```
def context(self, task: BaseTask) -> dict[str, Any]:
    context = super().context(task)

    context.update(
        {
            "parent_output": task.parents[0].output if task.parents else None,
            "task_outputs": self.task_outputs,
            "parent": task.parents[0] if task.parents else None,
            "child": task.children[0] if task.children else None,
        },
    )

    return context
```

insert_task(parent_task, task)

Source Code in griptape/structures/pipeline.py

```
def insert_task(self, parent_task: BaseTask, task: BaseTask) -> BaseTask:
    task.preprocess(self)

    if parent_task.children:
```

(continues on next page)

(continued from previous page)

```
        child_task = parent_task.children[0]

        task.child_ids.append(child_task.id)
        child_task.parent_ids.append(task.id)

        child_task.parent_ids.remove(parent_task.id)
        parent_task.child_ids.remove(child_task.id)

    task.parent_ids.append(parent_task.id)
    parent_task.child_ids.append(task.id)

    parent_index = self.tasks.index(parent_task)
    self._tasks.insert(parent_index + 1, task)

    return task
```

try_run(*args)

Source Code in griptape/structures/pipeline.py

```
@observable
def try_run(self, *args) -> Pipeline:
    self.__run_from_task(self.input_task)

    return self
```

structure

Bases: *RuleMixin* “) , *SerializableMixin* ”) , *RunnableMixin['Structure'* “) , ABC

Source Code in griptape/structures/structure.py

```
@define
class Structure(RuleMixin, SerializableMixin, RunnableMixin["Structure"], ABC):
    id: str = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata=
↪{"serializable": True})
    _tasks: list[Union[BaseTask, list[BaseTask]]] = field(
        factory=list, kw_only=True, alias="tasks", metadata={"serializable": True}
    )
    conversation_memory: Optional[BaseConversationMemory] = field(
        default=Factory(lambda: ConversationMemory()),
        kw_only=True,
        metadata={"serializable": True},
    )
    conversation_memory_strategy: Literal["per_structure", "per_task"] = field(
        default="per_structure", kw_only=True, metadata={"serializable": True}
    )
    task_memory: TaskMemory = field(
        default=Factory(lambda self: TaskMemory(), takes_self=True),
        kw_only=True,
    )
    meta_memory: MetaMemory = field(default=Factory(lambda: MetaMemory()), kw_
```

(continues on next page)

(continued from previous page)

```
↪only=True)
    fail_fast: bool = field(default=True, kw_only=True, metadata={"serializable":␣
↪True})
    _execution_args: tuple = ()
    _event_queue: Queue[BaseEvent] = field(default=Factory(lambda: Queue()),␣
↪init=False)

    def __attrs_post_init__(self) -> None:
        tasks = self._tasks.copy()
        self._tasks.clear()
        self.add_tasks(*tasks)

    def __add__(self, other: BaseTask | list[BaseTask | list[BaseTask]]) ->␣
↪list[BaseTask]:
        return self.add_tasks(*other) if isinstance(other, list) else self.add_
↪tasks(other)

    @property
    def tasks(self) -> list[BaseTask]:
        tasks = []

        for task in self._tasks:
            if isinstance(task, list):
                tasks.extend(task)
            else:
                tasks.append(task)
        return tasks

    @property
    def execution_args(self) -> tuple:
        return self._execution_args

    @property
    def input_task(self) -> Optional[BaseTask]:
        return self.tasks[0] if self.tasks else None

    @property
    def output_task(self) -> Optional[BaseTask]:
        return self.tasks[-1] if self.tasks else None

    @property
    def output(self) -> BaseArtifact:
        if self.output_task is None:
            raise ValueError("Structure has no output Task. Add a Task to the␣
↪Structure to generate output.")
        if self.output_task.output is None:
            raise ValueError("Structures output Task has no output. Run the Structure␣
↪to generate output.")
        return self.output_task.output

    @property
    def task_outputs(self) -> dict[str, Optional[BaseArtifact]]:
```

(continues on next page)

(continued from previous page)

```
        return {task.id: task.output for task in self.tasks}

    @property
    def finished_tasks(self) -> list[BaseTask]:
        return [s for s in self.tasks if s.is_finished()]

    def is_finished(self) -> bool:
        return all(not s.can_run() for s in self.tasks)

    def is_running(self) -> bool:
        return any(s for s in self.tasks if s.is_running())

    def find_task(self, task_id: str) -> BaseTask:
        if (task := self.try_find_task(task_id)) is not None:
            return task
        raise ValueError(f"Task with id {task_id} doesnt exist.")

    def try_find_task(self, task_id: str) -> Optional[BaseTask]:
        for task in self.tasks:
            if task.id == task_id:
                return task
        return None

    def add_tasks(self, *tasks: BaseTask | list[BaseTask]) -> list[BaseTask]:
        added_tasks = []
        for task in tasks:
            if isinstance(task, list):
                added_tasks.extend(self.add_tasks(*task))
            else:
                added_tasks.append(self.add_task(task))
        return added_tasks

    def context(self, task: BaseTask) -> dict[str, Any]:
        return {"args": self.execution_args, "structure": self}

    def resolve_relationships(self) -> None:
        task_by_id = {}
        for task in self.tasks:
            if task.id in task_by_id:
                raise ValueError(f"Duplicate task with id {task.id} found.")
            task_by_id[task.id] = task

        for task in self.tasks:
            # Ensure parents include this task as a child
            for parent_id in task.parent_ids:
                if parent_id not in task_by_id:
                    raise ValueError(f"Task with id {parent_id} doesnt exist.")
                parent = task_by_id[parent_id]
                if task.id not in parent.child_ids:
                    parent.child_ids.append(task.id)

            # Ensure children include this task as a parent
```

(continues on next page)

(continued from previous page)

```
            for child_id in task.child_ids:
                if child_id not in task_by_id:
                    raise ValueError(f"Task with id {child_id} doesnt exist.")
                child = task_by_id[child_id]
                if task.id not in child.parent_ids:
                    child.parent_ids.append(task.id)

    @observable
    def before_run(self, args: Any) -> None:
        super().before_run(args)
        self._execution_args = args

        [task.reset() for task in self.tasks]

        if self.input_task is not None:
            EventBus.publish_event(
                StartStructureRunEvent(
                    structure_id=self.id,
                    input_task_input=self.input_task.input,
                    input_task_output=self.input_task.output,
                ),
            )

        self.resolve_relationships()

    @observable
    def after_run(self) -> None:
        super().after_run()

        if self.output_task is not None:
            if (
                self.conversation_memory_strategy == "per_structure"
                and self.conversation_memory is not None
                and self.input_task is not None
                and self.output_task.output is not None
            ):
                run = Run(input=self.input_task.input, output=self.output_task.output)

                self.conversation_memory.add_run(run)

            EventBus.publish_event(
                FinishStructureRunEvent(
                    structure_id=self.id,
                    output_task_input=self.output_task.input,
                    output_task_output=self.output_task.output,
                ),
                flush=True,
            )

    @abstractmethod
    def add_task(self, task: BaseTask) -> BaseTask: ...
```

(continues on next page)

(continued from previous page)

```
    @observable
    def run(self, *args) -> Structure:
        self.before_run(args)

        result = self.try_run(*args)

        self.after_run()

        return result

    @observable
    def run_stream(self, *args, event_types: Optional[list[type[BaseEvent]]] = None) -
↪> Iterator[BaseEvent]:
        if event_types is None:
            event_types = [BaseEvent]
        elif FinishStructureRunEvent not in event_types:
            event_types = [*event_types, FinishStructureRunEvent]

        with EventListener(self._event_queue.put, event_types=event_types):
            t = Thread(target=with_contextvars(self.run), args=args)
            t.start()

            while True:
                event = self._event_queue.get()
                if isinstance(event, FinishStructureRunEvent) and event.structure_id␣
↪== self.id:
                    break
                else:
                    yield event
            t.join()

    @abstractmethod
    def try_run(self, *args) -> Structure: ...
```

- `_event_queue = field(default=Factory(lambda: Queue()), init=False)` class-attribute instance-attribute
- `_execution_args = ()` class-attribute instance-attribute
- `_tasks = field(factory=list, kw_only=True, alias='tasks', metadata={'serializable': True})` class-attribute instance-attribute
- `conversation_memory = field(default=Factory(lambda: ConversationMemory()), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `conversation_memory_strategy = field(default='per_structure', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `execution_args` property
- `fail_fast = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `finished_tasks` property
- `id = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

- `input_task` property
- `meta_memory = field(default=Factory(lambda: MetaMemory()), kw_only=True)` class-attribute instance-attribute
- `output` property
- `output_task` property
- `task_memory = field(default=Factory(lambda self: TaskMemory(), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `task_outputs` property
- `tasks` property

add(other)

Source Code in griptape/structures/structure.py

```
def __add__(self, other: BaseTask | list[BaseTask | list[BaseTask]]) ->␣
↪list[BaseTask]:
    return self.add_tasks(*other) if isinstance(other, list) else self.add_
↪tasks(other)
```

attrs_post_init()

Source Code in griptape/structures/structure.py

```
def __attrs_post_init__(self) -> None:
    tasks = self._tasks.copy()
    self._tasks.clear()
    self.add_tasks(*tasks)
```

add_task(task)abstractmethod

Source Code in griptape/structures/structure.py

```
@abstractmethod
def add_task(self, task: BaseTask) -> BaseTask: ...
```

add_tasks(*tasks)

Source Code in griptape/structures/structure.py

```
def add_tasks(self, *tasks: BaseTask | list[BaseTask]) -> list[BaseTask]:
    added_tasks = []
    for task in tasks:
        if isinstance(task, list):
            added_tasks.extend(self.add_tasks(*task))
        else:
            added_tasks.append(self.add_task(task))
    return added_tasks
```

after_run()

Source Code in griptape/structures/structure.py

```
@observable
def after_run(self) -> None:
    super().after_run()

    if self.output_task is not None:
        if (
            self.conversation_memory_strategy == "per_structure"
            and self.conversation_memory is not None
            and self.input_task is not None
            and self.output_task.output is not None
        ):
            run = Run(input=self.input_task.input, output=self.output_task.output)

            self.conversation_memory.add_run(run)

        EventBus.publish_event(
            FinishStructureRunEvent(
                structure_id=self.id,
                output_task_input=self.output_task.input,
                output_task_output=self.output_task.output,
            ),
            flush=True,
        )
```

before_run(args)

Source Code in griptape/structures/structure.py

```
@observable
def before_run(self, args: Any) -> None:
    super().before_run(args)
    self._execution_args = args

    [task.reset() for task in self.tasks]

    if self.input_task is not None:
        EventBus.publish_event(
            StartStructureRunEvent(
                structure_id=self.id,
                input_task_input=self.input_task.input,
                input_task_output=self.input_task.output,
            ),
        )

    self.resolve_relationships()
```

context(task)

Source Code in griptape/structures/structure.py

```
def context(self, task: BaseTask) -> dict[str, Any]:
    return {"args": self.execution_args, "structure": self}
```

find_task(task_id)

Source Code in griptape/structures/structure.py

```
def find_task(self, task_id: str) -> BaseTask:
    if (task := self.try_find_task(task_id)) is not None:
        return task
    raise ValueError(f"Task with id {task_id} doesnt exist.")
```

is_finished()

Source Code in griptape/structures/structure.py

```
def is_finished(self) -> bool:
    return all(not s.can_run() for s in self.tasks)
```

is_running()

Source Code in griptape/structures/structure.py

```
def is_running(self) -> bool:
    return any(s for s in self.tasks if s.is_running())
```

resolve_relationships()

Source Code in griptape/structures/structure.py

```
def resolve_relationships(self) -> None:
    task_by_id = {}
    for task in self.tasks:
        if task.id in task_by_id:
            raise ValueError(f"Duplicate task with id {task.id} found.")
        task_by_id[task.id] = task

    for task in self.tasks:
        # Ensure parents include this task as a child
        for parent_id in task.parent_ids:
            if parent_id not in task_by_id:
                raise ValueError(f"Task with id {parent_id} doesnt exist.")
            parent = task_by_id[parent_id]
            if task.id not in parent.child_ids:
                parent.child_ids.append(task.id)

        # Ensure children include this task as a parent
        for child_id in task.child_ids:
```

(continues on next page)

(continued from previous page)

```
            if child_id not in task_by_id:
                raise ValueError(f"Task with id {child_id} doesnt exist.")
            child = task_by_id[child_id]
            if task.id not in child.parent_ids:
                child.parent_ids.append(task.id)
```

run(*args)

Source Code in griptape/structures/structure.py

```
@observable
def run(self, *args) -> Structure:
    self.before_run(args)

    result = self.try_run(*args)

    self.after_run()

    return result
```

run_stream(*args, event_types=None)

Source Code in griptape/structures/structure.py

```
@observable
def run_stream(self, *args, event_types: Optional[list[type[BaseEvent]]] = None) ->␣
↪Iterator[BaseEvent]:
    if event_types is None:
        event_types = [BaseEvent]
    elif FinishStructureRunEvent not in event_types:
        event_types = [*event_types, FinishStructureRunEvent]

    with EventListener(self._event_queue.put, event_types=event_types):
        t = Thread(target=with_contextvars(self.run), args=args)
        t.start()

        while True:
            event = self._event_queue.get()
            if isinstance(event, FinishStructureRunEvent) and event.structure_id ==␣
↪self.id:
                break
            else:
                yield event
        t.join()
```

try_find_task(task_id)

Source Code in griptape/structures/structure.py

```
def try_find_task(self, task_id: str) -> Optional[BaseTask]:
    for task in self.tasks:
```

(continues on next page)

(continued from previous page)

```
        if task.id == task_id:
            return task
    return None
```

try_run(*args)abstractmethod

Source Code in griptape/structures/structure.py

```
@abstractmethod
def try_run(self, *args) -> Structure: ...
```

workflow

Bases: *Override Default Structure Embedding Driver* “) , *FuturesExecutorMixin* ”)

Source Code in griptape/structures/workflow.py

```
@define
class Workflow(Structure, FuturesExecutorMixin):
    @property
    def input_task(self) -> Optional[BaseTask]:
        return self.order_tasks()[0] if self.tasks else None

    @property
    def output_task(self) -> Optional[BaseTask]:
        return self.order_tasks()[-1] if self.tasks else None

    @property
    def input_tasks(self) -> list[BaseTask]:
        return [task for task in self.tasks if not task.parents]

    @property
    def output_tasks(self) -> list[BaseTask]:
        return [task for task in self.tasks if not task.children]

    @property
    def outputs(self) -> list[BaseArtifact]:
        return [task.output for task in self.output_tasks if task.output is not None]

    def add_task(self, task: BaseTask) -> BaseTask:
        if (existing_task := self.try_find_task(task.id)) is not None:
            return existing_task

        task.preprocess(self)

        self._tasks.append(task)

        return task

    def insert_tasks(
        self,
        parent_tasks: BaseTask | list[BaseTask],
```

(continues on next page)

(continued from previous page)

```
        tasks: BaseTask | list[BaseTask],
        child_tasks: BaseTask | list[BaseTask],
        *,
        preserve_relationship: bool = False,
    ) -> list[BaseTask]:
        """Insert tasks between parent and child tasks in the workflow.

        Args:
            parent_tasks: The tasks that will be the parents of the new tasks.
            tasks: The tasks to insert between the parent and child tasks.
            child_tasks: The tasks that will be the children of the new tasks.
            preserve_relationship: Whether to preserve the parent/child relationship␣
↪when inserting between parent and child tasks.
        """
        if not isinstance(parent_tasks, list):
            parent_tasks = [parent_tasks]
        if not isinstance(tasks, list):
            tasks = [tasks]
        if not isinstance(child_tasks, list):
            child_tasks = [child_tasks]

        for task in tasks:
            self.insert_task(parent_tasks, task, child_tasks, preserve_
↪relationship=preserve_relationship)

        return tasks

    def insert_task(
        self,
        parent_tasks: list[BaseTask],
        task: BaseTask,
        child_tasks: list[BaseTask],
        *,
        preserve_relationship: bool = False,
    ) -> BaseTask:
        task.preprocess(self)

        self.__link_task_to_children(task, child_tasks)

        if not preserve_relationship:
            self.__remove_old_parent_child_relationships(parent_tasks, child_tasks)

        last_parent_index = self.__link_task_to_parents(task, parent_tasks)

        # Insert the new task once, just after the last parent task
        self._tasks.insert(last_parent_index + 1, task)

        return task

    @observable
    def try_run(self, *args) -> Workflow:
        exit_loop = False
```

(continues on next page)

(continued from previous page)

```
        with self.create_futures_executor() as futures_executor:
            while not self.is_finished() and not exit_loop:
                futures_list = {}
                ordered_tasks = self.order_tasks()

                for task in ordered_tasks:
                    if task.can_run():
                        future = futures_executor.submit(with_contextvars(task.run))
                        futures_list[future] = task

                # Wait for all tasks to complete
                for future in futures.as_completed(futures_list):
                    if isinstance(future.result(), ErrorArtifact) and self.fail_fast:
                        exit_loop = True

                        break

            return self

    def context(self, task: BaseTask) -> dict[str, Any]:
        context = super().context(task)

        context.update(
            {
                "task_outputs": self.task_outputs,
                "parent_outputs": task.parent_outputs,
                "parents_output_text": task.parents_output_text,
                "parents": {parent.id: parent for parent in task.parents},
                "children": {child.id: child for child in task.children},
            },
        )

        return context

    def to_graph(self) -> dict[str, set[str]]:
        graph: dict[str, set[str]] = {}

        for key_task in self.tasks:
            graph[key_task.id] = set()

            for value_task in self.tasks:
                if key_task.id in value_task.child_ids:
                    graph[key_task.id].add(value_task.id)

        return graph

    def order_tasks(self) -> list[BaseTask]:
        return [self.find_task(task_id) for task_id in TopologicalSorter(self.to_
↪graph()).static_order()]

    def __link_task_to_children(self, task: BaseTask, child_tasks: list[BaseTask]) ->␣
```

(continues on next page)

(continued from previous page)

```
↪None:
        for child_task in child_tasks:
            # Link the new task to the child task
            if child_task.id not in task.child_ids:
                task.child_ids.append(child_task.id)
            if task.id not in child_task.parent_ids:
                child_task.parent_ids.append(task.id)

    def __remove_old_parent_child_relationships(
        self,
        parent_tasks: list[BaseTask],
        child_tasks: list[BaseTask],
    ) -> None:
        for parent_task in parent_tasks:
            for child_task in child_tasks:
                # Remove the old parent/child relationship
                if child_task.id in parent_task.child_ids:
                    parent_task.child_ids.remove(child_task.id)
                if parent_task.id in child_task.parent_ids:
                    child_task.parent_ids.remove(parent_task.id)

    def __link_task_to_parents(self, task: BaseTask, parent_tasks: list[BaseTask]) ->␣
↪int:
        last_parent_index = -1
        for parent_task in parent_tasks:
            # Link the new task to the parent task
            if parent_task.id not in task.parent_ids:
                task.parent_ids.append(parent_task.id)
            if task.id not in parent_task.child_ids:
                parent_task.child_ids.append(task.id)

            try:
                parent_index = self.tasks.index(parent_task)
            except ValueError as exc:
                raise ValueError(f"Parent task {parent_task.id} not found in workflow.
↪") from exc
            else:
                last_parent_index = max(last_parent_index, parent_index)

        return last_parent_index
```

- `input_task` property
- `input_tasks` property
- `output_task` property
- `output_tasks` property
- `outputs` property

__link_task_to_children(task, child_tasks)

Source Code in griptape/structures/workflow.py

```
def __link_task_to_children(self, task: BaseTask, child_tasks: list[BaseTask]) ->␣
↪None:
    for child_task in child_tasks:
        # Link the new task to the child task
        if child_task.id not in task.child_ids:
            task.child_ids.append(child_task.id)
        if task.id not in child_task.parent_ids:
            child_task.parent_ids.append(task.id)
```

__link_task_to_parents(task, parent_tasks)

Source Code in griptape/structures/workflow.py

```
def __link_task_to_parents(self, task: BaseTask, parent_tasks: list[BaseTask]) -> int:
    last_parent_index = -1
    for parent_task in parent_tasks:
        # Link the new task to the parent task
        if parent_task.id not in task.parent_ids:
            task.parent_ids.append(parent_task.id)
        if task.id not in parent_task.child_ids:
            parent_task.child_ids.append(task.id)

        try:
            parent_index = self.tasks.index(parent_task)
        except ValueError as exc:
            raise ValueError(f"Parent task {parent_task.id} not found in workflow.")␣
↪from exc
        else:
            last_parent_index = max(last_parent_index, parent_index)

    return last_parent_index
```

__remove_old_parent_child_relationships(parent_tasks, child_tasks)

Source Code in griptape/structures/workflow.py

```
def __remove_old_parent_child_relationships(
    self,
    parent_tasks: list[BaseTask],
    child_tasks: list[BaseTask],
) -> None:
    for parent_task in parent_tasks:
        for child_task in child_tasks:
            # Remove the old parent/child relationship
            if child_task.id in parent_task.child_ids:
                parent_task.child_ids.remove(child_task.id)
            if parent_task.id in child_task.parent_ids:
                child_task.parent_ids.remove(parent_task.id)
```

add_task(task)

Source Code in griptape/structures/workflow.py

```
def add_task(self, task: BaseTask) -> BaseTask:
    if (existing_task := self.try_find_task(task.id)) is not None:
        return existing_task

    task.preprocess(self)

    self._tasks.append(task)

    return task
```

context(task)

Source Code in griptape/structures/workflow.py

```
def context(self, task: BaseTask) -> dict[str, Any]:
    context = super().context(task)

    context.update(
        {
            "task_outputs": self.task_outputs,
            "parent_outputs": task.parent_outputs,
            "parents_output_text": task.parents_output_text,
            "parents": {parent.id: parent for parent in task.parents},
            "children": {child.id: child for child in task.children},
        },
    )

    return context
```

insert_task(parent_tasks, task, child_tasks, *, preserve_relationship=False)

Source Code in griptape/structures/workflow.py

```
def insert_task(
    self,
    parent_tasks: list[BaseTask],
    task: BaseTask,
    child_tasks: list[BaseTask],
    *,
    preserve_relationship: bool = False,
) -> BaseTask:
    task.preprocess(self)

    self.__link_task_to_children(task, child_tasks)

    if not preserve_relationship:
        self.__remove_old_parent_child_relationships(parent_tasks, child_tasks)

    last_parent_index = self.__link_task_to_parents(task, parent_tasks)
```

(continues on next page)

(continued from previous page)

```
    # Insert the new task once, just after the last parent task
    self._tasks.insert(last_parent_index + 1, task)

    return task
```

insert_tasks(parent_tasks, tasks, child_tasks, *, preserve_relationship=False)

Insert tasks between parent and child tasks in the workflow.

Parameters

Name	Type	Description	Default
parent_tasks	<code class="language-pythc I list[BaseTask]</code>	The tasks that will be the parents of the new tasks. 	*required*
tasks	<code class="language-pythc I list[BaseTask]</code>	The tasks to insert between the parent and child tasks. 	*required*
child_tasks	<code class="language-pythc I list[BaseTask]</code>	The tasks that will be the children of the new tasks. 	*required*
preserve_relationship	*bool*	Whether to preserve the parent/child relationship when inserting between parent and child tasks. 	*False*

Source Code in griptape/structures/workflow.py

```
def insert_tasks(
    self,
    parent_tasks: BaseTask | list[BaseTask],
    tasks: BaseTask | list[BaseTask],
    child_tasks: BaseTask | list[BaseTask],
    *,
    preserve_relationship: bool = False,
) -> list[BaseTask]:
    """Insert tasks between parent and child tasks in the workflow.

    Args:
        parent_tasks: The tasks that will be the parents of the new tasks.
        tasks: The tasks to insert between the parent and child tasks.
        child_tasks: The tasks that will be the children of the new tasks.
        preserve_relationship: Whether to preserve the parent/child relationship when␣
↪inserting between parent and child tasks.
    """
    if not isinstance(parent_tasks, list):
        parent_tasks = [parent_tasks]
    if not isinstance(tasks, list):
        tasks = [tasks]
    if not isinstance(child_tasks, list):
        child_tasks = [child_tasks]

    for task in tasks:
        self.insert_task(parent_tasks, task, child_tasks, preserve_
↪relationship=preserve_relationship)

    return tasks
```

order_tasks()

Source Code in griptape/structures/workflow.py

```
def order_tasks(self) -> list[BaseTask]:
    return [self.find_task(task_id) for task_id in TopologicalSorter(self.to_graph()).
↪static_order()]
```

to_graph()

Source Code in griptape/structures/workflow.py

```
def to_graph(self) -> dict[str, set[str]]:
    graph: dict[str, set[str]] = {}

    for key_task in self.tasks:
        graph[key_task.id] = set()

        for value_task in self.tasks:
            if key_task.id in value_task.child_ids:
                graph[key_task.id].add(value_task.id)
```

(continues on next page)

(continued from previous page)

```
    return graph
```

try_run(*args)

Source Code in griptape/structures/workflow.py

```
@observable
def try_run(self, *args) -> Workflow:
    exit_loop = False

    with self.create_futures_executor() as futures_executor:
        while not self.is_finished() and not exit_loop:
            futures_list = {}
            ordered_tasks = self.order_tasks()

            for task in ordered_tasks:
                if task.can_run():
                    future = futures_executor.submit(with_contextvars(task.run))
                    futures_list[future] = task

            # Wait for all tasks to complete
            for future in futures.as_completed(futures_list):
                if isinstance(future.result(), ErrorArtifact) and self.fail_fast:
                    exit_loop = True

                    break

        return self
```

11.5.18 tasks

- `__all__ = ['ActionsSubtask', 'OutputSchemaValidationSubtask', 'AssistantTask', 'AudioTranscriptionTask', 'BaseAudioGenerationTask', 'BaseImageGenerationTask', 'BaseTask', 'BaseSubtask', 'BaseTextInputTask', 'BranchTask', 'CodeExecutionTask', 'ExtractionTask', 'InpaintingImageGenerationTask', 'OutpaintingImageGenerationTask', 'PromptImageGenerationTask', 'PromptTask', 'RagTask', 'StructureRunTask', 'TextSummaryTask', 'TextToSpeechTask', 'ToolTask', 'ToolkitTask', 'VariationImageGenerationTask']` module-attribute

Bases: *BaseSubtask[Union[ListArtifact, ErrorArtifact ")*

Source Code in griptape/tasks/actions_subtask.py

```
@define
class ActionsSubtask(BaseSubtask[Union[ListArtifact, ErrorArtifact]]):
    THOUGHT_PATTERN = r"(?s)^Thought:\s*(.*?)$"
    ACTIONS_PATTERN = r"(?s)Actions:[^\[]*(\[.*\])"
    ANSWER_PATTERN = r"(?s)^Answer:\s?([\s\S]*)$"

    RESPONSE_STOP_SEQUENCE = "<|Response|>"

    thought: Optional[str] = field(default=None, kw_only=True)
```

(continues on next page)

(continued from previous page)

```
    actions: list[ToolAction] = field(factory=list, kw_only=True)
    output: Optional[BaseArtifact] = field(default=None, init=False)
    generate_assistant_subtask_template: Callable[[ActionsSubtask], str] = field(
        default=Factory(lambda self: self.default_generate_assistant_subtask_template,
↪ takes_self=True),
        kw_only=True,
    )
    generate_user_subtask_template: Callable[[ActionsSubtask], str] = field(
        default=Factory(lambda self: self.default_generate_user_subtask_template,␣
↪takes_self=True),
        kw_only=True,
    )
    response_stop_sequence: str = field(default=RESPONSE_STOP_SEQUENCE, kw_only=True)
    _input: Union[str, list, tuple, BaseArtifact, Callable[[BaseTask], BaseArtifact]]␣
↪= field(
        default=lambda task: task.full_context["args"][0] if task.full_context["args
↪"] else TextArtifact(value=""),
        alias="input",
    )
    _memory: Optional[TaskMemory] = None
    _origin_task: Optional[BaseTask] = field(default=None, kw_only=True)

    @property
    def input(self) -> TextArtifact | AudioArtifact | ListArtifact:
        return self._process_task_input(self._input)

    @input.setter
    def input(self, value: str | list | tuple | BaseArtifact | Callable[[BaseTask],␣
↪BaseArtifact]) -> None:
        self._input = value

    def attach_to(self, parent_task: BaseTask) -> None:
        super().attach_to(parent_task)
        self.structure = parent_task.structure

        task_input = self.input
        try:
            if isinstance(task_input, TextArtifact) and task_input.meta.get("is_react_
↪prompt", False):
                self.__init_from_prompt(task_input.to_text())
            else:
                self.__init_from_artifact(task_input)

            # If StructuredOutputTool was used, treat the input to it as the output␣
↪of the subtask.
            structured_outputs = [a for a in self.actions if isinstance(a.tool,␣
↪StructuredOutputTool)]
            if structured_outputs:
                output_values = [JsonArtifact(a.input["values"]) for a in structured_
↪outputs]
                if len(structured_outputs) > 1:
                    self.output = ListArtifact(output_values)
```

(continues on next page)

(continued from previous page)

```
                else:
                    self.output = output_values[0]
        except Exception as e:
            logger.error("Subtask %s\nError parsing tool action: %s", self.origin_
↪task.id, e)

            self.output = ErrorArtifact(f"ToolAction input parsing error: {e}",␣
↪exception=e)

    def before_run(self) -> None:
        EventBus.publish_event(
            StartActionsSubtaskEvent(
                task_id=self.id,
                task_parent_ids=self.parent_ids,
                task_child_ids=self.child_ids,
                task_input=self.input,
                task_output=self.output,
                subtask_parent_task_id=self.origin_task.id,
                subtask_thought=self.thought,
                subtask_actions=self.actions_to_dicts(),
            ),
        )

        parts = [
            f"{self.__class__.__name__} {self.id}",
            *([f"\nThought: {self.thought}"] if self.thought else []),
            f"\nActions: {self.actions_to_json()}",
        ]
        logger.info("".join(parts))

    def try_run(self) -> ListArtifact | ErrorArtifact:
        try:
            if any(isinstance(a.output, ErrorArtifact) for a in self.actions):
                errors = [a.output.value for a in self.actions if isinstance(a.output,
↪ ErrorArtifact)]

                self.output = ErrorArtifact("\n\n".join(errors))
            else:
                results = self.run_actions(self.actions)

                actions_output = []
                for result in results:
                    tag, output = result
                    output.name = f"{tag} output"

                    actions_output.append(output)
                self.output = ListArtifact(actions_output)
        except Exception as e:
            logger.debug("Subtask %s\n%s", self.id, e)

            self.output = ErrorArtifact(str(e), exception=e)
        if self.output is not None:
```

(continues on next page)

(continued from previous page)

```
            return self.output
        return ErrorArtifact("no tool output")

    def run_actions(self, actions: list[ToolAction]) -> list[tuple[str,␣
↪BaseArtifact]]:
        with self.create_futures_executor() as futures_executor:
            return utils.execute_futures_list(
                [futures_executor.submit(with_contextvars(self.run_action), a) for a␣
↪in actions]
            )

    def run_action(self, action: ToolAction) -> tuple[str, BaseArtifact]:
        if action.tool is not None:
            if action.path is not None:
                output = action.tool.run(getattr(action.tool, action.path), self,␣
↪action)
            else:
                output = ErrorArtifact("action path not found")
        else:
            output = ErrorArtifact("action name not found")
        action.output = output

        return action.tag, output

    def after_run(self) -> None:
        response = self.output.to_text() if isinstance(self.output, BaseArtifact)␣
↪else str(self.output)

        EventBus.publish_event(
            FinishActionsSubtaskEvent(
                task_id=self.id,
                task_parent_ids=self.parent_ids,
                task_child_ids=self.child_ids,
                task_input=self.input,
                task_output=self.output,
                subtask_parent_task_id=self.origin_task.id,
                subtask_thought=self.thought,
                subtask_actions=self.actions_to_dicts(),
            ),
        )
        logger.info("%s %s\nResponse: %s", self.__class__.__name__, self.id, response)

    def actions_to_dicts(self) -> list[dict]:
        json_list = []

        for action in self.actions:
            json_dict = {}

            if action.tag:
                json_dict["tag"] = action.tag

            if action.name:
```

(continues on next page)

(continued from previous page)

```
                json_dict["name"] = action.name

            if action.path:
                json_dict["path"] = action.path

            if action.input:
                json_dict["input"] = action.input

            json_list.append(json_dict)

        return json_list

    def actions_to_json(self) -> str:
        return json.dumps(self.actions_to_dicts(), indent=2)

    def add_to_prompt_stack(self, stack: PromptStack) -> None:
        from griptape.tasks import PromptTask

        if isinstance(self.origin_task, PromptTask) and self.origin_task.prompt_
↪driver.use_native_tools:
            action_calls = [
                ToolAction(name=action.name, path=action.path, tag=action.tag,␣
↪input=action.input)
                for action in self.actions
            ]
            action_results = [
                ToolAction(
                    name=action.name,
                    path=action.path,
                    tag=action.tag,
                    output=action.output if action.output is not None else self.
↪output,
                )
                for action in self.actions
            ]

            stack.add_assistant_message(
                ListArtifact(
                    [
                        *([TextArtifact(self.thought)] if self.thought else []),
                        *[ActionArtifact(a) for a in action_calls],
                    ],
                ),
            )
            stack.add_user_message(
                ListArtifact(
                    [
                        *[ActionArtifact(a) for a in action_results],
                        *([] if self.output else [TextArtifact("Please keep going")]),
                    ],
                ),
            )
```

(continues on next page)

(continued from previous page)

```
        else:
            stack.add_assistant_message(self.generate_assistant_subtask_
↪template(self))
            stack.add_user_message(self.generate_user_subtask_template(self))

    def default_generate_assistant_subtask_template(self, subtask: ActionsSubtask) ->␣
↪str:
        return J2("tasks/prompt_task/assistant_actions_subtask.j2").render(
            stop_sequence=self.response_stop_sequence,
            subtask=subtask,
        )

    def default_generate_user_subtask_template(self, subtask: ActionsSubtask) -> str:
        return J2("tasks/prompt_task/user_actions_subtask.j2").render(
            stop_sequence=self.response_stop_sequence,
            subtask=subtask,
        )

    def _process_task_input(
        self,
        task_input: Union[str, tuple, list, BaseArtifact, Callable[[BaseTask],␣
↪BaseArtifact]],
    ) -> Union[TextArtifact, AudioArtifact, ListArtifact]:
        if isinstance(task_input, (TextArtifact, AudioArtifact, ListArtifact)):
            return task_input
        if isinstance(task_input, ActionArtifact):
            return ListArtifact([task_input])
        if isinstance(task_input, Callable):
            return self._process_task_input(task_input(self))
        if isinstance(task_input, str):
            return self._process_task_input(TextArtifact(task_input))
        if isinstance(task_input, (list, tuple)):
            return ListArtifact([self._process_task_input(elem) for elem in task_
↪input])
        raise ValueError(f"Invalid input type: {type(task_input)} ")

    def __init_from_prompt(self, value: str) -> None:
        thought_matches = re.findall(self.THOUGHT_PATTERN, value, re.MULTILINE)
        actions_matches = re.findall(self.ACTIONS_PATTERN, value, re.DOTALL)
        answer_matches = re.findall(self.ANSWER_PATTERN, value, re.MULTILINE)

        self.actions = self.__parse_actions(actions_matches)

        if thought_matches:
            self.thought = thought_matches[-1]

        if not self.actions and self.output is None:
            if answer_matches:
                # A direct answer is provided, set it as the output.
                self.output = TextArtifact(answer_matches[-1])
            else:
                # The LLM failed to follow the ReAct prompt, set the LLMs raw␣
```

(continues on next page)

(continued from previous page)

```
↪response as the output.
                self.output = TextArtifact(value)

    def __init_from_artifact(self, artifact: TextArtifact | AudioArtifact |␣
↪ListArtifact) -> None:
        """Parses the input Artifact to extract either a final answer or thought and␣
↪actions.

        When the input Artifact is a TextArtifact, it is assumed to be the final␣
↪answer.
        When the input Artifact is a ListArtifact, it is assumed to contain both␣
↪thought and actions.
        Text Artifacts are parsed as the thought, and ToolAction Artifacts parsed as␣
↪the actions.

        Args:
            artifact: The input Artifacts.

        Returns:
            None
        """
        # When using native tools, we can assume that a TextArtifact or AudioArtifact␣
↪is the LLM providing its final answer.
        if isinstance(artifact, (TextArtifact, AudioArtifact)):
            self.output = artifact
            return

        self.actions = [
            self.__process_action_object(artifact.value.to_dict())
            for artifact in artifact.value
            if isinstance(artifact, ActionArtifact)
        ]

        # When parsing from Artifacts we cant determine the thought unless there are␣
↪also Actions
        if self.actions:
            thoughts = [artifact.value for artifact in artifact.value if␣
↪isinstance(artifact, TextArtifact)]
            if thoughts:
                self.thought = thoughts[0]
        elif self.output is None:
            self.output = TextArtifact(artifact.to_text())

    def __parse_actions(self, actions_matches: list[str]) -> list[ToolAction]:
        if len(actions_matches) == 0:
            return []
        try:
            data = actions_matches[-1]
            actions_list: list[dict] = json.loads(data, strict=False)

            return [self.__process_action_object(action_object) for action_object in␣
↪actions_list]
```

(continues on next page)

(continued from previous page)

```
        except json.JSONDecodeError as e:
            logger.debug("Subtask %s\nInvalid actions JSON: %s", self.origin_task.id,␣
↪e)

            self.output = ErrorArtifact(f"Actions JSON decoding error: {e}",␣
↪exception=e)

            return []

    def __process_action_object(self, action_object: dict) -> ToolAction:
        # Load action tag; throw exception if the key is not present
        action_tag = action_object["tag"]

        # Load action name; throw exception if the key is not present
        action_name = action_object["name"]

        # Load action method; throw exception if the key is not present
        action_path = action_object["path"]

        # Load optional input value; dont throw exceptions if key is not present
        if "input" in action_object:
            # Some LLMs dont support nested parameters and therefore wont generate
↪"values".
            # So we need to manually add it here.
            if "values" not in action_object["input"]:
                action_object["input"] = {"values": action_object["input"]}

            # The schema library has a bug, where something like `Or(str, None)`␣
↪doesnt get
            # correctly translated into JSON schema. For some optional input fields␣
↪LLMs sometimes
            # still provide null value, which trips up the validator. The temporary␣
↪solution that
            # works is to strip all key-values where value is null.
            action_input = remove_null_values_in_dict_recursively(action_object["input
↪"])
        else:
            action_input = {}

        # Load the action itself
        if isinstance(self.origin_task, ActionsSubtaskOriginMixin):
            tool = self.origin_task.find_tool(action_name)
        else:
            raise Exception("ActionSubtask must be attached to a Task that implements␣
↪ActionSubtaskOriginMixin.")

        action = ToolAction(tag=action_tag, name=action_name, path=action_path,␣
↪input=action_input, tool=tool)

        self.__validate_action(action)

        return action
```

(continues on next page)

(continued from previous page)

```
    def __validate_action(self, action: ToolAction) -> None:
        if action.tool is None:
            return

        if action.path is None:
            raise Exception("ToolAction path not found.")
        activity = getattr(action.tool, action.path)

        if activity is None:
            raise Exception("Activity not found.")

        activity_schema = action.tool.activity_schema(activity)

        if activity_schema is None or action.input is None:
            return

        try:
            action.tool.validate_activity_schema(activity_schema, action.input)
        except ValueError as e:
            logger.debug("Subtask %s\nInvalid action JSON: %s", self.origin_task.id,␣
↪e)

            action.output = ErrorArtifact(f"Activity input JSON validation error: {e}
↪", exception=e)
```

- `ACTIONS_PATTERN = '(?s)Actions:[^\\[]*(\\[.*\\])'` class-attribute instance-attribute
- `ANSWER_PATTERN = '(?s)^Answer:\\s?([\\s\\S]*)$'` class-attribute instance-attribute
- `RESPONSE_STOP_SEQUENCE = '<|Response|>'` class-attribute instance-attribute
- `THOUGHT_PATTERN = '(?s)^Thought:\\s*(.*?)$'` class-attribute instance-attribute
- `_input = field(default=lambda task: task.full_context['args'][0] if task.full_context['args'] else TextArtifact(value=''), alias='input')` class-attribute instance-attribute
- `_memory = None` class-attribute instance-attribute
- `_origin_task = field(default=None, kw_only=True)` class-attribute instance-attribute
- `actions = field(factory=list, kw_only=True)` class-attribute instance-attribute
- `generate_assistant_subtask_template = field(default=Factory(lambda self: self.default_generate_assistant_subtask_template, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `generate_user_subtask_template = field(default=Factory(lambda self: self.default_generate_user_subtask_template, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `input` property writable
- `output = field(default=None, init=False)` class-attribute instance-attribute
- `response_stop_sequence = field(default=RESPONSE_STOP_SEQUENCE, kw_only=True)` class-attribute instance-attribute

- `thought = field(default=None, kw_only=True)` class-attribute instance-attribute

__init_from_artifact(artifact)

Parses the input Artifact to extract either a final answer or thought and actions.

When the input Artifact is a TextArtifact, it is assumed to be the final answer. When the input Artifact is a ListArtifact, it is assumed to contain both thought and actions. Text Artifacts are parsed as the thought, and ToolAction Artifacts parsed as the actions.

Parameters

Name	Type	Description	Default
artifact	<code class="language-pythc I AudioArtifact I ListArtifact</code>	The input Artifacts. 	*required*

Returns

Type	Description
None	None

Source Code in griptape/tasks/actions_subtask.py

```
def __init_from_artifact(self, artifact: TextArtifact | AudioArtifact | ListArtifact)␣
↪-> None:
    """Parses the input Artifact to extract either a final answer or thought and␣
↪actions.

    When the input Artifact is a TextArtifact, it is assumed to be the final answer.
    When the input Artifact is a ListArtifact, it is assumed to contain both thought␣
↪and actions.
    Text Artifacts are parsed as the thought, and ToolAction Artifacts parsed as the␣
↪actions.

    Args:
        artifact: The input Artifacts.

    Returns:
        None
    """
    # When using native tools, we can assume that a TextArtifact or AudioArtifact is␣
↪the LLM providing its final answer.
```

(continues on next page)

(continued from previous page)

```
    if isinstance(artifact, (TextArtifact, AudioArtifact)):
        self.output = artifact
        return

    self.actions = [
        self.__process_action_object(artifact.value.to_dict())
        for artifact in artifact.value
        if isinstance(artifact, ActionArtifact)
    ]

    # When parsing from Artifacts we cant determine the thought unless there are also␣
↪Actions
    if self.actions:
        thoughts = [artifact.value for artifact in artifact.value if␣
↪isinstance(artifact, TextArtifact)]
        if thoughts:
            self.thought = thoughts[0]
    elif self.output is None:
        self.output = TextArtifact(artifact.to_text())
```

__init_from_prompt(value)

Source Code in griptape/tasks/actions_subtask.py

```
def __init_from_prompt(self, value: str) -> None:
    thought_matches = re.findall(self.THOUGHT_PATTERN, value, re.MULTILINE)
    actions_matches = re.findall(self.ACTIONS_PATTERN, value, re.DOTALL)
    answer_matches = re.findall(self.ANSWER_PATTERN, value, re.MULTILINE)

    self.actions = self.__parse_actions(actions_matches)

    if thought_matches:
        self.thought = thought_matches[-1]

    if not self.actions and self.output is None:
        if answer_matches:
            # A direct answer is provided, set it as the output.
            self.output = TextArtifact(answer_matches[-1])
        else:
            # The LLM failed to follow the ReAct prompt, set the LLMs raw response as␣
↪the output.
            self.output = TextArtifact(value)
```

__parse_actions(actions_matches)

Source Code in griptape/tasks/actions_subtask.py

```
def __parse_actions(self, actions_matches: list[str]) -> list[ToolAction]:
    if len(actions_matches) == 0:
        return []
    try:
        data = actions_matches[-1]
```

(continues on next page)

(continued from previous page)

```
        actions_list: list[dict] = json.loads(data, strict=False)

        return [self.__process_action_object(action_object) for action_object in␣
↪actions_list]
    except json.JSONDecodeError as e:
        logger.debug("Subtask %s\nInvalid actions JSON: %s", self.origin_task.id, e)

        self.output = ErrorArtifact(f"Actions JSON decoding error: {e}", exception=e)

        return []
```

__process_action_object(action_object)

Source Code in griptape/tasks/actions_subtask.py

```
def __process_action_object(self, action_object: dict) -> ToolAction:
    # Load action tag; throw exception if the key is not present
    action_tag = action_object["tag"]

    # Load action name; throw exception if the key is not present
    action_name = action_object["name"]

    # Load action method; throw exception if the key is not present
    action_path = action_object["path"]

    # Load optional input value; dont throw exceptions if key is not present
    if "input" in action_object:
        # Some LLMs dont support nested parameters and therefore wont generate "values
↪".
        # So we need to manually add it here.
        if "values" not in action_object["input"]:
            action_object["input"] = {"values": action_object["input"]}

        # The schema library has a bug, where something like `Or(str, None)` doesnt␣
↪get
        # correctly translated into JSON schema. For some optional input fields LLMs␣
↪sometimes
        # still provide null value, which trips up the validator. The temporary␣
↪solution that
        # works is to strip all key-values where value is null.
        action_input = remove_null_values_in_dict_recursively(action_object["input"])
    else:
        action_input = {}

    # Load the action itself
    if isinstance(self.origin_task, ActionsSubtaskOriginMixin):
        tool = self.origin_task.find_tool(action_name)
    else:
        raise Exception("ActionSubtask must be attached to a Task that implements␣
↪ActionSubtaskOriginMixin.")

    action = ToolAction(tag=action_tag, name=action_name, path=action_path,␣
```

(continues on next page)

(continued from previous page)

```
↪input=action_input, tool=tool)

    self.__validate_action(action)

    return action
```

__validate_action(action)

Source Code in griptape/tasks/actions_subtask.py

```
def __validate_action(self, action: ToolAction) -> None:
    if action.tool is None:
        return

    if action.path is None:
        raise Exception("ToolAction path not found.")
    activity = getattr(action.tool, action.path)

    if activity is None:
        raise Exception("Activity not found.")

    activity_schema = action.tool.activity_schema(activity)

    if activity_schema is None or action.input is None:
        return

    try:
        action.tool.validate_activity_schema(activity_schema, action.input)
    except ValueError as e:
        logger.debug("Subtask %s\nInvalid action JSON: %s", self.origin_task.id, e)

        action.output = ErrorArtifact(f"Activity input JSON validation error: {e}",␣
↪exception=e)
```

_process_task_input(task_input)

Source Code in griptape/tasks/actions_subtask.py

```
def _process_task_input(
    self,
    task_input: Union[str, tuple, list, BaseArtifact, Callable[[BaseTask],␣
↪BaseArtifact]],
) -> Union[TextArtifact, AudioArtifact, ListArtifact]:
    if isinstance(task_input, (TextArtifact, AudioArtifact, ListArtifact)):
        return task_input
    if isinstance(task_input, ActionArtifact):
        return ListArtifact([task_input])
    if isinstance(task_input, Callable):
        return self._process_task_input(task_input(self))
    if isinstance(task_input, str):
        return self._process_task_input(TextArtifact(task_input))
    if isinstance(task_input, (list, tuple)):
```

(continues on next page)

(continued from previous page)

```
        return ListArtifact([self._process_task_input(elem) for elem in task_input])
    raise ValueError(f"Invalid input type: {type(task_input)} ")
```

actions_to_dicts()

Source Code in griptape/tasks/actions_subtask.py

```
def actions_to_dicts(self) -> list[dict]:
    json_list = []

    for action in self.actions:
        json_dict = {}

        if action.tag:
            json_dict["tag"] = action.tag

        if action.name:
            json_dict["name"] = action.name

        if action.path:
            json_dict["path"] = action.path

        if action.input:
            json_dict["input"] = action.input

        json_list.append(json_dict)

    return json_list
```

actions_to_json()

Source Code in griptape/tasks/actions_subtask.py

```
def actions_to_json(self) -> str:
    return json.dumps(self.actions_to_dicts(), indent=2)
```

add_to_prompt_stack(stack)

Source Code in griptape/tasks/actions_subtask.py

```
def add_to_prompt_stack(self, stack: PromptStack) -> None:
    from griptape.tasks import PromptTask

    if isinstance(self.origin_task, PromptTask) and self.origin_task.prompt_driver.
↪use_native_tools:
        action_calls = [
            ToolAction(name=action.name, path=action.path, tag=action.tag,␣
↪input=action.input)
            for action in self.actions
        ]
        action_results = [
            ToolAction(
```

(continues on next page)

(continued from previous page)

```
                name=action.name,
                path=action.path,
                tag=action.tag,
                output=action.output if action.output is not None else self.output,
            )
            for action in self.actions
        ]

        stack.add_assistant_message(
            ListArtifact(
                [
                    *([TextArtifact(self.thought)] if self.thought else []),
                    *[ActionArtifact(a) for a in action_calls],
                ],
            ),
        )
        stack.add_user_message(
            ListArtifact(
                [
                    *[ActionArtifact(a) for a in action_results],
                    *([] if self.output else [TextArtifact("Please keep going")]),
                ],
            ),
        )
    else:
        stack.add_assistant_message(self.generate_assistant_subtask_template(self))
        stack.add_user_message(self.generate_user_subtask_template(self))
```

after_run()

Source Code in griptape/tasks/actions_subtask.py

```
def after_run(self) -> None:
    response = self.output.to_text() if isinstance(self.output, BaseArtifact) else␣
↪str(self.output)

    EventBus.publish_event(
        FinishActionsSubtaskEvent(
            task_id=self.id,
            task_parent_ids=self.parent_ids,
            task_child_ids=self.child_ids,
            task_input=self.input,
            task_output=self.output,
            subtask_parent_task_id=self.origin_task.id,
            subtask_thought=self.thought,
            subtask_actions=self.actions_to_dicts(),
        ),
    )
    logger.info("%s %s\nResponse: %s", self.__class__.__name__, self.id, response)
```

attach_to(parent_task)

Source Code in griptape/tasks/actions_subtask.py

```
def attach_to(self, parent_task: BaseTask) -> None:
    super().attach_to(parent_task)
    self.structure = parent_task.structure

    task_input = self.input
    try:
        if isinstance(task_input, TextArtifact) and task_input.meta.get("is_react_
↪prompt", False):
            self.__init_from_prompt(task_input.to_text())
        else:
            self.__init_from_artifact(task_input)

        # If StructuredOutputTool was used, treat the input to it as the output of␣
↪the subtask.
        structured_outputs = [a for a in self.actions if isinstance(a.tool,␣
↪StructuredOutputTool)]
        if structured_outputs:
            output_values = [JsonArtifact(a.input["values"]) for a in structured_
↪outputs]
            if len(structured_outputs) > 1:
                self.output = ListArtifact(output_values)
            else:
                self.output = output_values[0]
    except Exception as e:
        logger.error("Subtask %s\nError parsing tool action: %s", self.origin_task.id,
↪ e)

        self.output = ErrorArtifact(f"ToolAction input parsing error: {e}",␣
↪exception=e)
```

before_run()

Source Code in griptape/tasks/actions_subtask.py

```
def before_run(self) -> None:
    EventBus.publish_event(
        StartActionsSubtaskEvent(
            task_id=self.id,
            task_parent_ids=self.parent_ids,
            task_child_ids=self.child_ids,
            task_input=self.input,
            task_output=self.output,
            subtask_parent_task_id=self.origin_task.id,
            subtask_thought=self.thought,
            subtask_actions=self.actions_to_dicts(),
        ),
    )

    parts = [
        f"{self.__class__.__name__} {self.id}",
```

(continues on next page)

(continued from previous page)

```
        *([f"\nThought: {self.thought}"] if self.thought else []),
        f"\nActions: {self.actions_to_json()}",
    ]
    logger.info("".join(parts))
```

default_generate_assistant_subtask_template(subtask)

Source Code in griptape/tasks/actions_subtask.py

```
def default_generate_assistant_subtask_template(self, subtask: ActionsSubtask) -> str:
    return J2("tasks/prompt_task/assistant_actions_subtask.j2").render(
        stop_sequence=self.response_stop_sequence,
        subtask=subtask,
    )
```

default_generate_user_subtask_template(subtask)

Source Code in griptape/tasks/actions_subtask.py

```
def default_generate_user_subtask_template(self, subtask: ActionsSubtask) -> str:
    return J2("tasks/prompt_task/user_actions_subtask.j2").render(
        stop_sequence=self.response_stop_sequence,
        subtask=subtask,
    )
```

run_action(action)

Source Code in griptape/tasks/actions_subtask.py

```
def run_action(self, action: ToolAction) -> tuple[str, BaseArtifact]:
    if action.tool is not None:
        if action.path is not None:
            output = action.tool.run(getattr(action.tool, action.path), self, action)
        else:
            output = ErrorArtifact("action path not found")
    else:
        output = ErrorArtifact("action name not found")
    action.output = output

    return action.tag, output
```

run_actions(actions)

Source Code in griptape/tasks/actions_subtask.py

```
def run_actions(self, actions: list[ToolAction]) -> list[tuple[str, BaseArtifact]]:
    with self.create_futures_executor() as futures_executor:
        return utils.execute_futures_list(
            [futures_executor.submit(with_contextvars(self.run_action), a) for a in␣
↪actions]
        )
```

try_run()

Source Code in griptape/tasks/actions_subtask.py

```
def try_run(self) -> ListArtifact | ErrorArtifact:
    try:
        if any(isinstance(a.output, ErrorArtifact) for a in self.actions):
            errors = [a.output.value for a in self.actions if isinstance(a.output,␣
↪ErrorArtifact)]

            self.output = ErrorArtifact("\n\n".join(errors))
        else:
            results = self.run_actions(self.actions)

            actions_output = []
            for result in results:
                tag, output = result
                output.name = f"{tag} output"

                actions_output.append(output)
            self.output = ListArtifact(actions_output)
    except Exception as e:
        logger.debug("Subtask %s\n%s", self.id, e)

        self.output = ErrorArtifact(str(e), exception=e)
    if self.output is not None:
        return self.output
    return ErrorArtifact("no tool output")
```

AssistantTask

Bases: *BaseTextInputTask[TextArtifact* “)

Attributes

Name	Type	Description
<a href="./#griptape.tasks.AssistantTask.assistant_driver" title="assistant_driver = field(kw_only=True) class-attribute instance-attribute (griptape.tasks.AssistantTask.assistant_driver)"><code class="language-python">	<a href="../drivers/assistant#griptape.drivers.assistant.BaseAssistantDriver" title="BaseAssistantDriver (griptape.drivers.assistant.BaseAssistantDriver)"><code class="language-python">BaseAssistantDriver</code>	Driver to run the Assistant.

Source Code in griptape/tasks/assistant_task.py

```
@define
class AssistantTask(BaseTextInputTask[TextArtifact]):
```

(continues on next page)

(continued from previous page)

```
    """Task to run an Assistant.

    Attributes:
        assistant_driver: Driver to run the Assistant.
    """

    assistant_driver: BaseAssistantDriver = field(kw_only=True)

    def try_run(self) -> TextArtifact:
        return self.assistant_driver.run(self.input)
```

- `assistant_driver = field(kw_only=True)` class-attribute instance-attribute

try_run()

Source Code in griptape/tasks/assistant_task.py

```
def try_run(self) -> TextArtifact:
    return self.assistant_driver.run(self.input)
```

AudioTranscriptionTask

Bases: *BaseAudioInputTask[TextArtifact* “)

Source Code in griptape/tasks/audio_transcription_task.py

```
@define
class AudioTranscriptionTask(BaseAudioInputTask[TextArtifact]):
    audio_transcription_driver: BaseAudioTranscriptionDriver = field(
        default=Factory(lambda: Defaults.drivers_config.audio_transcription_driver),
        kw_only=True,
    )

    def try_run(self) -> TextArtifact:
        return self.audio_transcription_driver.run(self.input)
```

- `audio_transcription_driver = field(default=Factory(lambda: Defaults.drivers_config.audio_transcription_driver), kw_only=True)` class-attribute instance-attribute

try_run()

Source Code in griptape/tasks/audio_transcription_task.py

```
def try_run(self) -> TextArtifact:
    return self.audio_transcription_driver.run(self.input)
```

BaseAudioGenerationTask

Bases: *ArtifactFileOutputMixin* “) , *RuleMixin* ”) , *BaseTask[AudioArtifact* “) , ABC

Source Code in griptape/tasks/base_audio_generation_task.py

```
@define
class BaseAudioGenerationTask(ArtifactFileOutputMixin, RuleMixin,␣
↪BaseTask[AudioArtifact], ABC):
    def before_run(self) -> None:
        super().before_run()

        logger.info("%s %s\nInput: %s", self.__class__.__name__, self.id, self.input.
↪to_text())

    def after_run(self) -> None:
        super().after_run()

        logger.info(
            "%s %s\nOutput: %s",
            self.__class__.__name__,
            self.id,
            self.output.to_text() if self.output is not None else "",
        )
```

after_run()

Source Code in griptape/tasks/base_audio_generation_task.py

```
def after_run(self) -> None:
    super().after_run()

    logger.info(
        "%s %s\nOutput: %s",
        self.__class__.__name__,
        self.id,
        self.output.to_text() if self.output is not None else "",
    )
```

before_run()

Source Code in griptape/tasks/base_audio_generation_task.py

```
def before_run(self) -> None:
    super().before_run()

    logger.info("%s %s\nInput: %s", self.__class__.__name__, self.id, self.input.to_
↪text())
```

BaseImageGenerationTask

Bases: *ArtifactFileOutputMixin* “) , *RuleMixin* ”) , *BaseTask[ImageArtifact* “) , ABC

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">list[Ruleset]</code>	List of negatively-weighted rulesets applied to the text prompt, if supported by the driver.
<code class="language-python">	<code class="language-python">list[Rule]</code>	List of negatively-weighted rules applied to the text prompt, if supported by the driver.
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.BaseImageGenerationTask.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk in output_dir.

continues on next page

Table 366 – continued from previous page

Name	Type	Description
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.BaseImageGenerationTask.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk as output_file.

Source Code in griptape/tasks/base_image_generation_task.py

```
@define
class BaseImageGenerationTask(ArtifactFileOutputMixin, RuleMixin,␣
↪BaseTask[ImageArtifact], ABC):
    """Provides a base class for image generation-related tasks.

    Attributes:
        negative_rulesets: List of negatively-weighted rulesets applied to the text␣
↪prompt, if supported by the driver.
        negative_rules: List of negatively-weighted rules applied to the text prompt,␣
↪if supported by the driver.
        output_dir: If provided, the generated image will be written to disk in␣
↪output_dir.
        output_file: If provided, the generated image will be written to disk as␣
↪output_file.
    """

    DEFAULT_NEGATIVE_RULESET_NAME = "Negative Ruleset"

    image_generation_driver: BaseImageGenerationDriver = field(
        default=Factory(lambda: Defaults.drivers_config.image_generation_driver),
        kw_only=True,
    )
    _negative_rulesets: list[Ruleset] = field(factory=list, kw_only=True, alias=
↪"negative_rulesets")
    negative_rules: list[Rule] = field(factory=list, kw_only=True)

    @property
    def negative_rulesets(self) -> list[Ruleset]:
        negative_rulesets = self._negative_rulesets
```

(continues on next page)

(continued from previous page)

```
        if self.negative_rules:
            negative_rulesets.append(Ruleset(name=self.DEFAULT_NEGATIVE_RULESET_NAME,␣
↪rules=self.negative_rules))

        return negative_rulesets

    def _read_from_file(self, path: str) -> ImageArtifact:
        logger.info("Reading image from %s", os.path.abspath(path))
        return ImageLoader().load(Path(path))

    def _get_prompts(self, prompt: str) -> list[str]:
        return [prompt, *[rule.value for ruleset in self.rulesets for rule in ruleset.
↪rules]]

    def _get_negative_prompts(self) -> list[str]:
        return [rule.value for ruleset in self.negative_rulesets for rule in ruleset.
↪rules]
```

- `DEFAULT_NEGATIVE_RULESET_NAME = 'Negative Ruleset'` class-attribute instance-attribute
- `_negative_rulesets = field(factory=list, kw_only=True, alias='negative_rulesets')` class-attribute instance-attribute
- `image_generation_driver = field(default=Factory(lambda: Defaults.drivers_config.image_generation_driver), kw_only=True)` class-attribute instance-attribute
- `negative_rules = field(factory=list, kw_only=True)` class-attribute instance-attribute
- `negative_rulesets` property

_get_negative_prompts()

Source Code in griptape/tasks/base_image_generation_task.py

```
def _get_negative_prompts(self) -> list[str]:
    return [rule.value for ruleset in self.negative_rulesets for rule in ruleset.
↪rules]
```

_get_prompts(prompt)

Source Code in griptape/tasks/base_image_generation_task.py

```
def _get_prompts(self, prompt: str) -> list[str]:
    return [prompt, *[rule.value for ruleset in self.rulesets for rule in ruleset.
↪rules]]
```

_read_from_file(path)

Source Code in griptape/tasks/base_image_generation_task.py

```
def _read_from_file(self, path: str) -> ImageArtifact:
    logger.info("Reading image from %s", os.path.abspath(path))
    return ImageLoader().load(Path(path))
```

BaseSubtask

Bases: *BaseTask[T “)*

Source Code in griptape/tasks/base_subtask.py

```
@define
class BaseSubtask(BaseTask[T]):
    @property
    def origin_task(self) -> BaseTask:
        if self._origin_task is not None:
            return self._origin_task
        raise Exception("ActionSubtask has no origin task.")

    @property
    def parents(self) -> list[BaseTask]:
        if isinstance(self.origin_task, ActionsSubtaskOriginMixin):
            return [self.origin_task.find_subtask(parent_id) for parent_id in self.
↪parent_ids]
        raise Exception("ActionSubtask must be attached to a Task that implements␣
↪ActionSubtaskOriginMixin.")

    @property
    def children(self) -> list[BaseTask]:
        if isinstance(self.origin_task, ActionsSubtaskOriginMixin):
            return [self.origin_task.find_subtask(child_id) for child_id in self.
↪child_ids]
        raise Exception("ActionSubtask must be attached to a Task that implements␣
↪ActionSubtaskOriginMixin.")

    def add_child(self, child: BaseTask) -> BaseTask:
        if child.id not in self.child_ids:
            self.child_ids.append(child.id)
        return child

    def add_parent(self, parent: BaseTask) -> BaseTask:
        if parent.id not in self.parent_ids:
            self.parent_ids.append(parent.id)
        return parent

    def attach_to(self, parent_task: BaseTask) -> None:
        self._origin_task = parent_task

    @abstractmethod
    def add_to_prompt_stack(self, stack: PromptStack) -> None: ...
```

- `children` property
- `origin_task` property
- `parents` property

add_child(child)

Source Code in griptape/tasks/base_subtask.py

```
def add_child(self, child: BaseTask) -> BaseTask:
    if child.id not in self.child_ids:
        self.child_ids.append(child.id)
    return child
```

add_parent(parent)

Source Code in griptape/tasks/base_subtask.py

```
def add_parent(self, parent: BaseTask) -> BaseTask:
    if parent.id not in self.parent_ids:
        self.parent_ids.append(parent.id)
    return parent
```

add_to_prompt_stack(stack)abstractmethod

Source Code in griptape/tasks/base_subtask.py

```
@abstractmethod
def add_to_prompt_stack(self, stack: PromptStack) -> None: ...
```

attach_to(parent_task)

Source Code in griptape/tasks/base_subtask.py

```
def attach_to(self, parent_task: BaseTask) -> None:
    self._origin_task = parent_task
```

BaseTask

Bases: *FuturesExecutorMixin* “) , *SerializableMixin* ”) , *RunnableMixin['BaseTask'* “) , ABC, *Generic[T*

Source Code in griptape/tasks/base_task.py

```
@define
class BaseTask(FuturesExecutorMixin, SerializableMixin, RunnableMixin["BaseTask"],␣
↪ABC, Generic[T]):
    class State(Enum):
        PENDING = 1
        RUNNING = 2
        FINISHED = 3
        SKIPPED = 4

    id: str = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata=
↪{"serializable": True})
    state: State = field(default=State.PENDING, kw_only=True, metadata={"serializable
↪": True})
    parent_ids: list[str] = field(factory=list, kw_only=True, metadata={"serializable
↪": True})
    child_ids: list[str] = field(factory=list, kw_only=True, metadata={"serializable
```

(continues on next page)

(continued from previous page)

```
↪": True})
    max_meta_memory_entries: Optional[int] = field(default=20, kw_only=True, metadata=
↪{"serializable": True})
    structure: Optional[Structure] = field(default=None, kw_only=True)

    output: Optional[T] = field(default=None, init=False)
    context: dict[str, Any] = field(factory=dict, kw_only=True, metadata={
↪"serializable": True})
    _execution_args: tuple = field(factory=tuple, init=False)

    @property
    def execution_args(self) -> tuple:
        return self._execution_args

    def __rshift__(self, other: BaseTask | list[BaseTask]) -> BaseTask |␣
↪list[BaseTask]:
        if isinstance(other, list):
            self.add_children(other)
        else:
            self.add_child(other)

        return other

    def __lshift__(self, other: BaseTask | list[BaseTask]) -> BaseTask |␣
↪list[BaseTask]:
        if isinstance(other, list):
            self.add_parents(other)
        else:
            self.add_parent(other)

        return other

    def __attrs_post_init__(self) -> None:
        if self.structure is not None:
            self.structure.add_task(self)

    @property
    @abstractmethod
    def input(self) -> BaseArtifact: ...

    @property
    def parents(self) -> list[BaseTask]:
        if self.structure is not None:
            return [self.structure.find_task(parent_id) for parent_id in self.parent_
↪ids]
        raise ValueError("Structure must be set to access parents")

    @property
    def children(self) -> list[BaseTask]:
        if self.structure is not None:
            return [self.structure.find_task(child_id) for child_id in self.child_ids]
        raise ValueError("Structure must be set to access children")
```

(continues on next page)

(continued from previous page)

```
    @property
    def parent_outputs(self) -> dict[str, BaseArtifact]:
        return {parent.id: parent.output for parent in self.parents if parent.output}

    @property
    def parents_output_text(self) -> str:
        return "\n".join([parent.output.to_text() for parent in self.parents if␣
↪parent.output])

    @property
    def meta_memories(self) -> list[BaseMetaEntry]:
        if self.structure is not None and self.structure.meta_memory:
            if self.max_meta_memory_entries:
                return self.structure.meta_memory.entries[: self.max_meta_memory_
↪entries]
            return self.structure.meta_memory.entries
        return []

    def __str__(self) -> str:
        return str(self.output.value) if self.output is not None else ""

    def add_parents(self, parents: list[BaseTask]) -> None:
        for parent in parents:
            self.add_parent(parent)

    def add_parent(self, parent: BaseTask) -> BaseTask:
        if parent.id not in self.parent_ids:
            self.parent_ids.append(parent.id)

        if self.id not in parent.child_ids:
            parent.child_ids.append(self.id)

        if self.structure is not None and parent not in self.structure.tasks:
            self.structure.add_task(parent)

        return self

    def add_children(self, children: list[BaseTask]) -> None:
        for child in children:
            self.add_child(child)

    def add_child(self, child: BaseTask) -> BaseTask:
        if child.id not in self.child_ids:
            self.child_ids.append(child.id)

        if self.id not in child.parent_ids:
            child.parent_ids.append(self.id)

        if self.structure is not None and child not in self.structure.tasks:
            self.structure.add_task(child)
```

(continues on next page)

(continued from previous page)

```
        return self

    def preprocess(self, structure: Structure) -> BaseTask:
        self.structure = structure

        return self

    def is_pending(self) -> bool:
        return self.state == BaseTask.State.PENDING

    def is_finished(self) -> bool:
        return self.state == BaseTask.State.FINISHED

    def is_running(self) -> bool:
        return self.state == BaseTask.State.RUNNING

    def is_skipped(self) -> bool:
        return self.state == BaseTask.State.SKIPPED

    def before_run(self) -> None:
        super().before_run()
        if self.structure is not None:
            EventBus.publish_event(
                StartTaskEvent(
                    task_id=self.id,
                    task_parent_ids=self.parent_ids,
                    task_child_ids=self.child_ids,
                    task_input=self.input,
                    task_output=self.output,
                ),
            )

    def run(self, *args) -> T:
        try:
            self._execution_args = args

            self.state = BaseTask.State.RUNNING

            self.before_run()

            self.output = self.try_run()

            self.after_run()
        except Exception as e:
            logger.exception("%s %s\n%s", self.__class__.__name__, self.id, e)

            self.output = cast("T", ErrorArtifact(str(e), exception=e))
        finally:
            self.state = BaseTask.State.FINISHED

        return self.output
```

(continues on next page)

(continued from previous page)

```
    def after_run(self) -> None:
        super().after_run()
        if self.structure is not None:
            EventBus.publish_event(
                FinishTaskEvent(
                    task_id=self.id,
                    task_parent_ids=self.parent_ids,
                    task_child_ids=self.child_ids,
                    task_input=self.input,
                    task_output=self.output,
                ),
            )

    def can_run(self) -> bool:
        # If this Task has been skipped or is not pending, it should not run
        if self.is_skipped() or not self.is_pending():
            return False

        # If this Task has parents, and _all_ of them are skipped, it should not run
        if self.parents and all(parent.is_skipped() for parent in self.parents):
            self.state = BaseTask.State.SKIPPED
            return False

        # If _all_ this Tasks unskipped parents are finished, it should run
        unskipped_parents = [parent for parent in self.parents if not parent.is_
↪skipped()]

        return all(parent.is_finished() for parent in unskipped_parents)

    def reset(self) -> BaseTask:
        self.state = BaseTask.State.PENDING
        self.output = None
        self._execution_args = ()

        return self

    @abstractmethod
    def try_run(self) -> T: ...

    @property
    def full_context(self) -> dict[str, Any]:
        # Need to deep copy so that the serialized context doesnt contain non-
↪serializable data
        context = deepcopy(self.context)
        if self.structure is None:
            context.update({"args": self._execution_args})
        else:
            context.update(self.structure.context(self))

        return context
```

- `_execution_args = field(factory=tuple, init=False)` class-attribute instance-attribute

- `child_ids = field(factory=list, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `children` property
- `context = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `execution_args` property
- `full_context` property
- `id = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `input` abstractmethod property
- `max_meta_memory_entries = field(default=20, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `meta_memories` property
- `output = field(default=None, init=False)` class-attribute instance-attribute
- `parent_ids = field(factory=list, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `parent_outputs` property
- `parents` property
- `parents_output_text` property
- `state = field(default=State.PENDING, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structure = field(default=None, kw_only=True)` class-attribute instance-attribute

State

Bases:

Enum

Source Code in griptape/tasks/base_task.py

```
class State(Enum):
    PENDING = 1
    RUNNING = 2
    FINISHED = 3
    SKIPPED = 4
```

- `FINISHED = 3` class-attribute instance-attribute
- `PENDING = 1` class-attribute instance-attribute
- `RUNNING = 2` class-attribute instance-attribute
- `SKIPPED = 4` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/tasks/base_task.py

```
def __attrs_post_init__(self) -> None:
    if self.structure is not None:
        self.structure.add_task(self)
```

lshift(other)

Source Code in griptape/tasks/base_task.py

```
def __lshift__(self, other: BaseTask | list[BaseTask]) -> BaseTask | list[BaseTask]:
    if isinstance(other, list):
        self.add_parents(other)
    else:
        self.add_parent(other)

    return other
```

rshift(other)

Source Code in griptape/tasks/base_task.py

```
def __rshift__(self, other: BaseTask | list[BaseTask]) -> BaseTask | list[BaseTask]:
    if isinstance(other, list):
        self.add_children(other)
    else:
        self.add_child(other)

    return other
```

str()

Source Code in griptape/tasks/base_task.py

```
def __str__(self) -> str:
    return str(self.output.value) if self.output is not None else ""
```

add_child(child)

Source Code in griptape/tasks/base_task.py

```
def add_child(self, child: BaseTask) -> BaseTask:
    if child.id not in self.child_ids:
        self.child_ids.append(child.id)

    if self.id not in child.parent_ids:
        child.parent_ids.append(self.id)

    if self.structure is not None and child not in self.structure.tasks:
        self.structure.add_task(child)

    return self
```

add_children(children)

Source Code in griptape/tasks/base_task.py

```
def add_children(self, children: list[BaseTask]) -> None:
    for child in children:
        self.add_child(child)
```

add_parent(parent)

Source Code in griptape/tasks/base_task.py

```
def add_parent(self, parent: BaseTask) -> BaseTask:
    if parent.id not in self.parent_ids:
        self.parent_ids.append(parent.id)

    if self.id not in parent.child_ids:
        parent.child_ids.append(self.id)

    if self.structure is not None and parent not in self.structure.tasks:
        self.structure.add_task(parent)

    return self
```

add_parents(parents)

Source Code in griptape/tasks/base_task.py

```
def add_parents(self, parents: list[BaseTask]) -> None:
    for parent in parents:
        self.add_parent(parent)
```

after_run()

Source Code in griptape/tasks/base_task.py

```
def after_run(self) -> None:
    super().after_run()
    if self.structure is not None:
        EventBus.publish_event(
            FinishTaskEvent(
                task_id=self.id,
                task_parent_ids=self.parent_ids,
                task_child_ids=self.child_ids,
                task_input=self.input,
                task_output=self.output,
            ),
        )
```

before_run()

Source Code in griptape/tasks/base_task.py

```
def before_run(self) -> None:
    super().before_run()
    if self.structure is not None:
        EventBus.publish_event(
            StartTaskEvent(
                task_id=self.id,
                task_parent_ids=self.parent_ids,
                task_child_ids=self.child_ids,
                task_input=self.input,
                task_output=self.output,
            ),
        )
```

can_run()

Source Code in griptape/tasks/base_task.py

```
def can_run(self) -> bool:
    # If this Task has been skipped or is not pending, it should not run
    if self.is_skipped() or not self.is_pending():
        return False

    # If this Task has parents, and _all_ of them are skipped, it should not run
    if self.parents and all(parent.is_skipped() for parent in self.parents):
        self.state = BaseTask.State.SKIPPED
        return False

    # If _all_ this Tasks unskipped parents are finished, it should run
    unskipped_parents = [parent for parent in self.parents if not parent.is_skipped()]

    return all(parent.is_finished() for parent in unskipped_parents)
```

is_finished()

Source Code in griptape/tasks/base_task.py

```
def is_finished(self) -> bool:
    return self.state == BaseTask.State.FINISHED
```

is_pending()

Source Code in griptape/tasks/base_task.py

```
def is_pending(self) -> bool:
    return self.state == BaseTask.State.PENDING
```

is_running()

Source Code in griptape/tasks/base_task.py

```
def is_running(self) -> bool:
    return self.state == BaseTask.State.RUNNING
```

is_skipped()

Source Code in griptape/tasks/base_task.py

```
def is_skipped(self) -> bool:
    return self.state == BaseTask.State.SKIPPED
```

preprocess(structure)

Source Code in griptape/tasks/base_task.py

```
def preprocess(self, structure: Structure) -> BaseTask:
    self.structure = structure

    return self
```

reset()

Source Code in griptape/tasks/base_task.py

```
def reset(self) -> BaseTask:
    self.state = BaseTask.State.PENDING
    self.output = None
    self._execution_args = ()

    return self
```

run(*args)

Source Code in griptape/tasks/base_task.py

```
def run(self, *args) -> T:
    try:
        self._execution_args = args

        self.state = BaseTask.State.RUNNING

        self.before_run()

        self.output = self.try_run()

        self.after_run()
    except Exception as e:
        logger.exception("%s %s\n%s", self.__class__.__name__, self.id, e)

        self.output = cast("T", ErrorArtifact(str(e), exception=e))
    finally:
        self.state = BaseTask.State.FINISHED

    return self.output
```

try_run()abstractmethod

Source Code in griptape/tasks/base_task.py

```
@abstractmethod
def try_run(self) -> T: ...
```

BaseTextInputTask

Bases: *RuleMixin* “) , *BaseTask[T* ”) , ABC

Source Code in griptape/tasks/base_text_input_task.py

```
@define
class BaseTextInputTask(RuleMixin, BaseTask[T], ABC):
    DEFAULT_INPUT_TEMPLATE = "{{ args[0] }}"

    _input: Union[str, TextArtifact, Callable[[BaseTask], TextArtifact]] = field(
        default=DEFAULT_INPUT_TEMPLATE,
        alias="input",
    )

    @property
    def input(self) -> TextArtifact:
        if isinstance(self._input, TextArtifact):
            return self._input
        if isinstance(self._input, Callable):
            return self._input(self)
        return TextArtifact(J2().render_from_string(self._input, **self.full_context))

    @input.setter
    def input(self, value: str | TextArtifact | Callable[[BaseTask], TextArtifact]) ->
↪ None:
        self._input = value

    def before_run(self) -> None:
        super().before_run()

        logger.info("%s %s\nInput: %s", self.__class__.__name__, self.id, self.input.
↪to_text())

    def after_run(self) -> None:
        super().after_run()

        logger.info(
            "%s %s\nOutput: %s",
            self.__class__.__name__,
            self.id,
            self.output.to_text() if self.output is not None else "",
        )
```

- `DEFAULT_INPUT_TEMPLATE = '{{ args[0] }}'` class-attribute instance-attribute
- `_input = field(default=DEFAULT_INPUT_TEMPLATE, alias='input')` class-attribute instance-attribute

- `input` property writable

after_run()

Source Code in griptape/tasks/base_text_input_task.py

```
def after_run(self) -> None:
    super().after_run()

    logger.info(
        "%s %s\nOutput: %s",
        self.__class__.__name__,
        self.id,
        self.output.to_text() if self.output is not None else "",
    )
```

before_run()

Source Code in griptape/tasks/base_text_input_task.py

```
def before_run(self) -> None:
    super().before_run()

    logger.info("%s %s\nInput: %s", self.__class__.__name__, self.id, self.input.to_
↪text())
```

BranchTask

Bases: *CodeExecutionTask[Union[InfoArtifact, ListArtifact[InfoArtifact* “)

Source Code in griptape/tasks/branch_task.py

```
@define
class BranchTask(CodeExecutionTask[Union[InfoArtifact, ListArtifact[InfoArtifact]]]):
    on_run: Callable[[BranchTask], Union[InfoArtifact, ListArtifact[InfoArtifact]]] =␣
↪field(kw_only=True)

    def try_run(self) -> InfoArtifact | ListArtifact[InfoArtifact]:
        result = self.on_run(self)

        if isinstance(result, ListArtifact):
            branch_task_ids = {artifact.value for artifact in result}
        else:
            branch_task_ids = {result.value}

        if not all(branch_task_id in self.child_ids for branch_task_id in branch_task_
↪ids):
            raise ValueError(f"Branch task returned invalid child task id {branch_
↪task_ids}")

        if self.structure is not None:
            children_to_skip = [child for child in self.children if child.id not in␣
↪branch_task_ids]
            for child in children_to_skip:
```

(continues on next page)

(continued from previous page)

```
                child.state = BaseTask.State.SKIPPED

        return result
```

- `on_run = field(kw_only=True)` class-attribute instance-attribute

try_run()

Source Code in griptape/tasks/branch_task.py

```
def try_run(self) -> InfoArtifact | ListArtifact[InfoArtifact]:
    result = self.on_run(self)

    if isinstance(result, ListArtifact):
        branch_task_ids = {artifact.value for artifact in result}
    else:
        branch_task_ids = {result.value}

    if not all(branch_task_id in self.child_ids for branch_task_id in branch_task_
↪ids):
        raise ValueError(f"Branch task returned invalid child task id {branch_task_
↪ids}")

    if self.structure is not None:
        children_to_skip = [child for child in self.children if child.id not in␣
↪branch_task_ids]
        for child in children_to_skip:
            child.state = BaseTask.State.SKIPPED

    return result
```

CodeExecutionTask

Bases: *BaseTask[T “)*

Source Code in griptape/tasks/code_execution_task.py

```
@define
class CodeExecutionTask(BaseTask[T]):
    DEFAULT_INPUT_TEMPLATE = "{{ args[0] }}"
    _input: Union[str, TextArtifact, Callable[[BaseTask], TextArtifact]] = field(
        default=DEFAULT_INPUT_TEMPLATE,
        alias="input",
    )
    on_run: Callable[[CodeExecutionTask[T]], T] = field(kw_only=True)

    @property
    def input(self) -> TextArtifact:
        if isinstance(self._input, TextArtifact):
            return self._input
        if callable(self._input):
            return self._input(self)
        return TextArtifact(J2().render_from_string(self._input, **self.full_context))
```

(continues on next page)

(continued from previous page)

```
    def try_run(self) -> T:
        return self.on_run(self)
```

- `DEFAULT_INPUT_TEMPLATE = '{{ args[0] }}'` class-attribute instance-attribute
- `_input = field(default=DEFAULT_INPUT_TEMPLATE, alias='input')` class-attribute instance-attribute
- `input` property
- `on_run = field(kw_only=True)` class-attribute instance-attribute

try_run()

Source Code in griptape/tasks/code_execution_task.py

```
def try_run(self) -> T:
    return self.on_run(self)
```

ExtractionTask

Bases: *BaseTextInputTask[ListArtifact* “)

Source Code in griptape/tasks/extraction_task.py

```
@define
class ExtractionTask(BaseTextInputTask[ListArtifact]):
    extraction_engine: BaseExtractionEngine = field(kw_only=True)
    args: dict = field(kw_only=True, factory=dict)

    def try_run(self) -> ListArtifact:
        return self.extraction_engine.extract_artifacts(ListArtifact([self.input]),␣
↪rulesets=self.rulesets, **self.args)
```

- `args = field(kw_only=True, factory=dict)` class-attribute instance-attribute
- `extraction_engine = field(kw_only=True)` class-attribute instance-attribute

try_run()

Source Code in griptape/tasks/extraction_task.py

```
def try_run(self) -> ListArtifact:
    return self.extraction_engine.extract_artifacts(ListArtifact([self.input]),␣
↪rulesets=self.rulesets, **self.args)
```

InpaintingImageGenerationTask

Bases: *BaseImageGenerationTask* “)

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">BaseImageGenerationDrive	The driver used to generate the image.
<code class="language-python">	<code class="language-python">list[Ruleset]</code>	List of negatively-weighted rulesets applied to the text prompt, if supported by the driver.
<code class="language-python">	<code class="language-python">list[Rule]</code>	List of negatively-weighted rules applied to the text prompt, if supported by the driver.

continues on next page

Table 367 – continued from previous page

Name	Type	Description
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.InpaintingImageGenerationTask.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk in output_dir.
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.InpaintingImageGenerationTask.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk as output_file.

Source Code in griptape/tasks/inpainting_image_generation_task.py

```
@define
class InpaintingImageGenerationTask(BaseImageGenerationTask):
    """A task that modifies a select region within an image using a mask.

    Accepts a text prompt, image, and mask as
    input in one of the following formats:
    - tuple of (template string, ImageArtifact, ImageArtifact)
    - tuple of (TextArtifact, ImageArtifact, ImageArtifact)
    - Callable that returns a tuple of (TextArtifact, ImageArtifact, ImageArtifact).

    Attributes:
```

(continues on next page)

(continued from previous page)

```
        image_generation_driver: The driver used to generate the image.
        negative_rulesets: List of negatively-weighted rulesets applied to the text␣
↪prompt, if supported by the driver.
        negative_rules: List of negatively-weighted rules applied to the text prompt,␣
↪if supported by the driver.
        output_dir: If provided, the generated image will be written to disk in␣
↪output_dir.
        output_file: If provided, the generated image will be written to disk as␣
↪output_file.
    """

    _input: Union[
        tuple[Union[str, TextArtifact], ImageArtifact, ImageArtifact],␣
↪Callable[[BaseTask], ListArtifact], ListArtifact
    ] = field(default=None, alias="input")

    @property
    def input(self) -> ListArtifact:
        if isinstance(self._input, ListArtifact):
            return self._input
        if isinstance(self._input, tuple):
            if isinstance(self._input[0], TextArtifact):
                input_text = self._input[0]
            else:
                input_text = TextArtifact(J2().render_from_string(self._input[0],␣
↪**self.full_context))

            return ListArtifact([input_text, self._input[1], self._input[2]])
        if isinstance(self._input, Callable):
            return self._input(self)
        raise ValueError("Input must be a tuple of (text, image, mask) or a callable␣
↪that returns such a tuple.")

    @input.setter
    def input(
        self,
        value: tuple[str | TextArtifact, ImageArtifact, ImageArtifact] |␣
↪Callable[[BaseTask], ListArtifact],
    ) -> None:
        self._input = value

    def try_run(self) -> ImageArtifact:
        prompt_artifact = self.input[0]

        image_artifact = self.input[1]
        if not isinstance(image_artifact, ImageArtifact):
            raise ValueError("Image must be an ImageArtifact.")

        mask_artifact = self.input[2]
        if not isinstance(mask_artifact, ImageArtifact):
            raise ValueError("Mask must be an ImageArtifact.")
```

(continues on next page)

(continued from previous page)

```
        output_image_artifact = self.image_generation_driver.run_image_inpainting(
            prompts=self._get_prompts(prompt_artifact.to_text()),
            negative_prompts=self._get_negative_prompts(),
            image=image_artifact,
            mask=mask_artifact,
        )

        if self.output_dir or self.output_file:
            self._write_to_file(output_image_artifact)

        return output_image_artifact
```

- `_input = field(default=None, alias='input')` class-attribute instance-attribute
- `input` property writable

try_run()

Source Code in griptape/tasks/inpainting_image_generation_task.py

```
def try_run(self) -> ImageArtifact:
    prompt_artifact = self.input[0]

    image_artifact = self.input[1]
    if not isinstance(image_artifact, ImageArtifact):
        raise ValueError("Image must be an ImageArtifact.")

    mask_artifact = self.input[2]
    if not isinstance(mask_artifact, ImageArtifact):
        raise ValueError("Mask must be an ImageArtifact.")

    output_image_artifact = self.image_generation_driver.run_image_inpainting(
        prompts=self._get_prompts(prompt_artifact.to_text()),
        negative_prompts=self._get_negative_prompts(),
        image=image_artifact,
        mask=mask_artifact,
    )

    if self.output_dir or self.output_file:
        self._write_to_file(output_image_artifact)

    return output_image_artifact
```

OutpaintingImageGenerationTask

Bases: *BaseImageGenerationTask* “)

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">BaseImageGenerationDrive	The engine used to generate the image.
<code class="language-python">	<code class="language-python">list[Ruleset]</code>	List of negatively-weighted rulesets applied to the text prompt, if supported by the driver.
<code class="language-python">	<code class="language-python">list[Rule]</code>	List of negatively-weighted rules applied to the text prompt, if supported by the driver.

continues on next page

Table 368 – continued from previous page

Name	Type	Description
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.OutpaintingImageGenerationTask.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk in output_dir.
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.OutpaintingImageGenerationTask.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk as output_file.

Source Code in griptape/tasks/outpainting_image_generation_task.py

```
@define
class OutpaintingImageGenerationTask(BaseImageGenerationTask):
    """A task that modifies an image outside the bounds of a mask.

    Accepts a text prompt, image, and mask as
    input in one of the following formats:
    - tuple of (template string, ImageArtifact, ImageArtifact)
    - tuple of (TextArtifact, ImageArtifact, ImageArtifact)
    - Callable that returns a tuple of (TextArtifact, ImageArtifact, ImageArtifact).

    Attributes:
```

(continues on next page)

(continued from previous page)

```
        image_generation_driver: The engine used to generate the image.
        negative_rulesets: List of negatively-weighted rulesets applied to the text␣
↪prompt, if supported by the driver.
        negative_rules: List of negatively-weighted rules applied to the text prompt,␣
↪if supported by the driver.
        output_dir: If provided, the generated image will be written to disk in␣
↪output_dir.
        output_file: If provided, the generated image will be written to disk as␣
↪output_file.
    """

    _input: Union[
        tuple[Union[str, TextArtifact], ImageArtifact, ImageArtifact],␣
↪Callable[[BaseTask], ListArtifact], ListArtifact
    ] = field(default=None, alias="input")

    @property
    def input(self) -> ListArtifact:
        if isinstance(self._input, ListArtifact):
            return self._input
        if isinstance(self._input, tuple):
            if isinstance(self._input[0], TextArtifact):
                input_text = self._input[0]
            else:
                input_text = TextArtifact(J2().render_from_string(self._input[0],␣
↪**self.full_context))

            return ListArtifact([input_text, self._input[1], self._input[2]])
        if isinstance(self._input, Callable):
            return self._input(self)
        raise ValueError("Input must be a tuple of (text, image, mask) or a callable␣
↪that returns such a tuple.")

    @input.setter
    def input(
        self,
        value: tuple[str | TextArtifact, ImageArtifact, ImageArtifact] |␣
↪Callable[[BaseTask], ListArtifact],
    ) -> None:
        self._input = value

    def try_run(self) -> ImageArtifact:
        prompt_artifact = self.input[0]

        image_artifact = self.input[1]
        if not isinstance(image_artifact, ImageArtifact):
            raise ValueError("Image must be an ImageArtifact.")

        mask_artifact = self.input[2]
        if not isinstance(mask_artifact, ImageArtifact):
            raise ValueError("Mask must be an ImageArtifact.")
```

(continues on next page)

(continued from previous page)

```
        output_image_artifact = self.image_generation_driver.run_image_outpainting(
            prompts=self._get_prompts(prompt_artifact.to_text()),
            negative_prompts=self._get_negative_prompts(),
            image=image_artifact,
            mask=mask_artifact,
        )

        if self.output_dir or self.output_file:
            self._write_to_file(output_image_artifact)

        return output_image_artifact
```

- `_input = field(default=None, alias='input')` class-attribute instance-attribute
- `input` property writable

try_run()

Source Code in griptape/tasks/outpainting_image_generation_task.py

```
def try_run(self) -> ImageArtifact:
    prompt_artifact = self.input[0]

    image_artifact = self.input[1]
    if not isinstance(image_artifact, ImageArtifact):
        raise ValueError("Image must be an ImageArtifact.")

    mask_artifact = self.input[2]
    if not isinstance(mask_artifact, ImageArtifact):
        raise ValueError("Mask must be an ImageArtifact.")

    output_image_artifact = self.image_generation_driver.run_image_outpainting(
        prompts=self._get_prompts(prompt_artifact.to_text()),
        negative_prompts=self._get_negative_prompts(),
        image=image_artifact,
        mask=mask_artifact,
    )

    if self.output_dir or self.output_file:
        self._write_to_file(output_image_artifact)

    return output_image_artifact
```

OutputSchemaValidationSubtask

Bases: *BaseSubtask* “)

Source Code in griptape/tasks/output_schema_validation_subtask.py

```
@define
class OutputSchemaValidationSubtask(BaseSubtask):
    _input: BaseArtifact = field(alias="input")
    output_schema: Union[Schema, type[BaseModel]] = field(kw_only=True)
```

(continues on next page)

(continued from previous page)

```
    structured_output_strategy: StructuredOutputStrategy = field(
        default="rule", kw_only=True, metadata={"serializable": True}
    )
    generate_assistant_subtask_template: Callable[[OutputSchemaValidationSubtask],␣
↪str] = field(
        default=Factory(lambda self: self.default_generate_assistant_subtask_template,
↪ takes_self=True),
        kw_only=True,
    )
    generate_user_subtask_template: Callable[[OutputSchemaValidationSubtask], str] =␣
↪field(
        default=Factory(lambda self: self.default_generate_user_subtask_template,␣
↪takes_self=True),
        kw_only=True,
    )
    _validation_errors: str | None = field(default=None, init=False)

    @property
    def input(self) -> BaseArtifact:
        return self._input

    @input.setter
    def input(self, value: BaseArtifact) -> None:
        self._input = value

    @property
    def validation_errors(self) -> str | None:
        return self._validation_errors

    def attach_to(self, parent_task: BaseTask) -> None:
        super().attach_to(parent_task)
        try:
            # With `native` or `rule` strategies, the output will be a json string␣
↪that can be parsed.
            # With the `tool` strategy, the output will already be a `JsonArtifact`.
            if self.structured_output_strategy in ("native", "rule"):
                value_to_validate = (
                    self.input.value if isinstance(self.input.value, str) else json.
↪dumps(self.input.value)
                )
                if isinstance(self.output_schema, Schema):
                    self.output_schema.validate(json.loads(value_to_validate))
                    self.output = JsonArtifact(self.input.value)
                else:
                    model = TypeAdapter(self.output_schema).validate_json(value_to_
↪validate)
                    self.output = ModelArtifact(model)
            else:
                self.output = self.input
        except SchemaError as e:
            self._validation_errors = str(e)
        except ValidationError as e:
```

(continues on next page)

(continued from previous page)

```
            self._validation_errors = str(e.errors())

    def before_run(self) -> None:
        logger.info("%s Validating: %s", self.__class__.__name__, self.input.value)

    def try_run(self) -> BaseArtifact:
        if self._validation_errors is None:
            return self._input
        return ErrorArtifact(
            value=f"Validation error: {self._validation_errors}",
        )

    def after_run(self) -> None:
        if self._validation_errors is None:
            logger.info("%s Validation successful", self.__class__.__name__)
        else:
            logger.error("%s Validation error: %s", self.__class__.__name__, self._
↪validation_errors)

    def add_to_prompt_stack(self, stack: PromptStack) -> None:
        if self.output is None:
            return
        stack.add_assistant_message(self.generate_assistant_subtask_template(self))
        stack.add_user_message(self.generate_user_subtask_template(self))

    def default_generate_assistant_subtask_template(self, subtask:␣
↪OutputSchemaValidationSubtask) -> str:
        return J2("tasks/prompt_task/assistant_output_schema_validation_subtask.j2").
↪render(
            subtask=subtask,
        )

    def default_generate_user_subtask_template(self, subtask:␣
↪OutputSchemaValidationSubtask) -> str:
        return J2("tasks/prompt_task/user_output_schema_validation_subtask.j2").
↪render(
            subtask=subtask,
        )
```

- `_input = field(alias='input')` class-attribute instance-attribute
- `_validation_errors = field(default=None, init=False)` class-attribute instance-attribute
- `generate_assistant_subtask_template = field(default=Factory(lambda self: self.default_generate_assistant_subtask_template, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `generate_user_subtask_template = field(default=Factory(lambda self: self.default_generate_user_subtask_template, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `input` property writable
- `output_schema = field(kw_only=True)` class-attribute instance-attribute
- `structured_output_strategy = field(default='rule', kw_only=True,`

`metadata={'serializable': True})` class-attribute instance-attribute

- `validation_errors` property

add_to_prompt_stack(stack)

Source Code in griptape/tasks/output_schema_validation_subtask.py

```
def add_to_prompt_stack(self, stack: PromptStack) -> None:
    if self.output is None:
        return
    stack.add_assistant_message(self.generate_assistant_subtask_template(self))
    stack.add_user_message(self.generate_user_subtask_template(self))
```

after_run()

Source Code in griptape/tasks/output_schema_validation_subtask.py

```
def after_run(self) -> None:
    if self._validation_errors is None:
        logger.info("%s Validation successful", self.__class__.__name__)
    else:
        logger.error("%s Validation error: %s", self.__class__.__name__, self._
↪validation_errors)
```

attach_to(parent_task)

Source Code in griptape/tasks/output_schema_validation_subtask.py

```
def attach_to(self, parent_task: BaseTask) -> None:
    super().attach_to(parent_task)
    try:
        # With `native` or `rule` strategies, the output will be a json string that␣
↪can be parsed.
        # With the `tool` strategy, the output will already be a `JsonArtifact`.
        if self.structured_output_strategy in ("native", "rule"):
            value_to_validate = (
                self.input.value if isinstance(self.input.value, str) else json.
↪dumps(self.input.value)
            )
            if isinstance(self.output_schema, Schema):
                self.output_schema.validate(json.loads(value_to_validate))
                self.output = JsonArtifact(self.input.value)
            else:
                model = TypeAdapter(self.output_schema).validate_json(value_to_
↪validate)
                self.output = ModelArtifact(model)
        else:
            self.output = self.input
    except SchemaError as e:
        self._validation_errors = str(e)
    except ValidationError as e:
        self._validation_errors = str(e.errors())
```

before_run()

Source Code in griptape/tasks/output_schema_validation_subtask.py

```
def before_run(self) -> None:
    logger.info("%s Validating: %s", self.__class__.__name__, self.input.value)
```

default_generate_assistant_subtask_template(subtask)

Source Code in griptape/tasks/output_schema_validation_subtask.py

```
def default_generate_assistant_subtask_template(self, subtask:␣
↪OutputSchemaValidationSubtask) -> str:
    return J2("tasks/prompt_task/assistant_output_schema_validation_subtask.j2").
↪render(
        subtask=subtask,
    )
```

default_generate_user_subtask_template(subtask)

Source Code in griptape/tasks/output_schema_validation_subtask.py

```
def default_generate_user_subtask_template(self, subtask:␣
↪OutputSchemaValidationSubtask) -> str:
    return J2("tasks/prompt_task/user_output_schema_validation_subtask.j2").render(
        subtask=subtask,
    )
```

try_run()

Source Code in griptape/tasks/output_schema_validation_subtask.py

```
def try_run(self) -> BaseArtifact:
    if self._validation_errors is None:
        return self._input
    return ErrorArtifact(
        value=f"Validation error: {self._validation_errors}",
    )
```

PromptImageGenerationTask

Bases: *BaseImageGenerationTask* “)

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">BaseImageGenerationDrive	The engine used to generate the image.
<code class="language-python">	<code class="language-python">list[Ruleset]</code>	List of negatively-weighted rulesets applied to the text prompt, if supported by the driver.
<code class="language-python">	<code class="language-python">list[Rule]</code>	List of negatively-weighted rules applied to the text prompt, if supported by the driver.

continues on next page

Table 369 – continued from previous page

Name	Type	Description
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.PromptImageGenerationTask.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk in output_dir.
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.PromptImageGenerationTask.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk as output_file.

Source Code in griptape/tasks/prompt_image_generation_task.py

```
@define
class PromptImageGenerationTask(BaseImageGenerationTask):
    """Used to generate an image from a text prompt.

    Accepts prompt as input in one of the following formats:
    - template string
    - TextArtifact
    - Callable that returns a TextArtifact.

    Attributes:
        image_generation_driver: The engine used to generate the image.
        negative_rulesets: List of negatively-weighted rulesets applied to the text␣
↪prompt, if supported by the driver.
```

(continues on next page)

(continued from previous page)

```
        negative_rules: List of negatively-weighted rules applied to the text prompt,␣
↪if supported by the driver.
        output_dir: If provided, the generated image will be written to disk in␣
↪output_dir.
        output_file: If provided, the generated image will be written to disk as␣
↪output_file.
    """

    DEFAULT_INPUT_TEMPLATE = "{{ args[0] }}"

    _input: Union[str, TextArtifact, Callable[[BaseTask], TextArtifact]] = field(
        default=DEFAULT_INPUT_TEMPLATE, alias="input"
    )

    @property
    def input(self) -> TextArtifact:
        if isinstance(self._input, TextArtifact):
            return self._input
        if isinstance(self._input, Callable):
            return self._input(self)
        return TextArtifact(J2().render_from_string(self._input, **self.full_context))

    @input.setter
    def input(self, value: TextArtifact) -> None:
        self._input = value

    def try_run(self) -> ImageArtifact:
        image_artifact = self.image_generation_driver.run_text_to_image(
            prompts=self._get_prompts(self.input.to_text()),
            negative_prompts=self._get_negative_prompts(),
        )

        if self.output_dir or self.output_file:
            self._write_to_file(image_artifact)

        return image_artifact
```

- `DEFAULT_INPUT_TEMPLATE = '{{ args[0] }}'` class-attribute instance-attribute
- `_input = field(default=DEFAULT_INPUT_TEMPLATE, alias='input')` class-attribute instance-attribute
- `input` property writable

try_run()

Source Code in griptape/tasks/prompt_image_generation_task.py

```
def try_run(self) -> ImageArtifact:
    image_artifact = self.image_generation_driver.run_text_to_image(
        prompts=self._get_prompts(self.input.to_text()),
        negative_prompts=self._get_negative_prompts(),
    )
```

(continues on next page)

(continued from previous page)

```
    if self.output_dir or self.output_file:
        self._write_to_file(image_artifact)

    return image_artifact
```

PromptTask

Bases: *BaseTask[Union[TextArtifact, AudioArtifact, GenericArtifact, JsonArtifact, ListArtifact, ErrorArtifact* “) , *RuleMixin* ”) , *ActionsSubtaskOriginMixin* “)

Source Code in griptape/tasks/prompt_task.py

```
@define
class PromptTask(
    BaseTask[Union[TextArtifact, AudioArtifact, GenericArtifact, JsonArtifact,␣
↪ListArtifact, ErrorArtifact]],
    RuleMixin,
    ActionsSubtaskOriginMixin,
):
    DEFAULT_MAX_STEPS = 20
    # Stop sequence for chain-of-thought in the framework. Using this "token-like"␣
↪string to make it more unique,
    # so that it doesnt trigger on accident.
    RESPONSE_STOP_SEQUENCE = "<|Response|>"
    prompt_driver: BasePromptDriver = field(
        default=Factory(lambda: Defaults.drivers_config.prompt_driver), kw_only=True,␣
↪metadata={"serializable": True}
    )
    output_schema: Optional[Union[Schema, type[BaseModel]]] = field(default=None, kw_
↪only=True)
    generate_system_template: Callable[[PromptTask], str] = field(
        default=Factory(lambda self: self.default_generate_system_template, takes_
↪self=True),
        kw_only=True,
    )
    _conversation_memory: Union[Optional[BaseConversationMemory], NothingType] =␣
↪field(
        default=Factory(lambda: NOTHING), kw_only=True, alias="conversation_memory"
    )
    _input: Union[str, list, tuple, BaseArtifact, Callable[[BaseTask], BaseArtifact]]␣
↪= field(
        default=lambda task: task.full_context["args"][0] if task.full_context["args
↪"] else TextArtifact(value=""),
        alias="input",
    )
    tools: list[BaseTool] = field(factory=list, kw_only=True, metadata={"serializable
↪": True})
    max_subtasks: int = field(default=DEFAULT_MAX_STEPS, kw_only=True, metadata={
↪"serializable": True})
    task_memory: Optional[TaskMemory] = field(default=None, kw_only=True)
    subtasks: list[BaseSubtask] = field(factory=list)
```

(continues on next page)

(continued from previous page)

```
    generate_assistant_subtask_template: Callable[[ActionsSubtask], str] = field(
        default=Factory(lambda self: self.default_generate_assistant_subtask_template,
↪ takes_self=True),
        kw_only=True,
    )
    generate_user_subtask_template: Callable[[ActionsSubtask], str] = field(
        default=Factory(lambda self: self.default_generate_user_subtask_template,␣
↪takes_self=True),
        kw_only=True,
    )
    response_stop_sequence: str = field(default=RESPONSE_STOP_SEQUENCE, kw_only=True)
    reflect_on_tool_use: bool = field(default=True, kw_only=True)
    subtask_runners: list[Callable[[BaseArtifact], BaseArtifact]] = field(
        default=Factory(
            lambda self: [self.default_run_actions_subtasks, self.default_run_output_
↪schema_validation_subtasks],
            takes_self=True,
        ),
        kw_only=True,
    )

    @property
    def rulesets(self) -> list:
        default_rules = self.rules
        rulesets = self._rulesets.copy()

        if self.structure is not None:
            if self.structure._rulesets:
                rulesets = self.structure._rulesets + self._rulesets
            if self.structure.rules:
                default_rules = self.structure.rules + self.rules

        if default_rules:
            rulesets.append(Ruleset(name=self.DEFAULT_RULESET_NAME, rules=default_
↪rules))

        return rulesets

    @property
    def input(self) -> BaseArtifact:
        return self._process_task_input(self._input)

    @input.setter
    def input(self, value: str | list | tuple | BaseArtifact | Callable[[BaseTask],␣
↪BaseArtifact]) -> None:
        self._input = value

    @property
    def conversation_memory(self) -> Optional[BaseConversationMemory]:
        if self._conversation_memory is NOTHING:
            if self.structure is None:
                return None
```

(continues on next page)

(continued from previous page)

```
            return self.structure.conversation_memory
        return self._conversation_memory

    @conversation_memory.setter
    def conversation_memory(self, value: Optional[BaseConversationMemory]) -> None:
        self._conversation_memory = value

    @property
    def prompt_stack(self) -> PromptStack:
        stack = PromptStack(tools=self.tools, output_schema=self.output_schema)
        memory = self.conversation_memory

        system_template = self.generate_system_template(self)
        if system_template:
            stack.add_system_message(system_template)

        stack.add_user_message(self.input)

        if self.output:
            stack.add_assistant_message(self.output.to_text())
        else:
            for s in self.subtasks:
                s.add_to_prompt_stack(stack)

        if memory is not None:
            # inserting at index 1 to place memory right after system prompt
            memory.add_to_prompt_stack(self.prompt_driver, stack, 1 if system_
↪template else 0)

        return stack

    @property
    def tool_output_memory(self) -> list[TaskMemory]:
        unique_memory_dict = {}

        for memories in [tool.output_memory for tool in self.tools if tool.output_
↪memory]:
            for memory_list in memories.values():
                for memory in memory_list:
                    if memory.name not in unique_memory_dict:
                        unique_memory_dict[memory.name] = memory

        return list(unique_memory_dict.values())

    @tools.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_tools(self, _: Attribute, tools: list[BaseTool]) -> None:
        tool_names = [t.name for t in tools]

        if len(tool_names) > len(set(tool_names)):
            raise ValueError("tools names have to be unique in task")

    @output_schema.validator  # pyright: ignore[reportAttributeAccessIssue,␣
```

(continues on next page)

(continued from previous page)

```
↪reportOptionalMemberAccess]
    def validate_output_schema(self, _: Attribute, output_schema:␣
↪Optional[Union[Schema, type[BaseModel]]]) -> None:
        if (
            output_schema is None
            or isinstance(self.output_schema, Schema)
            or (isinstance(self.output_schema, type) and issubclass(self.output_
↪schema, BaseModel))
        ):
            return
        raise ValueError(f"Unsupported output schema type: {type(self.output_schema)}
↪")

    def __attrs_post_init__(self) -> None:
        super().__attrs_post_init__()
        if self.task_memory:
            self.set_default_tools_memory(self.task_memory)

    output: Optional[BaseArtifact] = field(default=None, init=False)

    def before_run(self) -> None:
        super().before_run()

        logger.info("%s %s\nInput: %s", self.__class__.__name__, self.id, self.input.
↪to_text())

    def after_run(self) -> None:
        super().after_run()

        logger.info(
            "%s %s\nOutput: %s",
            self.__class__.__name__,
            self.id,
            self.output.to_text() if self.output is not None else "",
        )
        conversation_memory = self.conversation_memory
        if (
            (self.structure is None or self.structure.conversation_memory_strategy ==
↪"per_task")
            and conversation_memory is not None
            and self.output is not None
        ):
            run = Run(input=self.input, output=self.output)

            conversation_memory.add_run(run)

    def try_run(self) -> ListArtifact | TextArtifact | AudioArtifact |␣
↪GenericArtifact | JsonArtifact | ErrorArtifact:
        self.subtasks.clear()
        if self.response_stop_sequence not in self.prompt_driver.tokenizer.stop_
↪sequences:
            self.prompt_driver.tokenizer.stop_sequences.extend([self.response_stop_
```

(continues on next page)

(continued from previous page)

```
↪sequence])

        output = self.prompt_driver.run(self.prompt_stack).to_artifact(
            meta={"is_react_prompt": not self.prompt_driver.use_native_tools}
        )
        for subtask_runner in self.subtask_runners:
            output = subtask_runner(output)

        if isinstance(output, (ListArtifact, TextArtifact, AudioArtifact,␣
↪JsonArtifact, ModelArtifact, ErrorArtifact)):
            return output
        raise ValueError(f"Unsupported output type: {type(output)}")

    def preprocess(self, structure: Structure) -> BaseTask:
        super().preprocess(structure)

        if self.task_memory is None and structure.task_memory:
            self.set_default_tools_memory(structure.task_memory)

        return self

    def default_generate_system_template(self, _: PromptTask) -> str:
        schema = self.actions_schema().json_schema("Actions Schema")
        schema["minItems"] = 1  # The `schema` library doesnt support `minItems` so␣
↪we must add it manually.

        return J2("tasks/prompt_task/system.j2").render(
            rulesets=J2("rulesets/rulesets.j2").render(rulesets=self.rulesets),
            action_names=str.join(", ", [tool.name for tool in self.tools]),
            actions_schema=utils.minify_json(json.dumps(schema)),
            meta_memory=J2("memory/meta/meta_memory.j2").render(meta_memories=self.
↪meta_memories),
            use_native_tools=self.prompt_driver.use_native_tools,
            stop_sequence=self.response_stop_sequence,
            reflect_on_tool_use=self.reflect_on_tool_use,
        )

    def default_generate_assistant_subtask_template(self, subtask: ActionsSubtask) ->␣
↪str:
        return J2("tasks/prompt_task/assistant_actions_subtask.j2").render(
            stop_sequence=self.response_stop_sequence,
            subtask=subtask,
        )

    def default_generate_user_subtask_template(self, subtask: ActionsSubtask) -> str:
        return J2("tasks/prompt_task/user_actions_subtask.j2").render(
            stop_sequence=self.response_stop_sequence,
            subtask=subtask,
        )

    def actions_schema(self) -> Schema:
        return self._actions_schema_for_tools(self.tools)
```

(continues on next page)

(continued from previous page)

```
    def set_default_tools_memory(self, memory: TaskMemory) -> None:
        self.task_memory = memory

        for tool in self.tools:
            if self.task_memory:
                if tool.input_memory is None:
                    tool.input_memory = [self.task_memory]
                if tool.output_memory is None and tool.off_prompt:
                    tool.output_memory = {getattr(a, "name"): [self.task_memory] for␣
↪a in tool.activities()}

    def find_subtask(self, subtask_id: str) -> BaseSubtask:
        for subtask in self.subtasks:
            if subtask.id == subtask_id:
                return subtask
        raise ValueError(f"Subtask with id {subtask_id} not found.")

    def add_subtask(self, subtask: BaseSubtask) -> BaseSubtask:
        subtask.attach_to(self)
        subtask.structure = self.structure

        if len(self.subtasks) > 0:
            self.subtasks[-1].add_child(subtask)
            subtask.add_parent(self.subtasks[-1])

        self.subtasks.append(subtask)

        return subtask

    def find_tool(self, tool_name: str) -> BaseTool:
        for tool in self.tools:
            if tool.name == tool_name:
                return tool
        raise ValueError(f"Tool with name {tool_name} not found.")

    def find_memory(self, memory_name: str) -> TaskMemory:
        for memory in self.tool_output_memory:
            if memory.name == memory_name:
                return memory
        raise ValueError(f"Memory with name {memory_name} not found.")

    def default_run_actions_subtasks(self, subtask_input: BaseArtifact) ->␣
↪BaseArtifact:
        if not self.tools:
            return subtask_input
        subtask = self.add_subtask(
            ActionsSubtask(
                subtask_input,
                # TODO: Remove these fields in Prompt Task in Gen AI Builder 2.0
                generate_user_subtask_template=self.generate_user_subtask_template,
                generate_assistant_subtask_template=self.generate_assistant_subtask_
```

(continues on next page)

(continued from previous page)

```
↪template,
                response_stop_sequence=self.response_stop_sequence,
            )
        )

        while subtask.output is None:
            if len(self.subtasks) >= self.max_subtasks:
                subtask.output = ErrorArtifact(f"Exceeded tool limit of {self.max_
↪subtasks} subtasks per task")
            else:
                subtask.run()

                if self.reflect_on_tool_use:
                    output = self.prompt_driver.run(self.prompt_stack).to_artifact(
                        meta={"is_react_prompt": not self.prompt_driver.use_native_
↪tools}
                    )
                    subtask = self.add_subtask(ActionsSubtask(output))

        return subtask.output

    def default_run_output_schema_validation_subtasks(self, subtask_input:␣
↪BaseArtifact) -> BaseArtifact:
        if self.output_schema is None:
            return subtask_input
        subtask = self.add_subtask(OutputSchemaValidationSubtask(subtask_input,␣
↪output_schema=self.output_schema))

        while subtask.output is None:
            if len(self.subtasks) >= self.max_subtasks:
                subtask.output = ErrorArtifact(f"Exceeded tool limit of {self.max_
↪subtasks} subtasks per task")
            else:
                subtask.run()

                output = subtask.output
                output = self.prompt_driver.run(self.prompt_stack).to_artifact(
                    meta={"is_react_prompt": not self.prompt_driver.use_native_tools}
                )
                subtask = self.add_subtask(OutputSchemaValidationSubtask(output,␣
↪output_schema=self.output_schema))

        return subtask.output

    def _process_task_input(
        self,
        task_input: str | tuple | list | BaseArtifact | Callable[[BaseTask],␣
↪BaseArtifact],
    ) -> BaseArtifact:
        if isinstance(task_input, TextArtifact):
            return TextArtifact(J2().render_from_string(task_input.value, **self.full_
↪context), meta=task_input.meta)
```

(continues on next page)

(continued from previous page)

```
        if isinstance(task_input, Callable):
            return self._process_task_input(task_input(self))
        if isinstance(task_input, ListArtifact):
            return ListArtifact([self._process_task_input(elem) for elem in task_
↪input.value])
        if isinstance(task_input, BaseArtifact):
            return task_input
        if isinstance(task_input, (list, tuple)):
            return ListArtifact([self._process_task_input(elem) for elem in task_
↪input])
        return self._process_task_input(TextArtifact(task_input))
```

- `DEFAULT_MAX_STEPS = 20` class-attribute instance-attribute
- `RESPONSE_STOP_SEQUENCE = '<|Response|>'` class-attribute instance-attribute
- `_conversation_memory = field(default=Factory(lambda: NOTHING), kw_only=True, alias='conversation_memory')` class-attribute instance-attribute
- `_input = field(default=lambda task: task.full_context['args'][0] if task.full_context['args'] else TextArtifact(value=''), alias='input')` class-attribute instance-attribute
- `conversation_memory` property writable
- `generate_assistant_subtask_template = field(default=Factory(lambda self: self.default_generate_assistant_subtask_template, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `generate_system_template = field(default=Factory(lambda self: self.default_generate_system_template, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `generate_user_subtask_template = field(default=Factory(lambda self: self.default_generate_user_subtask_template, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `input` property writable
- `max_subtasks = field(default=DEFAULT_MAX_STEPS, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output = field(default=None, init=False)` class-attribute instance-attribute
- `output_schema = field(default=None, kw_only=True)` class-attribute instance-attribute
- `prompt_driver = field(default=Factory(lambda: Defaults.drivers_config.prompt_driver), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `prompt_stack` property
- `reflect_on_tool_use = field(default=True, kw_only=True)` class-attribute instance-attribute
- `response_stop_sequence = field(default=RESPONSE_STOP_SEQUENCE, kw_only=True)` class-attribute instance-attribute
- `rulesets` property

- `subtask_runners = field(default=Factory(lambda self: [self.default_run_actions_subtasks, self.default_run_output_schema_validation_subtasks], takes_self=True), kw_only=True)` class-attribute instance-attribute
- `subtasks = field(factory=list)` class-attribute instance-attribute
- `task_memory = field(default=None, kw_only=True)` class-attribute instance-attribute
- `tool_output_memory` property
- `tools = field(factory=list, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/tasks/prompt_task.py

```
def __attrs_post_init__(self) -> None:
    super().__attrs_post_init__()
    if self.task_memory:
        self.set_default_tools_memory(self.task_memory)
```

_process_task_input(task_input)

Source Code in griptape/tasks/prompt_task.py

```
def _process_task_input(
    self,
    task_input: str | tuple | list | BaseArtifact | Callable[[BaseTask],␣
↪BaseArtifact],
) -> BaseArtifact:
    if isinstance(task_input, TextArtifact):
        return TextArtifact(J2().render_from_string(task_input.value, **self.full_
↪context), meta=task_input.meta)
    if isinstance(task_input, Callable):
        return self._process_task_input(task_input(self))
    if isinstance(task_input, ListArtifact):
        return ListArtifact([self._process_task_input(elem) for elem in task_input.
↪value])
    if isinstance(task_input, BaseArtifact):
        return task_input
    if isinstance(task_input, (list, tuple)):
        return ListArtifact([self._process_task_input(elem) for elem in task_input])
    return self._process_task_input(TextArtifact(task_input))
```

actions_schema()

Source Code in griptape/tasks/prompt_task.py

```
def actions_schema(self) -> Schema:
    return self._actions_schema_for_tools(self.tools)
```

add_subtask(subtask)

Source Code in griptape/tasks/prompt_task.py

```
def add_subtask(self, subtask: BaseSubtask) -> BaseSubtask:
    subtask.attach_to(self)
    subtask.structure = self.structure

    if len(self.subtasks) > 0:
        self.subtasks[-1].add_child(subtask)
        subtask.add_parent(self.subtasks[-1])

    self.subtasks.append(subtask)

    return subtask
```

after_run()

Source Code in griptape/tasks/prompt_task.py

```
def after_run(self) -> None:
    super().after_run()

    logger.info(
        "%s %s\nOutput: %s",
        self.__class__.__name__,
        self.id,
        self.output.to_text() if self.output is not None else "",
    )
    conversation_memory = self.conversation_memory
    if (
        (self.structure is None or self.structure.conversation_memory_strategy ==
↪"per_task")
        and conversation_memory is not None
        and self.output is not None
    ):
        run = Run(input=self.input, output=self.output)

        conversation_memory.add_run(run)
```

before_run()

Source Code in griptape/tasks/prompt_task.py

```
def before_run(self) -> None:
    super().before_run()

    logger.info("%s %s\nInput: %s", self.__class__.__name__, self.id, self.input.to_
↪text())
```

default_generate_assistant_subtask_template(subtask)

Source Code in griptape/tasks/prompt_task.py

```
def default_generate_assistant_subtask_template(self, subtask: ActionsSubtask) -> str:
    return J2("tasks/prompt_task/assistant_actions_subtask.j2").render(
        stop_sequence=self.response_stop_sequence,
        subtask=subtask,
    )
```

default*generate_system_template()*

Source Code in griptape/tasks/prompt_task.py

```
def default_generate_system_template(self, _: PromptTask) -> str:
    schema = self.actions_schema().json_schema("Actions Schema")
    schema["minItems"] = 1  # The `schema` library doesnt support `minItems` so we␣
↪must add it manually.

    return J2("tasks/prompt_task/system.j2").render(
        rulesets=J2("rulesets/rulesets.j2").render(rulesets=self.rulesets),
        action_names=str.join(", ", [tool.name for tool in self.tools]),
        actions_schema=utils.minify_json(json.dumps(schema)),
        meta_memory=J2("memory/meta/meta_memory.j2").render(meta_memories=self.meta_
↪memories),
        use_native_tools=self.prompt_driver.use_native_tools,
        stop_sequence=self.response_stop_sequence,
        reflect_on_tool_use=self.reflect_on_tool_use,
    )
```

default_generate_user_subtask_template(subtask)

Source Code in griptape/tasks/prompt_task.py

```
def default_generate_user_subtask_template(self, subtask: ActionsSubtask) -> str:
    return J2("tasks/prompt_task/user_actions_subtask.j2").render(
        stop_sequence=self.response_stop_sequence,
        subtask=subtask,
    )
```

default_run_actions_subtasks(subtask_input)

Source Code in griptape/tasks/prompt_task.py

```
def default_run_actions_subtasks(self, subtask_input: BaseArtifact) -> BaseArtifact:
    if not self.tools:
        return subtask_input
    subtask = self.add_subtask(
        ActionsSubtask(
            subtask_input,
            # TODO: Remove these fields in Prompt Task in Gen AI Builder 2.0
            generate_user_subtask_template=self.generate_user_subtask_template,
            generate_assistant_subtask_template=self.generate_assistant_subtask_
↪template,
```

(continues on next page)

(continued from previous page)

```
            response_stop_sequence=self.response_stop_sequence,
        )
    )

    while subtask.output is None:
        if len(self.subtasks) >= self.max_subtasks:
            subtask.output = ErrorArtifact(f"Exceeded tool limit of {self.max_
↪subtasks} subtasks per task")
        else:
            subtask.run()

            if self.reflect_on_tool_use:
                output = self.prompt_driver.run(self.prompt_stack).to_artifact(
                    meta={"is_react_prompt": not self.prompt_driver.use_native_tools}
                )
                subtask = self.add_subtask(ActionsSubtask(output))

    return subtask.output
```

default_run_output_schema_validation_subtasks(subtask_input)

Source Code in griptape/tasks/prompt_task.py

```
def default_run_output_schema_validation_subtasks(self, subtask_input: BaseArtifact) -
↪> BaseArtifact:
    if self.output_schema is None:
        return subtask_input
    subtask = self.add_subtask(OutputSchemaValidationSubtask(subtask_input, output_
↪schema=self.output_schema))

    while subtask.output is None:
        if len(self.subtasks) >= self.max_subtasks:
            subtask.output = ErrorArtifact(f"Exceeded tool limit of {self.max_
↪subtasks} subtasks per task")
        else:
            subtask.run()

            output = subtask.output
            output = self.prompt_driver.run(self.prompt_stack).to_artifact(
                meta={"is_react_prompt": not self.prompt_driver.use_native_tools}
            )
            subtask = self.add_subtask(OutputSchemaValidationSubtask(output, output_
↪schema=self.output_schema))

    return subtask.output
```

find_memory(memory_name)

Source Code in griptape/tasks/prompt_task.py

```
def find_memory(self, memory_name: str) -> TaskMemory:
    for memory in self.tool_output_memory:
```

(continues on next page)

(continued from previous page)

```
        if memory.name == memory_name:
            return memory
    raise ValueError(f"Memory with name {memory_name} not found.")
```

find_subtask(subtask_id)

Source Code in griptape/tasks/prompt_task.py

```
def find_subtask(self, subtask_id: str) -> BaseSubtask:
    for subtask in self.subtasks:
        if subtask.id == subtask_id:
            return subtask
    raise ValueError(f"Subtask with id {subtask_id} not found.")
```

find_tool(tool_name)

Source Code in griptape/tasks/prompt_task.py

```
def find_tool(self, tool_name: str) -> BaseTool:
    for tool in self.tools:
        if tool.name == tool_name:
            return tool
    raise ValueError(f"Tool with name {tool_name} not found.")
```

preprocess(structure)

Source Code in griptape/tasks/prompt_task.py

```
def preprocess(self, structure: Structure) -> BaseTask:
    super().preprocess(structure)

    if self.task_memory is None and structure.task_memory:
        self.set_default_tools_memory(structure.task_memory)

    return self
```

set_default_tools_memory(memory)

Source Code in griptape/tasks/prompt_task.py

```
def set_default_tools_memory(self, memory: TaskMemory) -> None:
    self.task_memory = memory

    for tool in self.tools:
        if self.task_memory:
            if tool.input_memory is None:
                tool.input_memory = [self.task_memory]
            if tool.output_memory is None and tool.off_prompt:
                tool.output_memory = {getattr(a, "name"): [self.task_memory] for a in␣
↪tool.activities()}
```

try_run()

Source Code in griptape/tasks/prompt_task.py

```
def try_run(self) -> ListArtifact | TextArtifact | AudioArtifact | GenericArtifact |␣
↪JsonArtifact | ErrorArtifact:
    self.subtasks.clear()
    if self.response_stop_sequence not in self.prompt_driver.tokenizer.stop_sequences:
        self.prompt_driver.tokenizer.stop_sequences.extend([self.response_stop_
↪sequence])

    output = self.prompt_driver.run(self.prompt_stack).to_artifact(
        meta={"is_react_prompt": not self.prompt_driver.use_native_tools}
    )
    for subtask_runner in self.subtask_runners:
        output = subtask_runner(output)

    if isinstance(output, (ListArtifact, TextArtifact, AudioArtifact, JsonArtifact,␣
↪ModelArtifact, ErrorArtifact)):
        return output
    raise ValueError(f"Unsupported output type: {type(output)}")
```

validate*output_schema(*, output_schema)

Source Code in griptape/tasks/prompt_task.py

```
@output_schema.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_output_schema(self, _: Attribute, output_schema: Optional[Union[Schema,␣
↪type[BaseModel]]]) -> None:
    if (
        output_schema is None
        or isinstance(self.output_schema, Schema)
        or (isinstance(self.output_schema, type) and issubclass(self.output_schema,␣
↪BaseModel))
    ):
        return
    raise ValueError(f"Unsupported output schema type: {type(self.output_schema)}")
```

validate*tools(*, tools)

Source Code in griptape/tasks/prompt_task.py

```
@tools.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_tools(self, _: Attribute, tools: list[BaseTool]) -> None:
    tool_names = [t.name for t in tools]

    if len(tool_names) > len(set(tool_names)):
        raise ValueError("tools names have to be unique in task")
```

RagTask

Bases: *BaseTextInputTask[Union[ListArtifact, ErrorArtifact* “)

Source Code in griptape/tasks/rag_task.py

```
@define
class RagTask(BaseTextInputTask[Union[ListArtifact, ErrorArtifact]]):
    rag_engine: RagEngine = field(kw_only=True, default=Factory(lambda: RagEngine()))

    def try_run(self) -> ListArtifact | ErrorArtifact:
        outputs = self.rag_engine.process_query(self.input.to_text()).outputs

        if len(outputs) > 0:
            return ListArtifact(outputs)
        return ErrorArtifact("empty output")
```

- `rag_engine = field(kw_only=True, default=Factory(lambda: RagEngine()))` class-attribute instance-attribute

try_run()

Source Code in griptape/tasks/rag_task.py

```
def try_run(self) -> ListArtifact | ErrorArtifact:
    outputs = self.rag_engine.process_query(self.input.to_text()).outputs

    if len(outputs) > 0:
        return ListArtifact(outputs)
    return ErrorArtifact("empty output")
```

StructureRunTask

Bases: *BaseTaskEvent* “)

Attributes

Name	Type	Description
<a href="./#griptape.tasks.StructureRunTask.structure_run_driver" title="structure_run_driver = field(kw_only=True) class-attribute instance-attribute (griptape.tasks.StructureRunTask.structure_run_driver)"><code class="language-python">	<a href="../drivers/structure_run/base_structure_run_driver.mdx#griptape.drivers.structure_run.base_structure_run_driver.BaseStructureRunDriver" title="BaseStructureRunDriver (griptape.drivers.structure_run.base_structure_run_driver.BaseStructureRunDriver)"><code class="language-python">BaseStructureRunDriver</c	Driver to run the Structure.

Source Code in griptape/tasks/structure_run_task.py

```
@define
class StructureRunTask(BaseTask):
    """Task to run a Structure.

    Attributes:
        structure_run_driver: Driver to run the Structure.
    """

    _input: Union[str, list, tuple, BaseArtifact, Callable[[BaseTask], BaseArtifact]]␣
↪= field(
        default=lambda task: task.full_context["args"][0] if task.full_context["args
↪"] else TextArtifact(value=""),
    )

    @property
    def input(self) -> BaseArtifact:
        return self._process_task_input(self._input)

    @input.setter
    def input(self, value: str | list | tuple | BaseArtifact | Callable[[BaseTask],␣
↪BaseArtifact]) -> None:
        self._input = value

    structure_run_driver: BaseStructureRunDriver = field(kw_only=True)

    def try_run(self) -> BaseArtifact:
        if isinstance(self.input, ListArtifact):
            return self.structure_run_driver.run(*self.input.value)
        return self.structure_run_driver.run(self.input)

    def _process_task_input(
        self,
        task_input: str | tuple | list | BaseArtifact | Callable[[BaseTask],␣
↪BaseArtifact],
    ) -> BaseArtifact:
        if isinstance(task_input, TextArtifact):
            task_input.value = J2().render_from_string(task_input.value, **self.full_
↪context)

            return task_input
        if isinstance(task_input, Callable):
            return self._process_task_input(task_input(self))
        if isinstance(task_input, ListArtifact):
            return ListArtifact([self._process_task_input(elem) for elem in task_
↪input.value])
        if isinstance(task_input, BaseArtifact):
            return task_input
        if isinstance(task_input, (list, tuple)):
            return ListArtifact([self._process_task_input(elem) for elem in task_
↪input])
        return self._process_task_input(TextArtifact(task_input))
```

- `_input = field(default=lambda task: task.full_context['args'][0] if task.`

`full_context['args'] else TextArtifact(value=''))` class-attribute instance-attribute

- `input` property writable
- `structure_run_driver = field(kw_only=True)` class-attribute instance-attribute

_process_task_input(task_input)

Source Code in griptape/tasks/structure_run_task.py

```
def _process_task_input(
    self,
    task_input: str | tuple | list | BaseArtifact | Callable[[BaseTask],␣
↪BaseArtifact],
) -> BaseArtifact:
    if isinstance(task_input, TextArtifact):
        task_input.value = J2().render_from_string(task_input.value, **self.full_
↪context)

        return task_input
    if isinstance(task_input, Callable):
        return self._process_task_input(task_input(self))
    if isinstance(task_input, ListArtifact):
        return ListArtifact([self._process_task_input(elem) for elem in task_input.
↪value])
    if isinstance(task_input, BaseArtifact):
        return task_input
    if isinstance(task_input, (list, tuple)):
        return ListArtifact([self._process_task_input(elem) for elem in task_input])
    return self._process_task_input(TextArtifact(task_input))
```

try_run()

Source Code in griptape/tasks/structure_run_task.py

```
def try_run(self) -> BaseArtifact:
    if isinstance(self.input, ListArtifact):
        return self.structure_run_driver.run(*self.input.value)
    return self.structure_run_driver.run(self.input)
```

TextSummaryTask

Bases: *BaseTextInputTask[TextArtifact* “)

Source Code in griptape/tasks/text_summary_task.py

```
@define
class TextSummaryTask(BaseTextInputTask[TextArtifact]):
    summary_engine: BaseSummaryEngine = field(default=Factory(lambda:␣
↪PromptSummaryEngine()), kw_only=True)

    def try_run(self) -> TextArtifact:
        return TextArtifact(self.summary_engine.summarize_text(self.input.to_text(),␣
↪rulesets=self.rulesets))
```

- `summary_engine = field(default=Factory(lambda: PromptSummaryEngine()), kw_only=True)` class-attribute instance-attribute

try_run()

Source Code in griptape/tasks/text_summary_task.py

```
def try_run(self) -> TextArtifact:
    return TextArtifact(self.summary_engine.summarize_text(self.input.to_text(),␣
↪rulesets=self.rulesets))
```

TextToSpeechTask

Bases: *BaseAudioGenerationTask* “)

Source Code in griptape/tasks/text_to_speech_task.py

```
@define
class TextToSpeechTask(BaseAudioGenerationTask):
    DEFAULT_INPUT_TEMPLATE = "{{ args[0] }}"

    _input: Union[str, TextArtifact, Callable[[BaseTask], TextArtifact]] =␣
↪field(default=DEFAULT_INPUT_TEMPLATE)
    text_to_speech_driver: BaseTextToSpeechDriver = field(
        default=Factory(lambda: Defaults.drivers_config.text_to_speech_driver), kw_
↪only=True
    )

    @property
    def input(self) -> TextArtifact:
        if isinstance(self._input, TextArtifact):
            return self._input
        if isinstance(self._input, Callable):
            return self._input(self)
        return TextArtifact(J2().render_from_string(self._input, **self.full_context))

    @input.setter
    def input(self, value: TextArtifact) -> None:
        self._input = value

    def try_run(self) -> AudioArtifact:
        audio_artifact = self.text_to_speech_driver.run_text_to_audio(prompts=[self.
↪input.to_text()])

        if self.output_dir or self.output_file:
            self._write_to_file(audio_artifact)

        return audio_artifact
```

- `DEFAULT_INPUT_TEMPLATE = '{{ args[0] }}'` class-attribute instance-attribute
- `_input = field(default=DEFAULT_INPUT_TEMPLATE)` class-attribute instance-attribute
- `input` property writable

- `text_to_speech_driver = field(default=Factory(lambda: Defaults.drivers_config.text_to_speech_driver), kw_only=True)` class-attribute instance-attribute

try_run()

Source Code in griptape/tasks/text_to_speech_task.py

```
def try_run(self) -> AudioArtifact:
    audio_artifact = self.text_to_speech_driver.run_text_to_audio(prompts=[self.input.
↪to_text()])

    if self.output_dir or self.output_file:
        self._write_to_file(audio_artifact)

    return audio_artifact
```

ToolTask

Bases: *PromptTask* “) , *ActionsSubtaskOriginMixin* ”)

Source Code in griptape/tasks/tool_task.py

```
@define
class ToolTask(PromptTask, ActionsSubtaskOriginMixin):
    DEFAULT_MAX_STEPS = 0
    ACTION_PATTERN = r"(?s)[^{]*({.*})"

    tool: BaseTool = field(kw_only=True, metadata={"serializable": True})
    subtask: Optional[ActionsSubtask] = field(default=None, kw_only=True)
    task_memory: Optional[TaskMemory] = field(default=None, kw_only=True)
    tools: list[BaseTool] = field(factory=list, kw_only=True, metadata={"serializable
↪": False})
    max_subtasks: int = field(default=DEFAULT_MAX_STEPS, kw_only=True, metadata={
↪"serializable": False})

    @property
    def prompt_stack(self) -> PromptStack:
        stack = super().prompt_stack
        stack.tools = [self.tool]

        return stack

    def __attrs_post_init__(self) -> None:
        super().__attrs_post_init__()
        if self.task_memory is not None:
            self.set_default_tools_memory(self.task_memory)

    def preprocess(self, structure: Structure) -> ToolTask:
        super().preprocess(structure)

        if self.task_memory is None and structure.task_memory is not None:
            self.set_default_tools_memory(structure.task_memory)

        return self
```

(continues on next page)

(continued from previous page)

```
    def default_generate_system_template(self, _: PromptTask) -> str:
        return J2("tasks/tool_task/system.j2").render(
            rulesets=J2("rulesets/rulesets.j2").render(rulesets=self.rulesets),
            action_schema=utils.minify_json(json.dumps(self.tool.schema())),
            meta_memory=J2("memory/meta/meta_memory.j2").render(meta_memories=self.
↪meta_memories),
            use_native_tools=self.prompt_driver.use_native_tools,
        )

    def actions_schema(self) -> Schema:
        return self._actions_schema_for_tools([self.tool])

    def try_run(self) -> ListArtifact | TextArtifact | ErrorArtifact:
        warnings.warn(
            "`ToolTask` is deprecated and will be removed in a future release. Use␣
↪`PromptTask` with `reflect_on_tool_use=False` instead.",
            DeprecationWarning,
            stacklevel=2,
        )
        result = self.prompt_driver.run(self.prompt_stack)

        if self.prompt_driver.use_native_tools:
            subtask_input = result.to_artifact(meta={"is_react_prompt": False})
        else:
            action_matches = re.findall(self.ACTION_PATTERN, result.to_text(), re.
↪DOTALL)

            if not action_matches:
                return ErrorArtifact("No action found in prompt output.")
            data = action_matches[-1]
            action_dict = json.loads(data)

            action_dict["tag"] = self.tool.name
            subtask_input = TextArtifact(
                J2("tasks/tool_task/subtask.j2").render(action_json=json.dumps(action_
↪dict)),
                meta={"is_react_prompt": True},
            )

        try:
            subtask = self.add_subtask(ActionsSubtask(subtask_input))

            output = subtask.run()

            if isinstance(output, ListArtifact):
                output = output[0]
        except Exception as e:
            output = ErrorArtifact(f"Error processing tool input: {e}", exception=e)
        return output

    def find_tool(self, tool_name: str) -> BaseTool:
```

(continues on next page)

(continued from previous page)

```
        if self.tool.name == tool_name:
            return self.tool
        raise ValueError(f"Tool with name {tool_name} not found.")

    def find_memory(self, memory_name: str) -> TaskMemory:
        raise NotImplementedError("ToolTask does not support Task Memory.")

    def find_subtask(self, subtask_id: str) -> ActionsSubtask:
        if self.subtask and self.subtask.id == subtask_id:
            return self.subtask
        raise ValueError(f"Subtask with id {subtask_id} not found.")

    def add_subtask(self, subtask: BaseSubtask) -> ActionsSubtask:
        if not isinstance(subtask, ActionsSubtask):
            raise TypeError("Subtask must be an instance of ActionsSubtask.")

        self.subtask = subtask
        self.subtask.attach_to(self)

        return self.subtask

    def set_default_tools_memory(self, memory: TaskMemory) -> None:
        self.task_memory = memory

        if self.task_memory:
            if self.tool.input_memory is None:
                self.tool.input_memory = [self.task_memory]
            if self.tool.output_memory is None and self.tool.off_prompt:
                self.tool.output_memory = {getattr(a, "name"): [self.task_memory] for␣
↪a in self.tool.activities()}
```

- `ACTION_PATTERN = '(?s)[^{]*({.*})'` class-attribute instance-attribute
- `DEFAULT_MAX_STEPS = 0` class-attribute instance-attribute
- `max_subtasks = field(default=DEFAULT_MAX_STEPS, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `prompt_stack` property
- `subtask = field(default=None, kw_only=True)` class-attribute instance-attribute
- `task_memory = field(default=None, kw_only=True)` class-attribute instance-attribute
- `tool = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tools = field(factory=list, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/tasks/tool_task.py

```
def __attrs_post_init__(self) -> None:
    super().__attrs_post_init__()
```

(continues on next page)

(continued from previous page)

```
    if self.task_memory is not None:
        self.set_default_tools_memory(self.task_memory)
```

actions_schema()

Source Code in griptape/tasks/tool_task.py

```
def actions_schema(self) -> Schema:
    return self._actions_schema_for_tools([self.tool])
```

add_subtask(subtask)

Source Code in griptape/tasks/tool_task.py

```
def add_subtask(self, subtask: BaseSubtask) -> ActionsSubtask:
    if not isinstance(subtask, ActionsSubtask):
        raise TypeError("Subtask must be an instance of ActionsSubtask.")

    self.subtask = subtask
    self.subtask.attach_to(self)

    return self.subtask
```

default*generate_system_template*()

Source Code in griptape/tasks/tool_task.py

```
def default_generate_system_template(self, _: PromptTask) -> str:
    return J2("tasks/tool_task/system.j2").render(
        rulesets=J2("rulesets/rulesets.j2").render(rulesets=self.rulesets),
        action_schema=utils.minify_json(json.dumps(self.tool.schema())),
        meta_memory=J2("memory/meta/meta_memory.j2").render(meta_memories=self.meta_
↪memories),
        use_native_tools=self.prompt_driver.use_native_tools,
    )
```

find_memory(memory_name)

Source Code in griptape/tasks/tool_task.py

```
def find_memory(self, memory_name: str) -> TaskMemory:
    raise NotImplementedError("ToolTask does not support Task Memory.")
```

find_subtask(subtask_id)

Source Code in griptape/tasks/tool_task.py

```
def find_subtask(self, subtask_id: str) -> ActionsSubtask:
    if self.subtask and self.subtask.id == subtask_id:
        return self.subtask
    raise ValueError(f"Subtask with id {subtask_id} not found.")
```

find_tool(tool_name)

Source Code in griptape/tasks/tool_task.py

```
def find_tool(self, tool_name: str) -> BaseTool:
    if self.tool.name == tool_name:
        return self.tool
    raise ValueError(f"Tool with name {tool_name} not found.")
```

preprocess(structure)

Source Code in griptape/tasks/tool_task.py

```
def preprocess(self, structure: Structure) -> ToolTask:
    super().preprocess(structure)

    if self.task_memory is None and structure.task_memory is not None:
        self.set_default_tools_memory(structure.task_memory)

    return self
```

set_default_tools_memory(memory)

Source Code in griptape/tasks/tool_task.py

```
def set_default_tools_memory(self, memory: TaskMemory) -> None:
    self.task_memory = memory

    if self.task_memory:
        if self.tool.input_memory is None:
            self.tool.input_memory = [self.task_memory]
        if self.tool.output_memory is None and self.tool.off_prompt:
            self.tool.output_memory = {getattr(a, "name"): [self.task_memory] for a␣
↪in self.tool.activities()}
```

try_run()

Source Code in griptape/tasks/tool_task.py

```
def try_run(self) -> ListArtifact | TextArtifact | ErrorArtifact:
    warnings.warn(
        "`ToolTask` is deprecated and will be removed in a future release. Use␣
↪`PromptTask` with `reflect_on_tool_use=False` instead.",
        DeprecationWarning,
        stacklevel=2,
    )
    result = self.prompt_driver.run(self.prompt_stack)

    if self.prompt_driver.use_native_tools:
        subtask_input = result.to_artifact(meta={"is_react_prompt": False})
    else:
        action_matches = re.findall(self.ACTION_PATTERN, result.to_text(), re.DOTALL)

        if not action_matches:
```

(continues on next page)

(continued from previous page)

```
            return ErrorArtifact("No action found in prompt output.")
        data = action_matches[-1]
        action_dict = json.loads(data)

        action_dict["tag"] = self.tool.name
        subtask_input = TextArtifact(
            J2("tasks/tool_task/subtask.j2").render(action_json=json.dumps(action_
↪dict)),
            meta={"is_react_prompt": True},
        )

    try:
        subtask = self.add_subtask(ActionsSubtask(subtask_input))

        output = subtask.run()

        if isinstance(output, ListArtifact):
            output = output[0]
    except Exception as e:
        output = ErrorArtifact(f"Error processing tool input: {e}", exception=e)
    return output
```

ToolkitTask

Bases: *PromptTask* “)

Source Code in griptape/tasks/toolkit_task.py

```
@define
class ToolkitTask(PromptTask):
    def try_run(self) -> ListArtifact | TextArtifact | AudioArtifact |␣
↪GenericArtifact | JsonArtifact | ErrorArtifact:
        warnings.warn(
            "`ToolkitTask` is deprecated and will be removed in a future release.␣
↪`PromptTask` is a drop-in replacement.",
            DeprecationWarning,
            stacklevel=2,
        )

        return super().try_run()
```

try_run()

Source Code in griptape/tasks/toolkit_task.py

```
def try_run(self) -> ListArtifact | TextArtifact | AudioArtifact | GenericArtifact |␣
↪JsonArtifact | ErrorArtifact:
    warnings.warn(
        "`ToolkitTask` is deprecated and will be removed in a future release.␣
↪`PromptTask` is a drop-in replacement.",
        DeprecationWarning,
        stacklevel=2,
```

(continues on next page)

(continued from previous page)

```
    )

    return super().try_run()
```

VariationImageGenerationTask

Bases: *BaseImageGenerationTask* “)

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">BaseImageGenerationDrive	The engine used to generate the image.
<code class="language-python">	<code class="language-python">list[Ruleset]</code>	List of negatively-weighted rulesets applied to the text prompt, if supported by the driver.

continues on next page

Table 371 – continued from previous page

Name	Type	Description
<code class="language-python">	<code class="language-python">list[Rule]</code>	List of negatively-weighted rules applied to the text prompt, if supported by the driver.
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.VariationImageGenerationTask.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk in output_dir.
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.VariationImageGenerationTask.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk as output_file.

Source Code in griptape/tasks/variation_image_generation_task.py

```
@define
class VariationImageGenerationTask(BaseImageGenerationTask):
    """A task that generates a variation of an image using a prompt.

    Accepts a text prompt and image as
    input in one of the following formats:
    - tuple of (template string, ImageArtifact)
    - tuple of (TextArtifact, ImageArtifact)
    - Callable that returns a tuple of (TextArtifact, ImageArtifact).

    Attributes:
        image_generation_driver: The engine used to generate the image.
        negative_rulesets: List of negatively-weighted rulesets applied to the text␣
↪prompt, if supported by the driver.
        negative_rules: List of negatively-weighted rules applied to the text prompt,␣
↪if supported by the driver.
        output_dir: If provided, the generated image will be written to disk in␣
↪output_dir.
        output_file: If provided, the generated image will be written to disk as␣
↪output_file.
    """

    image_generation_driver: BaseImageGenerationDriver = field(
        default=Factory(lambda: Defaults.drivers_config.image_generation_driver),
        kw_only=True,
    )
    _input: Union[tuple[Union[str, TextArtifact], ImageArtifact], Callable[[BaseTask],
↪ ListArtifact], ListArtifact] = (
        field(default=None, alias="input")
    )

    @property
    def input(self) -> ListArtifact:
        if isinstance(self._input, ListArtifact):
            return self._input
        if isinstance(self._input, tuple):
            if isinstance(self._input[0], TextArtifact):
                input_text = self._input[0]
            else:
                input_text = TextArtifact(J2().render_from_string(self._input[0],␣
↪**self.full_context))

            return ListArtifact([input_text, self._input[1]])
        if isinstance(self._input, Callable):
            return self._input(self)
        raise ValueError("Input must be a tuple of (text, image) or a callable that␣
↪returns such a tuple.")

    @input.setter
    def input(self, value: tuple[str | TextArtifact, ImageArtifact] |␣
↪Callable[[BaseTask], ListArtifact]) -> None:
        self._input = value
```

(continues on next page)

(continued from previous page)

```
    def try_run(self) -> ImageArtifact:
        prompt_artifact = self.input[0]

        image_artifact = self.input[1]
        if not isinstance(image_artifact, ImageArtifact):
            raise ValueError("Image must be an ImageArtifact.")

        output_image_artifact = self.image_generation_driver.run_image_variation(
            prompts=self._get_prompts(prompt_artifact.to_text()),
            negative_prompts=self._get_negative_prompts(),
            image=image_artifact,
        )

        if self.output_dir or self.output_file:
            self._write_to_file(output_image_artifact)

        return output_image_artifact
```

- `_input = field(default=None, alias='input')` class-attribute instance-attribute
- `image_generation_driver = field(default=Factory(lambda: Defaults.drivers_config.image_generation_driver), kw_only=True)` class-attribute instance-attribute
- `input` property writable

try_run()

Source Code in griptape/tasks/variation_image_generation_task.py

```
def try_run(self) -> ImageArtifact:
    prompt_artifact = self.input[0]

    image_artifact = self.input[1]
    if not isinstance(image_artifact, ImageArtifact):
        raise ValueError("Image must be an ImageArtifact.")

    output_image_artifact = self.image_generation_driver.run_image_variation(
        prompts=self._get_prompts(prompt_artifact.to_text()),
        negative_prompts=self._get_negative_prompts(),
        image=image_artifact,
    )

    if self.output_dir or self.output_file:
        self._write_to_file(output_image_artifact)

    return output_image_artifact
```

actions_subtask

- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *BaseSubtask[Union[ListArtifact, ErrorArtifact* “)

Source Code in griptape/tasks/actions_subtask.py

```
@define
class ActionsSubtask(BaseSubtask[Union[ListArtifact, ErrorArtifact]]):
    THOUGHT_PATTERN = r"(?s)^Thought:\s*(.*?)$"
    ACTIONS_PATTERN = r"(?s)Actions:[^\[]*(\[.*\])"
    ANSWER_PATTERN = r"(?s)^Answer:\s?([\s\S]*)$"

    RESPONSE_STOP_SEQUENCE = "<|Response|>"

    thought: Optional[str] = field(default=None, kw_only=True)
    actions: list[ToolAction] = field(factory=list, kw_only=True)
    output: Optional[BaseArtifact] = field(default=None, init=False)
    generate_assistant_subtask_template: Callable[[ActionsSubtask], str] = field(
        default=Factory(lambda self: self.default_generate_assistant_subtask_template,
↪ takes_self=True),
        kw_only=True,
    )
    generate_user_subtask_template: Callable[[ActionsSubtask], str] = field(
        default=Factory(lambda self: self.default_generate_user_subtask_template,␣
↪takes_self=True),
        kw_only=True,
    )
    response_stop_sequence: str = field(default=RESPONSE_STOP_SEQUENCE, kw_only=True)
    _input: Union[str, list, tuple, BaseArtifact, Callable[[BaseTask], BaseArtifact]]␣
↪= field(
        default=lambda task: task.full_context["args"][0] if task.full_context["args
↪"] else TextArtifact(value=""),
        alias="input",
    )
    _memory: Optional[TaskMemory] = None
    _origin_task: Optional[BaseTask] = field(default=None, kw_only=True)

    @property
    def input(self) -> TextArtifact | AudioArtifact | ListArtifact:
        return self._process_task_input(self._input)

    @input.setter
    def input(self, value: str | list | tuple | BaseArtifact | Callable[[BaseTask],␣
↪BaseArtifact]) -> None:
        self._input = value

    def attach_to(self, parent_task: BaseTask) -> None:
        super().attach_to(parent_task)
        self.structure = parent_task.structure

        task_input = self.input
        try:
            if isinstance(task_input, TextArtifact) and task_input.meta.get("is_react_
↪prompt", False):
                self.__init_from_prompt(task_input.to_text())
            else:
                self.__init_from_artifact(task_input)

            # If StructuredOutputTool was used, treat the input to it as the output␣
```

(continues on next page)

(continued from previous page)

```
↪of the subtask.
            structured_outputs = [a for a in self.actions if isinstance(a.tool,␣
↪StructuredOutputTool)]
            if structured_outputs:
                output_values = [JsonArtifact(a.input["values"]) for a in structured_
↪outputs]
                if len(structured_outputs) > 1:
                    self.output = ListArtifact(output_values)
                else:
                    self.output = output_values[0]
        except Exception as e:
            logger.error("Subtask %s\nError parsing tool action: %s", self.origin_
↪task.id, e)

            self.output = ErrorArtifact(f"ToolAction input parsing error: {e}",␣
↪exception=e)

    def before_run(self) -> None:
        EventBus.publish_event(
            StartActionsSubtaskEvent(
                task_id=self.id,
                task_parent_ids=self.parent_ids,
                task_child_ids=self.child_ids,
                task_input=self.input,
                task_output=self.output,
                subtask_parent_task_id=self.origin_task.id,
                subtask_thought=self.thought,
                subtask_actions=self.actions_to_dicts(),
            ),
        )

        parts = [
            f"{self.__class__.__name__} {self.id}",
            *([f"\nThought: {self.thought}"] if self.thought else []),
            f"\nActions: {self.actions_to_json()}",
        ]
        logger.info("".join(parts))

    def try_run(self) -> ListArtifact | ErrorArtifact:
        try:
            if any(isinstance(a.output, ErrorArtifact) for a in self.actions):
                errors = [a.output.value for a in self.actions if isinstance(a.output,
↪ ErrorArtifact)]

                self.output = ErrorArtifact("\n\n".join(errors))
            else:
                results = self.run_actions(self.actions)

                actions_output = []
                for result in results:
                    tag, output = result
                    output.name = f"{tag} output"
```

(continues on next page)

(continued from previous page)

```
                    actions_output.append(output)
                self.output = ListArtifact(actions_output)
        except Exception as e:
            logger.debug("Subtask %s\n%s", self.id, e)

            self.output = ErrorArtifact(str(e), exception=e)
        if self.output is not None:
            return self.output
        return ErrorArtifact("no tool output")

    def run_actions(self, actions: list[ToolAction]) -> list[tuple[str,␣
↪BaseArtifact]]:
        with self.create_futures_executor() as futures_executor:
            return utils.execute_futures_list(
                [futures_executor.submit(with_contextvars(self.run_action), a) for a␣
↪in actions]
            )

    def run_action(self, action: ToolAction) -> tuple[str, BaseArtifact]:
        if action.tool is not None:
            if action.path is not None:
                output = action.tool.run(getattr(action.tool, action.path), self,␣
↪action)
            else:
                output = ErrorArtifact("action path not found")
        else:
            output = ErrorArtifact("action name not found")
        action.output = output

        return action.tag, output

    def after_run(self) -> None:
        response = self.output.to_text() if isinstance(self.output, BaseArtifact)␣
↪else str(self.output)

        EventBus.publish_event(
            FinishActionsSubtaskEvent(
                task_id=self.id,
                task_parent_ids=self.parent_ids,
                task_child_ids=self.child_ids,
                task_input=self.input,
                task_output=self.output,
                subtask_parent_task_id=self.origin_task.id,
                subtask_thought=self.thought,
                subtask_actions=self.actions_to_dicts(),
            ),
        )
        logger.info("%s %s\nResponse: %s", self.__class__.__name__, self.id, response)

    def actions_to_dicts(self) -> list[dict]:
        json_list = []
```

(continues on next page)

(continued from previous page)

```
        for action in self.actions:
            json_dict = {}

            if action.tag:
                json_dict["tag"] = action.tag

            if action.name:
                json_dict["name"] = action.name

            if action.path:
                json_dict["path"] = action.path

            if action.input:
                json_dict["input"] = action.input

            json_list.append(json_dict)

        return json_list

    def actions_to_json(self) -> str:
        return json.dumps(self.actions_to_dicts(), indent=2)

    def add_to_prompt_stack(self, stack: PromptStack) -> None:
        from griptape.tasks import PromptTask

        if isinstance(self.origin_task, PromptTask) and self.origin_task.prompt_
↪driver.use_native_tools:
            action_calls = [
                ToolAction(name=action.name, path=action.path, tag=action.tag,␣
↪input=action.input)
                for action in self.actions
            ]
            action_results = [
                ToolAction(
                    name=action.name,
                    path=action.path,
                    tag=action.tag,
                    output=action.output if action.output is not None else self.
↪output,
                )
                for action in self.actions
            ]

            stack.add_assistant_message(
                ListArtifact(
                    [
                        *([TextArtifact(self.thought)] if self.thought else []),
                        *[ActionArtifact(a) for a in action_calls],
                    ],
                ),
            )
```

(continues on next page)

(continued from previous page)

```
            stack.add_user_message(
                ListArtifact(
                    [
                        *[ActionArtifact(a) for a in action_results],
                        *([] if self.output else [TextArtifact("Please keep going")]),
                    ],
                ),
            )
        else:
            stack.add_assistant_message(self.generate_assistant_subtask_
↪template(self))
            stack.add_user_message(self.generate_user_subtask_template(self))

    def default_generate_assistant_subtask_template(self, subtask: ActionsSubtask) ->␣
↪str:
        return J2("tasks/prompt_task/assistant_actions_subtask.j2").render(
            stop_sequence=self.response_stop_sequence,
            subtask=subtask,
        )

    def default_generate_user_subtask_template(self, subtask: ActionsSubtask) -> str:
        return J2("tasks/prompt_task/user_actions_subtask.j2").render(
            stop_sequence=self.response_stop_sequence,
            subtask=subtask,
        )

    def _process_task_input(
        self,
        task_input: Union[str, tuple, list, BaseArtifact, Callable[[BaseTask],␣
↪BaseArtifact]],
    ) -> Union[TextArtifact, AudioArtifact, ListArtifact]:
        if isinstance(task_input, (TextArtifact, AudioArtifact, ListArtifact)):
            return task_input
        if isinstance(task_input, ActionArtifact):
            return ListArtifact([task_input])
        if isinstance(task_input, Callable):
            return self._process_task_input(task_input(self))
        if isinstance(task_input, str):
            return self._process_task_input(TextArtifact(task_input))
        if isinstance(task_input, (list, tuple)):
            return ListArtifact([self._process_task_input(elem) for elem in task_
↪input])
        raise ValueError(f"Invalid input type: {type(task_input)} ")

    def __init_from_prompt(self, value: str) -> None:
        thought_matches = re.findall(self.THOUGHT_PATTERN, value, re.MULTILINE)
        actions_matches = re.findall(self.ACTIONS_PATTERN, value, re.DOTALL)
        answer_matches = re.findall(self.ANSWER_PATTERN, value, re.MULTILINE)

        self.actions = self.__parse_actions(actions_matches)

        if thought_matches:
```

(continues on next page)

(continued from previous page)

```
            self.thought = thought_matches[-1]

        if not self.actions and self.output is None:
            if answer_matches:
                # A direct answer is provided, set it as the output.
                self.output = TextArtifact(answer_matches[-1])
            else:
                # The LLM failed to follow the ReAct prompt, set the LLMs raw␣
↪response as the output.
                self.output = TextArtifact(value)

    def __init_from_artifact(self, artifact: TextArtifact | AudioArtifact |␣
↪ListArtifact) -> None:
        """Parses the input Artifact to extract either a final answer or thought and␣
↪actions.

        When the input Artifact is a TextArtifact, it is assumed to be the final␣
↪answer.
        When the input Artifact is a ListArtifact, it is assumed to contain both␣
↪thought and actions.
        Text Artifacts are parsed as the thought, and ToolAction Artifacts parsed as␣
↪the actions.

        Args:
            artifact: The input Artifacts.

        Returns:
            None
        """
        # When using native tools, we can assume that a TextArtifact or AudioArtifact␣
↪is the LLM providing its final answer.
        if isinstance(artifact, (TextArtifact, AudioArtifact)):
            self.output = artifact
            return

        self.actions = [
            self.__process_action_object(artifact.value.to_dict())
            for artifact in artifact.value
            if isinstance(artifact, ActionArtifact)
        ]

        # When parsing from Artifacts we cant determine the thought unless there are␣
↪also Actions
        if self.actions:
            thoughts = [artifact.value for artifact in artifact.value if␣
↪isinstance(artifact, TextArtifact)]
            if thoughts:
                self.thought = thoughts[0]
        elif self.output is None:
            self.output = TextArtifact(artifact.to_text())

    def __parse_actions(self, actions_matches: list[str]) -> list[ToolAction]:
```

(continues on next page)

(continued from previous page)

```
        if len(actions_matches) == 0:
            return []
        try:
            data = actions_matches[-1]
            actions_list: list[dict] = json.loads(data, strict=False)

            return [self.__process_action_object(action_object) for action_object in␣
↪actions_list]
        except json.JSONDecodeError as e:
            logger.debug("Subtask %s\nInvalid actions JSON: %s", self.origin_task.id,␣
↪e)

            self.output = ErrorArtifact(f"Actions JSON decoding error: {e}",␣
↪exception=e)

            return []

    def __process_action_object(self, action_object: dict) -> ToolAction:
        # Load action tag; throw exception if the key is not present
        action_tag = action_object["tag"]

        # Load action name; throw exception if the key is not present
        action_name = action_object["name"]

        # Load action method; throw exception if the key is not present
        action_path = action_object["path"]

        # Load optional input value; dont throw exceptions if key is not present
        if "input" in action_object:
            # Some LLMs dont support nested parameters and therefore wont generate
↪"values".
            # So we need to manually add it here.
            if "values" not in action_object["input"]:
                action_object["input"] = {"values": action_object["input"]}

            # The schema library has a bug, where something like `Or(str, None)`␣
↪doesnt get
            # correctly translated into JSON schema. For some optional input fields␣
↪LLMs sometimes
            # still provide null value, which trips up the validator. The temporary␣
↪solution that
            # works is to strip all key-values where value is null.
            action_input = remove_null_values_in_dict_recursively(action_object["input
↪"])
        else:
            action_input = {}

        # Load the action itself
        if isinstance(self.origin_task, ActionsSubtaskOriginMixin):
            tool = self.origin_task.find_tool(action_name)
        else:
            raise Exception("ActionSubtask must be attached to a Task that implements␣
```

(continues on next page)

(continued from previous page)

```
↪ActionSubtaskOriginMixin.")

        action = ToolAction(tag=action_tag, name=action_name, path=action_path,␣
↪input=action_input, tool=tool)

        self.__validate_action(action)

        return action

    def __validate_action(self, action: ToolAction) -> None:
        if action.tool is None:
            return

        if action.path is None:
            raise Exception("ToolAction path not found.")
        activity = getattr(action.tool, action.path)

        if activity is None:
            raise Exception("Activity not found.")

        activity_schema = action.tool.activity_schema(activity)

        if activity_schema is None or action.input is None:
            return

        try:
            action.tool.validate_activity_schema(activity_schema, action.input)
        except ValueError as e:
            logger.debug("Subtask %s\nInvalid action JSON: %s", self.origin_task.id,␣
↪e)

            action.output = ErrorArtifact(f"Activity input JSON validation error: {e}
↪", exception=e)
```

- `ACTIONS_PATTERN = '(?s)Actions:[^\\[]*(\\[.*\\])'` class-attribute instance-attribute
- `ANSWER_PATTERN = '(?s)^Answer:\\s?([\\s\\S]*)$'` class-attribute instance-attribute
- `RESPONSE_STOP_SEQUENCE = '<|Response|>'` class-attribute instance-attribute
- `THOUGHT_PATTERN = '(?s)^Thought:\\s*(.*?)$'` class-attribute instance-attribute
- `_input = field(default=lambda task: task.full_context['args'][0] if task.full_context['args'] else TextArtifact(value=''), alias='input')` class-attribute instance-attribute
- `_memory = None` class-attribute instance-attribute
- `_origin_task = field(default=None, kw_only=True)` class-attribute instance-attribute
- `actions = field(factory=list, kw_only=True)` class-attribute instance-attribute
- `generate_assistant_subtask_template = field(default=Factory(lambda self: self.default_generate_assistant_subtask_template, takes_self=True), kw_only=True)` class-attribute instance-attribute

- `generate_user_subtask_template = field(default=Factory(lambda self: self.default_generate_user_subtask_template, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `input` property writable
- `output = field(default=None, init=False)` class-attribute instance-attribute
- `response_stop_sequence = field(default=RESPONSE_STOP_SEQUENCE, kw_only=True)` class-attribute instance-attribute
- `thought = field(default=None, kw_only=True)` class-attribute instance-attribute

__init_from_artifact(artifact)

Parses the input Artifact to extract either a final answer or thought and actions.

When the input Artifact is a TextArtifact, it is assumed to be the final answer. When the input Artifact is a ListArtifact, it is assumed to contain both thought and actions. Text Artifacts are parsed as the thought, and ToolAction Artifacts parsed as the actions.

Parameters

Name	Type	Description	Default
artifact	<code class="language-pythc I AudioArtifact I ListArtifact</code>	The input Artifacts. 	*required*

Returns

Type	Description
None	None

Source Code in griptape/tasks/actions_subtask.py

```
def __init_from_artifact(self, artifact: TextArtifact | AudioArtifact | ListArtifact)␣
↪-> None:
    """Parses the input Artifact to extract either a final answer or thought and␣
↪actions.

    When the input Artifact is a TextArtifact, it is assumed to be the final answer.
    When the input Artifact is a ListArtifact, it is assumed to contain both thought␣
↪and actions.
    Text Artifacts are parsed as the thought, and ToolAction Artifacts parsed as the␣
```

(continues on next page)

(continued from previous page)

```
↪actions.

    Args:
        artifact: The input Artifacts.

    Returns:
        None
    """
    # When using native tools, we can assume that a TextArtifact or AudioArtifact is␣
↪the LLM providing its final answer.
    if isinstance(artifact, (TextArtifact, AudioArtifact)):
        self.output = artifact
        return

    self.actions = [
        self.__process_action_object(artifact.value.to_dict())
        for artifact in artifact.value
        if isinstance(artifact, ActionArtifact)
    ]

    # When parsing from Artifacts we cant determine the thought unless there are also␣
↪Actions
    if self.actions:
        thoughts = [artifact.value for artifact in artifact.value if␣
↪isinstance(artifact, TextArtifact)]
        if thoughts:
            self.thought = thoughts[0]
    elif self.output is None:
        self.output = TextArtifact(artifact.to_text())
```

__init_from_prompt(value)

Source Code in griptape/tasks/actions_subtask.py

```
def __init_from_prompt(self, value: str) -> None:
    thought_matches = re.findall(self.THOUGHT_PATTERN, value, re.MULTILINE)
    actions_matches = re.findall(self.ACTIONS_PATTERN, value, re.DOTALL)
    answer_matches = re.findall(self.ANSWER_PATTERN, value, re.MULTILINE)

    self.actions = self.__parse_actions(actions_matches)

    if thought_matches:
        self.thought = thought_matches[-1]

    if not self.actions and self.output is None:
        if answer_matches:
            # A direct answer is provided, set it as the output.
            self.output = TextArtifact(answer_matches[-1])
        else:
            # The LLM failed to follow the ReAct prompt, set the LLMs raw response as␣
↪the output.
            self.output = TextArtifact(value)
```

__parse_actions(actions_matches)

Source Code in griptape/tasks/actions_subtask.py

```
def __parse_actions(self, actions_matches: list[str]) -> list[ToolAction]:
    if len(actions_matches) == 0:
        return []
    try:
        data = actions_matches[-1]
        actions_list: list[dict] = json.loads(data, strict=False)

        return [self.__process_action_object(action_object) for action_object in␣
↪actions_list]
    except json.JSONDecodeError as e:
        logger.debug("Subtask %s\nInvalid actions JSON: %s", self.origin_task.id, e)

        self.output = ErrorArtifact(f"Actions JSON decoding error: {e}", exception=e)

        return []
```

__process_action_object(action_object)

Source Code in griptape/tasks/actions_subtask.py

```
def __process_action_object(self, action_object: dict) -> ToolAction:
    # Load action tag; throw exception if the key is not present
    action_tag = action_object["tag"]

    # Load action name; throw exception if the key is not present
    action_name = action_object["name"]

    # Load action method; throw exception if the key is not present
    action_path = action_object["path"]

    # Load optional input value; dont throw exceptions if key is not present
    if "input" in action_object:
        # Some LLMs dont support nested parameters and therefore wont generate "values
↪".
        # So we need to manually add it here.
        if "values" not in action_object["input"]:
            action_object["input"] = {"values": action_object["input"]}

        # The schema library has a bug, where something like `Or(str, None)` doesnt␣
↪get
        # correctly translated into JSON schema. For some optional input fields LLMs␣
↪sometimes
        # still provide null value, which trips up the validator. The temporary␣
↪solution that
        # works is to strip all key-values where value is null.
        action_input = remove_null_values_in_dict_recursively(action_object["input"])
    else:
        action_input = {}
```

(continues on next page)

(continued from previous page)

```
    # Load the action itself
    if isinstance(self.origin_task, ActionsSubtaskOriginMixin):
        tool = self.origin_task.find_tool(action_name)
    else:
        raise Exception("ActionSubtask must be attached to a Task that implements␣
↪ActionSubtaskOriginMixin.")

    action = ToolAction(tag=action_tag, name=action_name, path=action_path,␣
↪input=action_input, tool=tool)

    self.__validate_action(action)

    return action
```

__validate_action(action)

Source Code in griptape/tasks/actions_subtask.py

```
def __validate_action(self, action: ToolAction) -> None:
    if action.tool is None:
        return

    if action.path is None:
        raise Exception("ToolAction path not found.")
    activity = getattr(action.tool, action.path)

    if activity is None:
        raise Exception("Activity not found.")

    activity_schema = action.tool.activity_schema(activity)

    if activity_schema is None or action.input is None:
        return

    try:
        action.tool.validate_activity_schema(activity_schema, action.input)
    except ValueError as e:
        logger.debug("Subtask %s\nInvalid action JSON: %s", self.origin_task.id, e)

        action.output = ErrorArtifact(f"Activity input JSON validation error: {e}",␣
↪exception=e)
```

_process_task_input(task_input)

Source Code in griptape/tasks/actions_subtask.py

```
def _process_task_input(
    self,
    task_input: Union[str, tuple, list, BaseArtifact, Callable[[BaseTask],␣
↪BaseArtifact]],
) -> Union[TextArtifact, AudioArtifact, ListArtifact]:
```

(continues on next page)

(continued from previous page)

```
    if isinstance(task_input, (TextArtifact, AudioArtifact, ListArtifact)):
        return task_input
    if isinstance(task_input, ActionArtifact):
        return ListArtifact([task_input])
    if isinstance(task_input, Callable):
        return self._process_task_input(task_input(self))
    if isinstance(task_input, str):
        return self._process_task_input(TextArtifact(task_input))
    if isinstance(task_input, (list, tuple)):
        return ListArtifact([self._process_task_input(elem) for elem in task_input])
    raise ValueError(f"Invalid input type: {type(task_input)} ")
```

actions_to_dicts()

Source Code in griptape/tasks/actions_subtask.py

```
def actions_to_dicts(self) -> list[dict]:
    json_list = []

    for action in self.actions:
        json_dict = {}

        if action.tag:
            json_dict["tag"] = action.tag

        if action.name:
            json_dict["name"] = action.name

        if action.path:
            json_dict["path"] = action.path

        if action.input:
            json_dict["input"] = action.input

        json_list.append(json_dict)

    return json_list
```

actions_to_json()

Source Code in griptape/tasks/actions_subtask.py

```
def actions_to_json(self) -> str:
    return json.dumps(self.actions_to_dicts(), indent=2)
```

add_to_prompt_stack(stack)

Source Code in griptape/tasks/actions_subtask.py

```
def add_to_prompt_stack(self, stack: PromptStack) -> None:
    from griptape.tasks import PromptTask
```

(continues on next page)

(continued from previous page)

```
    if isinstance(self.origin_task, PromptTask) and self.origin_task.prompt_driver.
↪use_native_tools:
        action_calls = [
            ToolAction(name=action.name, path=action.path, tag=action.tag,␣
↪input=action.input)
            for action in self.actions
        ]
        action_results = [
            ToolAction(
                name=action.name,
                path=action.path,
                tag=action.tag,
                output=action.output if action.output is not None else self.output,
            )
            for action in self.actions
        ]

        stack.add_assistant_message(
            ListArtifact(
                [
                    *([TextArtifact(self.thought)] if self.thought else []),
                    *[ActionArtifact(a) for a in action_calls],
                ],
            ),
        )
        stack.add_user_message(
            ListArtifact(
                [
                    *[ActionArtifact(a) for a in action_results],
                    *([] if self.output else [TextArtifact("Please keep going")]),
                ],
            ),
        )
    else:
        stack.add_assistant_message(self.generate_assistant_subtask_template(self))
        stack.add_user_message(self.generate_user_subtask_template(self))
```

after_run()

Source Code in griptape/tasks/actions_subtask.py

```
def after_run(self) -> None:
    response = self.output.to_text() if isinstance(self.output, BaseArtifact) else␣
↪str(self.output)

    EventBus.publish_event(
        FinishActionsSubtaskEvent(
            task_id=self.id,
            task_parent_ids=self.parent_ids,
            task_child_ids=self.child_ids,
            task_input=self.input,
```

(continues on next page)

(continued from previous page)

```
            task_output=self.output,
            subtask_parent_task_id=self.origin_task.id,
            subtask_thought=self.thought,
            subtask_actions=self.actions_to_dicts(),
        ),
    )
    logger.info("%s %s\nResponse: %s", self.__class__.__name__, self.id, response)
```

attach_to(parent_task)

Source Code in griptape/tasks/actions_subtask.py

```
def attach_to(self, parent_task: BaseTask) -> None:
    super().attach_to(parent_task)
    self.structure = parent_task.structure

    task_input = self.input
    try:
        if isinstance(task_input, TextArtifact) and task_input.meta.get("is_react_
↪prompt", False):
            self.__init_from_prompt(task_input.to_text())
        else:
            self.__init_from_artifact(task_input)

        # If StructuredOutputTool was used, treat the input to it as the output of␣
↪the subtask.
        structured_outputs = [a for a in self.actions if isinstance(a.tool,␣
↪StructuredOutputTool)]
        if structured_outputs:
            output_values = [JsonArtifact(a.input["values"]) for a in structured_
↪outputs]
            if len(structured_outputs) > 1:
                self.output = ListArtifact(output_values)
            else:
                self.output = output_values[0]
    except Exception as e:
        logger.error("Subtask %s\nError parsing tool action: %s", self.origin_task.id,
↪ e)

        self.output = ErrorArtifact(f"ToolAction input parsing error: {e}",␣
↪exception=e)
```

before_run()

Source Code in griptape/tasks/actions_subtask.py

```
def before_run(self) -> None:
    EventBus.publish_event(
        StartActionsSubtaskEvent(
            task_id=self.id,
            task_parent_ids=self.parent_ids,
```

(continues on next page)

(continued from previous page)

```
            task_child_ids=self.child_ids,
            task_input=self.input,
            task_output=self.output,
            subtask_parent_task_id=self.origin_task.id,
            subtask_thought=self.thought,
            subtask_actions=self.actions_to_dicts(),
        ),
    )

    parts = [
        f"{self.__class__.__name__} {self.id}",
        *([f"\nThought: {self.thought}"] if self.thought else []),
        f"\nActions: {self.actions_to_json()}",
    ]
    logger.info("".join(parts))
```

default_generate_assistant_subtask_template(subtask)

Source Code in griptape/tasks/actions_subtask.py

```
def default_generate_assistant_subtask_template(self, subtask: ActionsSubtask) -> str:
    return J2("tasks/prompt_task/assistant_actions_subtask.j2").render(
        stop_sequence=self.response_stop_sequence,
        subtask=subtask,
    )
```

default_generate_user_subtask_template(subtask)

Source Code in griptape/tasks/actions_subtask.py

```
def default_generate_user_subtask_template(self, subtask: ActionsSubtask) -> str:
    return J2("tasks/prompt_task/user_actions_subtask.j2").render(
        stop_sequence=self.response_stop_sequence,
        subtask=subtask,
    )
```

run_action(action)

Source Code in griptape/tasks/actions_subtask.py

```
def run_action(self, action: ToolAction) -> tuple[str, BaseArtifact]:
    if action.tool is not None:
        if action.path is not None:
            output = action.tool.run(getattr(action.tool, action.path), self, action)
        else:
            output = ErrorArtifact("action path not found")
    else:
        output = ErrorArtifact("action name not found")
    action.output = output

    return action.tag, output
```

run_actions(actions)

Source Code in griptape/tasks/actions_subtask.py

```
def run_actions(self, actions: list[ToolAction]) -> list[tuple[str, BaseArtifact]]:
    with self.create_futures_executor() as futures_executor:
        return utils.execute_futures_list(
            [futures_executor.submit(with_contextvars(self.run_action), a) for a in␣
↪actions]
        )
```

try_run()

Source Code in griptape/tasks/actions_subtask.py

```
def try_run(self) -> ListArtifact | ErrorArtifact:
    try:
        if any(isinstance(a.output, ErrorArtifact) for a in self.actions):
            errors = [a.output.value for a in self.actions if isinstance(a.output,␣
↪ErrorArtifact)]

            self.output = ErrorArtifact("\n\n".join(errors))
        else:
            results = self.run_actions(self.actions)

            actions_output = []
            for result in results:
                tag, output = result
                output.name = f"{tag} output"

                actions_output.append(output)
            self.output = ListArtifact(actions_output)
    except Exception as e:
        logger.debug("Subtask %s\n%s", self.id, e)

        self.output = ErrorArtifact(str(e), exception=e)
    if self.output is not None:
        return self.output
    return ErrorArtifact("no tool output")
```

assistant_task

Bases: *BaseTextInputTask[TextArtifact* “)

Attributes

Name	Type	Description
<a href="assistant_task.mdx#griptape.tasks.assistant_task.AssistantTask.assistant_driver" title="assistant_driver = field(kw_only=True) class-attribute instance-attribute (griptape.tasks.assistant_task.AssistantTask.assistant_driver)"><code class="language-python">	<a href="../drivers/assistant#griptape.drivers.assistant.BaseAssistantDriver" title="BaseAssistantDriver (griptape.drivers.assistant.BaseAssistantDriver)"><code class="language-python">BaseAssistantDriver</code>	Driver to run the Assistant.

Source Code in griptape/tasks/assistant_task.py

```
@define
class AssistantTask(BaseTextInputTask[TextArtifact]):
    """Task to run an Assistant.

    Attributes:
        assistant_driver: Driver to run the Assistant.
    """

    assistant_driver: BaseAssistantDriver = field(kw_only=True)

    def try_run(self) -> TextArtifact:
        return self.assistant_driver.run(self.input)
```

- `assistant_driver = field(kw_only=True)` class-attribute instance-attribute

try_run()

Source Code in griptape/tasks/assistant_task.py

```
def try_run(self) -> TextArtifact:
    return self.assistant_driver.run(self.input)
```

audio_transcription_task

Bases: *BaseAudioInputTask[TextArtifact* “)

Source Code in griptape/tasks/audio_transcription_task.py

```
@define
class AudioTranscriptionTask(BaseAudioInputTask[TextArtifact]):
    audio_transcription_driver: BaseAudioTranscriptionDriver = field(
        default=Factory(lambda: Defaults.drivers_config.audio_transcription_driver),
        kw_only=True,
    )
```

(continues on next page)

(continued from previous page)

```
    def try_run(self) -> TextArtifact:
        return self.audio_transcription_driver.run(self.input)
```

- `audio_transcription_driver = field(default=Factory(lambda: Defaults.drivers_config.audio_transcription_driver), kw_only=True)` class-attribute instance-attribute

try_run()

Source Code in griptape/tasks/audio_transcription_task.py

```
def try_run(self) -> TextArtifact:
    return self.audio_transcription_driver.run(self.input)
```

base_audio_generation_task

- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *ArtifactFileOutputMixin* “) , *RuleMixin* ”) , *BaseTask[AudioArtifact* “) , ABC

Source Code in griptape/tasks/base_audio_generation_task.py

```
@define
class BaseAudioGenerationTask(ArtifactFileOutputMixin, RuleMixin,␣
↪BaseTask[AudioArtifact], ABC):
    def before_run(self) -> None:
        super().before_run()

        logger.info("%s %s\nInput: %s", self.__class__.__name__, self.id, self.input.
↪to_text())

    def after_run(self) -> None:
        super().after_run()

        logger.info(
            "%s %s\nOutput: %s",
            self.__class__.__name__,
            self.id,
            self.output.to_text() if self.output is not None else "",
        )
```

after_run()

Source Code in griptape/tasks/base_audio_generation_task.py

```
def after_run(self) -> None:
    super().after_run()

    logger.info(
        "%s %s\nOutput: %s",
        self.__class__.__name__,
        self.id,
        self.output.to_text() if self.output is not None else "",
    )
```

before_run()

Source Code in griptape/tasks/base_audio_generation_task.py

```
def before_run(self) -> None:
    super().before_run()

    logger.info("%s %s\nInput: %s", self.__class__.__name__, self.id, self.input.to_
↪text())
```

base_audio_input_task

- `T = TypeVar('T', bound=BaseArtifact)` module-attribute
- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *RuleMixin* “) , *BaseTask[T* ”) , ABC

Source Code in griptape/tasks/base_audio_input_task.py

```
@define
class BaseAudioInputTask(RuleMixin, BaseTask[T], ABC):
    _input: Union[AudioArtifact, Callable[[BaseTask], AudioArtifact]] = field(alias=
↪"input")

    @property
    def input(self) -> AudioArtifact:
        if isinstance(self._input, AudioArtifact):
            return self._input
        if isinstance(self._input, Callable):
            return self._input(self)
        raise ValueError("Input must be an AudioArtifact.")

    @input.setter
    def input(self, value: AudioArtifact | Callable[[BaseTask], AudioArtifact]) ->␣
↪None:
        self._input = value

    def before_run(self) -> None:
        super().before_run()

        logger.info("%s %s\nInput: %s", self.__class__.__name__, self.id, self.input.
↪to_text())

    def after_run(self) -> None:
        super().after_run()

        logger.info(
            "%s %s\nOutput: %s",
            self.__class__.__name__,
            self.id,
            self.output.to_text() if self.output is not None else "",
        )
```

- `_input = field(alias='input')` class-attribute instance-attribute

- `input` property writable

after_run()

Source Code in griptape/tasks/base_audio_input_task.py

```
def after_run(self) -> None:
    super().after_run()

    logger.info(
        "%s %s\nOutput: %s",
        self.__class__.__name__,
        self.id,
        self.output.to_text() if self.output is not None else "",
    )
```

before_run()

Source Code in griptape/tasks/base_audio_input_task.py

```
def before_run(self) -> None:
    super().before_run()

    logger.info("%s %s\nInput: %s", self.__class__.__name__, self.id, self.input.to_
↪text())
```

base_image_generation_task

- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *ArtifactFileOutputMixin* “) , *RuleMixin* ”) , *BaseTask[ImageArtifact* “) , ABC

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">list[Ruleset]</code>	List of negatively-weighted rulesets applied to the text prompt, if supported by the driver.

continues on next page

Table 375 – continued from previous page

Name	Type	Description
<code class="language-python">	<code class="language-python">list[Rule]</code>	List of negatively-weighted rules applied to the text prompt, if supported by the driver.
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.base_image_generation_task.BaseImageGenerationTask.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk in output_dir.

continues on next page

Table 375 – continued from previous page

Name	Type	Description
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.base_image_generation_task.BaseImageGenerationTask.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk as output_file.

Source Code in griptape/tasks/base_image_generation_task.py

```
@define
class BaseImageGenerationTask(ArtifactFileOutputMixin, RuleMixin,␣
↪BaseTask[ImageArtifact], ABC):
    """Provides a base class for image generation-related tasks.

    Attributes:
        negative_rulesets: List of negatively-weighted rulesets applied to the text␣
↪prompt, if supported by the driver.
        negative_rules: List of negatively-weighted rules applied to the text prompt,␣
↪if supported by the driver.
        output_dir: If provided, the generated image will be written to disk in␣
↪output_dir.
        output_file: If provided, the generated image will be written to disk as␣
↪output_file.
    """

    DEFAULT_NEGATIVE_RULESET_NAME = "Negative Ruleset"

    image_generation_driver: BaseImageGenerationDriver = field(
        default=Factory(lambda: Defaults.drivers_config.image_generation_driver),
        kw_only=True,
    )
    _negative_rulesets: list[Ruleset] = field(factory=list, kw_only=True, alias=
↪"negative_rulesets")
    negative_rules: list[Rule] = field(factory=list, kw_only=True)

    @property
    def negative_rulesets(self) -> list[Ruleset]:
```

(continues on next page)

(continued from previous page)

```
        negative_rulesets = self._negative_rulesets

        if self.negative_rules:
            negative_rulesets.append(Ruleset(name=self.DEFAULT_NEGATIVE_RULESET_NAME,␣
↪rules=self.negative_rules))

        return negative_rulesets

    def _read_from_file(self, path: str) -> ImageArtifact:
        logger.info("Reading image from %s", os.path.abspath(path))
        return ImageLoader().load(Path(path))

    def _get_prompts(self, prompt: str) -> list[str]:
        return [prompt, *[rule.value for ruleset in self.rulesets for rule in ruleset.
↪rules]]

    def _get_negative_prompts(self) -> list[str]:
        return [rule.value for ruleset in self.negative_rulesets for rule in ruleset.
↪rules]
```

- `DEFAULT_NEGATIVE_RULESET_NAME = 'Negative Ruleset'` class-attribute instance-attribute
- `_negative_rulesets = field(factory=list, kw_only=True, alias='negative_rulesets')` class-attribute instance-attribute
- `image_generation_driver = field(default=Factory(lambda: Defaults.drivers_config.image_generation_driver), kw_only=True)` class-attribute instance-attribute
- `negative_rules = field(factory=list, kw_only=True)` class-attribute instance-attribute
- `negative_rulesets` property

_get_negative_prompts()

Source Code in griptape/tasks/base_image_generation_task.py

```
def _get_negative_prompts(self) -> list[str]:
    return [rule.value for ruleset in self.negative_rulesets for rule in ruleset.
↪rules]
```

_get_prompts(prompt)

Source Code in griptape/tasks/base_image_generation_task.py

```
def _get_prompts(self, prompt: str) -> list[str]:
    return [prompt, *[rule.value for ruleset in self.rulesets for rule in ruleset.
↪rules]]
```

_read_from_file(path)

Source Code in griptape/tasks/base_image_generation_task.py

```
def _read_from_file(self, path: str) -> ImageArtifact:
    logger.info("Reading image from %s", os.path.abspath(path))
    return ImageLoader().load(Path(path))
```

base_subtask

- `T = TypeVar('T', bound=BaseArtifact)` module-attribute
- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *BaseTask[T* “)

Source Code in griptape/tasks/base_subtask.py

```
@define
class BaseSubtask(BaseTask[T]):
    @property
    def origin_task(self) -> BaseTask:
        if self._origin_task is not None:
            return self._origin_task
        raise Exception("ActionSubtask has no origin task.")

    @property
    def parents(self) -> list[BaseTask]:
        if isinstance(self.origin_task, ActionsSubtaskOriginMixin):
            return [self.origin_task.find_subtask(parent_id) for parent_id in self.
↪parent_ids]
        raise Exception("ActionSubtask must be attached to a Task that implements␣
↪ActionSubtaskOriginMixin.")

    @property
    def children(self) -> list[BaseTask]:
        if isinstance(self.origin_task, ActionsSubtaskOriginMixin):
            return [self.origin_task.find_subtask(child_id) for child_id in self.
↪child_ids]
        raise Exception("ActionSubtask must be attached to a Task that implements␣
↪ActionSubtaskOriginMixin.")

    def add_child(self, child: BaseTask) -> BaseTask:
        if child.id not in self.child_ids:
            self.child_ids.append(child.id)
        return child

    def add_parent(self, parent: BaseTask) -> BaseTask:
        if parent.id not in self.parent_ids:
            self.parent_ids.append(parent.id)
        return parent

    def attach_to(self, parent_task: BaseTask) -> None:
        self._origin_task = parent_task

    @abstractmethod
    def add_to_prompt_stack(self, stack: PromptStack) -> None: ...
```

- `children` property

- `origin_task` property
- `parents` property

add_child(child)

Source Code in griptape/tasks/base_subtask.py

```
def add_child(self, child: BaseTask) -> BaseTask:
    if child.id not in self.child_ids:
        self.child_ids.append(child.id)
    return child
```

add_parent(parent)

Source Code in griptape/tasks/base_subtask.py

```
def add_parent(self, parent: BaseTask) -> BaseTask:
    if parent.id not in self.parent_ids:
        self.parent_ids.append(parent.id)
    return parent
```

add_to_prompt_stack(stack)abstractmethod

Source Code in griptape/tasks/base_subtask.py

```
@abstractmethod
def add_to_prompt_stack(self, stack: PromptStack) -> None: ...
```

attach_to(parent_task)

Source Code in griptape/tasks/base_subtask.py

```
def attach_to(self, parent_task: BaseTask) -> None:
    self._origin_task = parent_task
```

base_task

- `T = TypeVar('T', bound=BaseArtifact)` module-attribute
- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *FuturesExecutorMixin* “) , *SerializableMixin* ”) , *RunnableMixin['BaseTask'* “) , ABC, *Generic[T*

Source Code in griptape/tasks/base_task.py

```
@define
class BaseTask(FuturesExecutorMixin, SerializableMixin, RunnableMixin["BaseTask"],␣
↪ABC, Generic[T]):
    class State(Enum):
        PENDING = 1
        RUNNING = 2
        FINISHED = 3
        SKIPPED = 4
```

(continues on next page)

(continued from previous page)

```
    id: str = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata=
↪{"serializable": True})
    state: State = field(default=State.PENDING, kw_only=True, metadata={"serializable
↪": True})
    parent_ids: list[str] = field(factory=list, kw_only=True, metadata={"serializable
↪": True})
    child_ids: list[str] = field(factory=list, kw_only=True, metadata={"serializable
↪": True})
    max_meta_memory_entries: Optional[int] = field(default=20, kw_only=True, metadata=
↪{"serializable": True})
    structure: Optional[Structure] = field(default=None, kw_only=True)

    output: Optional[T] = field(default=None, init=False)
    context: dict[str, Any] = field(factory=dict, kw_only=True, metadata={
↪"serializable": True})
    _execution_args: tuple = field(factory=tuple, init=False)

    @property
    def execution_args(self) -> tuple:
        return self._execution_args

    def __rshift__(self, other: BaseTask | list[BaseTask]) -> BaseTask |␣
↪list[BaseTask]:
        if isinstance(other, list):
            self.add_children(other)
        else:
            self.add_child(other)

        return other

    def __lshift__(self, other: BaseTask | list[BaseTask]) -> BaseTask |␣
↪list[BaseTask]:
        if isinstance(other, list):
            self.add_parents(other)
        else:
            self.add_parent(other)

        return other

    def __attrs_post_init__(self) -> None:
        if self.structure is not None:
            self.structure.add_task(self)

    @property
    @abstractmethod
    def input(self) -> BaseArtifact: ...

    @property
    def parents(self) -> list[BaseTask]:
        if self.structure is not None:
            return [self.structure.find_task(parent_id) for parent_id in self.parent_
```

(continues on next page)

(continued from previous page)

```
↪ids]
        raise ValueError("Structure must be set to access parents")

    @property
    def children(self) -> list[BaseTask]:
        if self.structure is not None:
            return [self.structure.find_task(child_id) for child_id in self.child_ids]
        raise ValueError("Structure must be set to access children")

    @property
    def parent_outputs(self) -> dict[str, BaseArtifact]:
        return {parent.id: parent.output for parent in self.parents if parent.output}

    @property
    def parents_output_text(self) -> str:
        return "\n".join([parent.output.to_text() for parent in self.parents if␣
↪parent.output])

    @property
    def meta_memories(self) -> list[BaseMetaEntry]:
        if self.structure is not None and self.structure.meta_memory:
            if self.max_meta_memory_entries:
                return self.structure.meta_memory.entries[: self.max_meta_memory_
↪entries]
            return self.structure.meta_memory.entries
        return []

    def __str__(self) -> str:
        return str(self.output.value) if self.output is not None else ""

    def add_parents(self, parents: list[BaseTask]) -> None:
        for parent in parents:
            self.add_parent(parent)

    def add_parent(self, parent: BaseTask) -> BaseTask:
        if parent.id not in self.parent_ids:
            self.parent_ids.append(parent.id)

        if self.id not in parent.child_ids:
            parent.child_ids.append(self.id)

        if self.structure is not None and parent not in self.structure.tasks:
            self.structure.add_task(parent)

        return self

    def add_children(self, children: list[BaseTask]) -> None:
        for child in children:
            self.add_child(child)

    def add_child(self, child: BaseTask) -> BaseTask:
        if child.id not in self.child_ids:
```

(continues on next page)

(continued from previous page)

```
            self.child_ids.append(child.id)

        if self.id not in child.parent_ids:
            child.parent_ids.append(self.id)

        if self.structure is not None and child not in self.structure.tasks:
            self.structure.add_task(child)

        return self

    def preprocess(self, structure: Structure) -> BaseTask:
        self.structure = structure

        return self

    def is_pending(self) -> bool:
        return self.state == BaseTask.State.PENDING

    def is_finished(self) -> bool:
        return self.state == BaseTask.State.FINISHED

    def is_running(self) -> bool:
        return self.state == BaseTask.State.RUNNING

    def is_skipped(self) -> bool:
        return self.state == BaseTask.State.SKIPPED

    def before_run(self) -> None:
        super().before_run()
        if self.structure is not None:
            EventBus.publish_event(
                StartTaskEvent(
                    task_id=self.id,
                    task_parent_ids=self.parent_ids,
                    task_child_ids=self.child_ids,
                    task_input=self.input,
                    task_output=self.output,
                ),
            )

    def run(self, *args) -> T:
        try:
            self._execution_args = args

            self.state = BaseTask.State.RUNNING

            self.before_run()

            self.output = self.try_run()

            self.after_run()
        except Exception as e:
```

(continues on next page)

(continued from previous page)

```
            logger.exception("%s %s\n%s", self.__class__.__name__, self.id, e)

            self.output = cast("T", ErrorArtifact(str(e), exception=e))
        finally:
            self.state = BaseTask.State.FINISHED

        return self.output

    def after_run(self) -> None:
        super().after_run()
        if self.structure is not None:
            EventBus.publish_event(
                FinishTaskEvent(
                    task_id=self.id,
                    task_parent_ids=self.parent_ids,
                    task_child_ids=self.child_ids,
                    task_input=self.input,
                    task_output=self.output,
                ),
            )

    def can_run(self) -> bool:
        # If this Task has been skipped or is not pending, it should not run
        if self.is_skipped() or not self.is_pending():
            return False

        # If this Task has parents, and _all_ of them are skipped, it should not run
        if self.parents and all(parent.is_skipped() for parent in self.parents):
            self.state = BaseTask.State.SKIPPED
            return False

        # If _all_ this Tasks unskipped parents are finished, it should run
        unskipped_parents = [parent for parent in self.parents if not parent.is_
↪skipped()]

        return all(parent.is_finished() for parent in unskipped_parents)

    def reset(self) -> BaseTask:
        self.state = BaseTask.State.PENDING
        self.output = None
        self._execution_args = ()

        return self

    @abstractmethod
    def try_run(self) -> T: ...

    @property
    def full_context(self) -> dict[str, Any]:
        # Need to deep copy so that the serialized context doesnt contain non-
↪serializable data
        context = deepcopy(self.context)
```

(continues on next page)

(continued from previous page)

```
        if self.structure is None:
            context.update({"args": self._execution_args})
        else:
            context.update(self.structure.context(self))

        return context
```

- `_execution_args = field(factory=tuple, init=False)` class-attribute instance-attribute
- `child_ids = field(factory=list, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `children` property
- `context = field(factory=dict, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `execution_args` property
- `full_context` property
- `id = field(default=Factory(lambda: uuid.uuid4().hex), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `input` abstractmethod property
- `max_meta_memory_entries = field(default=20, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `meta_memories` property
- `output = field(default=None, init=False)` class-attribute instance-attribute
- `parent_ids = field(factory=list, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `parent_outputs` property
- `parents` property
- `parents_output_text` property
- `state = field(default=State.PENDING, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structure = field(default=None, kw_only=True)` class-attribute instance-attribute

State

Bases:

Enum

Source Code in griptape/tasks/base_task.py

```
class State(Enum):
    PENDING = 1
    RUNNING = 2
    FINISHED = 3
    SKIPPED = 4
```

- `FINISHED = 3` class-attribute instance-attribute

- `PENDING = 1` class-attribute instance-attribute
- `RUNNING = 2` class-attribute instance-attribute
- `SKIPPED = 4` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/tasks/base_task.py

```
def __attrs_post_init__(self) -> None:
    if self.structure is not None:
        self.structure.add_task(self)
```

lshift(other)

Source Code in griptape/tasks/base_task.py

```
def __lshift__(self, other: BaseTask | list[BaseTask]) -> BaseTask | list[BaseTask]:
    if isinstance(other, list):
        self.add_parents(other)
    else:
        self.add_parent(other)

    return other
```

rshift(other)

Source Code in griptape/tasks/base_task.py

```
def __rshift__(self, other: BaseTask | list[BaseTask]) -> BaseTask | list[BaseTask]:
    if isinstance(other, list):
        self.add_children(other)
    else:
        self.add_child(other)

    return other
```

str()

Source Code in griptape/tasks/base_task.py

```
def __str__(self) -> str:
    return str(self.output.value) if self.output is not None else ""
```

add_child(child)

Source Code in griptape/tasks/base_task.py

```
def add_child(self, child: BaseTask) -> BaseTask:
    if child.id not in self.child_ids:
        self.child_ids.append(child.id)
```

(continues on next page)

(continued from previous page)

```
    if self.id not in child.parent_ids:
        child.parent_ids.append(self.id)

    if self.structure is not None and child not in self.structure.tasks:
        self.structure.add_task(child)

    return self
```

add_children(children)

Source Code in griptape/tasks/base_task.py

```
def add_children(self, children: list[BaseTask]) -> None:
    for child in children:
        self.add_child(child)
```

add_parent(parent)

Source Code in griptape/tasks/base_task.py

```
def add_parent(self, parent: BaseTask) -> BaseTask:
    if parent.id not in self.parent_ids:
        self.parent_ids.append(parent.id)

    if self.id not in parent.child_ids:
        parent.child_ids.append(self.id)

    if self.structure is not None and parent not in self.structure.tasks:
        self.structure.add_task(parent)

    return self
```

add_parents(parents)

Source Code in griptape/tasks/base_task.py

```
def add_parents(self, parents: list[BaseTask]) -> None:
    for parent in parents:
        self.add_parent(parent)
```

after_run()

Source Code in griptape/tasks/base_task.py

```
def after_run(self) -> None:
    super().after_run()
    if self.structure is not None:
        EventBus.publish_event(
            FinishTaskEvent(
                task_id=self.id,
                task_parent_ids=self.parent_ids,
```

(continues on next page)

(continued from previous page)

```
                task_child_ids=self.child_ids,
                task_input=self.input,
                task_output=self.output,
            ),
        )
```

before_run()

Source Code in griptape/tasks/base_task.py

```
def before_run(self) -> None:
    super().before_run()
    if self.structure is not None:
        EventBus.publish_event(
            StartTaskEvent(
                task_id=self.id,
                task_parent_ids=self.parent_ids,
                task_child_ids=self.child_ids,
                task_input=self.input,
                task_output=self.output,
            ),
        )
```

can_run()

Source Code in griptape/tasks/base_task.py

```
def can_run(self) -> bool:
    # If this Task has been skipped or is not pending, it should not run
    if self.is_skipped() or not self.is_pending():
        return False

    # If this Task has parents, and _all_ of them are skipped, it should not run
    if self.parents and all(parent.is_skipped() for parent in self.parents):
        self.state = BaseTask.State.SKIPPED
        return False

    # If _all_ this Tasks unskipped parents are finished, it should run
    unskipped_parents = [parent for parent in self.parents if not parent.is_skipped()]

    return all(parent.is_finished() for parent in unskipped_parents)
```

is_finished()

Source Code in griptape/tasks/base_task.py

```
def is_finished(self) -> bool:
    return self.state == BaseTask.State.FINISHED
```

is_pending()

Source Code in griptape/tasks/base_task.py

```
def is_pending(self) -> bool:
    return self.state == BaseTask.State.PENDING
```

is_running()

Source Code in griptape/tasks/base_task.py

```
def is_running(self) -> bool:
    return self.state == BaseTask.State.RUNNING
```

is_skipped()

Source Code in griptape/tasks/base_task.py

```
def is_skipped(self) -> bool:
    return self.state == BaseTask.State.SKIPPED
```

preprocess(structure)

Source Code in griptape/tasks/base_task.py

```
def preprocess(self, structure: Structure) -> BaseTask:
    self.structure = structure

    return self
```

reset()

Source Code in griptape/tasks/base_task.py

```
def reset(self) -> BaseTask:
    self.state = BaseTask.State.PENDING
    self.output = None
    self._execution_args = ()

    return self
```

run(*args)

Source Code in griptape/tasks/base_task.py

```
def run(self, *args) -> T:
    try:
        self._execution_args = args

        self.state = BaseTask.State.RUNNING

        self.before_run()
```

(continues on next page)

(continued from previous page)

```
        self.output = self.try_run()

        self.after_run()
    except Exception as e:
        logger.exception("%s %s\n%s", self.__class__.__name__, self.id, e)

        self.output = cast("T", ErrorArtifact(str(e), exception=e))
    finally:
        self.state = BaseTask.State.FINISHED

    return self.output
```

try_run()abstractmethod

Source Code in griptape/tasks/base_task.py

```
@abstractmethod
def try_run(self) -> T: ...
```

base_text_input_task

- `T = TypeVar('T', bound=BaseArtifact)` module-attribute
- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *RuleMixin* “) , *BaseTask[T* ”) , ABC

Source Code in griptape/tasks/base_text_input_task.py

```
@define
class BaseTextInputTask(RuleMixin, BaseTask[T], ABC):
    DEFAULT_INPUT_TEMPLATE = "{{ args[0] }}"

    _input: Union[str, TextArtifact, Callable[[BaseTask], TextArtifact]] = field(
        default=DEFAULT_INPUT_TEMPLATE,
        alias="input",
    )

    @property
    def input(self) -> TextArtifact:
        if isinstance(self._input, TextArtifact):
            return self._input
        if isinstance(self._input, Callable):
            return self._input(self)
        return TextArtifact(J2().render_from_string(self._input, **self.full_context))

    @input.setter
    def input(self, value: str | TextArtifact | Callable[[BaseTask], TextArtifact]) ->
↪ None:
        self._input = value

    def before_run(self) -> None:
```

(continues on next page)

(continued from previous page)

```
        super().before_run()

        logger.info("%s %s\nInput: %s", self.__class__.__name__, self.id, self.input.
↪to_text())

    def after_run(self) -> None:
        super().after_run()

        logger.info(
            "%s %s\nOutput: %s",
            self.__class__.__name__,
            self.id,
            self.output.to_text() if self.output is not None else "",
        )
```

- `DEFAULT_INPUT_TEMPLATE = '{{ args[0] }}'` class-attribute instance-attribute
- `_input = field(default=DEFAULT_INPUT_TEMPLATE, alias='input')` class-attribute instance-attribute
- `input` property writable

after_run()

Source Code in griptape/tasks/base_text_input_task.py

```
def after_run(self) -> None:
    super().after_run()

    logger.info(
        "%s %s\nOutput: %s",
        self.__class__.__name__,
        self.id,
        self.output.to_text() if self.output is not None else "",
    )
```

before_run()

Source Code in griptape/tasks/base_text_input_task.py

```
def before_run(self) -> None:
    super().before_run()

    logger.info("%s %s\nInput: %s", self.__class__.__name__, self.id, self.input.to_
↪text())
```

branch_task

Bases: *CodeExecutionTask[Union[InfoArtifact, ListArtifact[InfoArtifact* “)

Source Code in griptape/tasks/branch_task.py

```
@define
class BranchTask(CodeExecutionTask[Union[InfoArtifact, ListArtifact[InfoArtifact]]]):
    on_run: Callable[[BranchTask], Union[InfoArtifact, ListArtifact[InfoArtifact]]] =␣
↪field(kw_only=True)

    def try_run(self) -> InfoArtifact | ListArtifact[InfoArtifact]:
        result = self.on_run(self)

        if isinstance(result, ListArtifact):
            branch_task_ids = {artifact.value for artifact in result}
        else:
            branch_task_ids = {result.value}

        if not all(branch_task_id in self.child_ids for branch_task_id in branch_task_
↪ids):
            raise ValueError(f"Branch task returned invalid child task id {branch_
↪task_ids}")

        if self.structure is not None:
            children_to_skip = [child for child in self.children if child.id not in␣
↪branch_task_ids]
            for child in children_to_skip:
                child.state = BaseTask.State.SKIPPED

        return result
```

- `on_run = field(kw_only=True)` class-attribute instance-attribute

try_run()

Source Code in griptape/tasks/branch_task.py

```
def try_run(self) -> InfoArtifact | ListArtifact[InfoArtifact]:
    result = self.on_run(self)

    if isinstance(result, ListArtifact):
        branch_task_ids = {artifact.value for artifact in result}
    else:
        branch_task_ids = {result.value}

    if not all(branch_task_id in self.child_ids for branch_task_id in branch_task_
↪ids):
        raise ValueError(f"Branch task returned invalid child task id {branch_task_
↪ids}")

    if self.structure is not None:
        children_to_skip = [child for child in self.children if child.id not in␣
↪branch_task_ids]
        for child in children_to_skip:
            child.state = BaseTask.State.SKIPPED

    return result
```

code_execution_task

- `T = TypeVar('T', bound=BaseArtifact)` module-attribute

Bases: *BaseTask[T* “)

Source Code in griptape/tasks/code_execution_task.py

```
@define
class CodeExecutionTask(BaseTask[T]):
    DEFAULT_INPUT_TEMPLATE = "{{ args[0] }}"
    _input: Union[str, TextArtifact, Callable[[BaseTask], TextArtifact]] = field(
        default=DEFAULT_INPUT_TEMPLATE,
        alias="input",
    )
    on_run: Callable[[CodeExecutionTask[T]], T] = field(kw_only=True)

    @property
    def input(self) -> TextArtifact:
        if isinstance(self._input, TextArtifact):
            return self._input
        if callable(self._input):
            return self._input(self)
        return TextArtifact(J2().render_from_string(self._input, **self.full_context))

    def try_run(self) -> T:
        return self.on_run(self)
```

- `DEFAULT_INPUT_TEMPLATE = '{{ args[0] }}'` class-attribute instance-attribute
- `_input = field(default=DEFAULT_INPUT_TEMPLATE, alias='input')` class-attribute instance-attribute
- `input` property
- `on_run = field(kw_only=True)` class-attribute instance-attribute

try_run()

Source Code in griptape/tasks/code_execution_task.py

```
def try_run(self) -> T:
    return self.on_run(self)
```

extraction_task

Bases: *BaseTextInputTask[ListArtifact* “)

Source Code in griptape/tasks/extraction_task.py

```
@define
class ExtractionTask(BaseTextInputTask[ListArtifact]):
    extraction_engine: BaseExtractionEngine = field(kw_only=True)
    args: dict = field(kw_only=True, factory=dict)

    def try_run(self) -> ListArtifact:
```

(continues on next page)

(continued from previous page)

```
        return self.extraction_engine.extract_artifacts(ListArtifact([self.input]),␣
↪rulesets=self.rulesets, **self.args)
```

- `args = field(kw_only=True, factory=dict)` class-attribute instance-attribute
- `extraction_engine = field(kw_only=True)` class-attribute instance-attribute

try_run()

Source Code in griptape/tasks/extraction_task.py

```
def try_run(self) -> ListArtifact:
    return self.extraction_engine.extract_artifacts(ListArtifact([self.input]),␣
↪rulesets=self.rulesets, **self.args)
```

inpainting_image_generation_task

Bases: *BaseImageGenerationTask* “)

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">BaseImageGenerationDrive	The driver used to generate the image.

continues on next page

Table 376 – continued from previous page

Name	Type	Description
<code class="language-python">	<code class="language-python">list[Ruleset]</code>	List of negatively-weighted rulesets applied to the text prompt, if supported by the driver.
<code class="language-python">	<code class="language-python">list[Rule]</code>	List of negatively-weighted rules applied to the text prompt, if supported by the driver.
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.inpainting_image_generation_task.InpaintingImageGenerationTask.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk in output_dir.

continues on next page

Table 376 – continued from previous page

Name	Type	Description
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.inpainting_image_generation_task.InpaintingImageGenerationTask.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk as output_file.

Source Code in griptape/tasks/inpainting_image_generation_task.py

```
@define
class InpaintingImageGenerationTask(BaseImageGenerationTask):
    """A task that modifies a select region within an image using a mask.

    Accepts a text prompt, image, and mask as
    input in one of the following formats:
    - tuple of (template string, ImageArtifact, ImageArtifact)
    - tuple of (TextArtifact, ImageArtifact, ImageArtifact)
    - Callable that returns a tuple of (TextArtifact, ImageArtifact, ImageArtifact).

    Attributes:
        image_generation_driver: The driver used to generate the image.
        negative_rulesets: List of negatively-weighted rulesets applied to the text␣
↪prompt, if supported by the driver.
        negative_rules: List of negatively-weighted rules applied to the text prompt,␣
↪if supported by the driver.
        output_dir: If provided, the generated image will be written to disk in␣
↪output_dir.
        output_file: If provided, the generated image will be written to disk as␣
↪output_file.
    """

    _input: Union[
        tuple[Union[str, TextArtifact], ImageArtifact, ImageArtifact],␣
↪Callable[[BaseTask], ListArtifact], ListArtifact
    ] = field(default=None, alias="input")

    @property
```

(continues on next page)

(continued from previous page)

```
    def input(self) -> ListArtifact:
        if isinstance(self._input, ListArtifact):
            return self._input
        if isinstance(self._input, tuple):
            if isinstance(self._input[0], TextArtifact):
                input_text = self._input[0]
            else:
                input_text = TextArtifact(J2().render_from_string(self._input[0],␣
↪**self.full_context))

            return ListArtifact([input_text, self._input[1], self._input[2]])
        if isinstance(self._input, Callable):
            return self._input(self)
        raise ValueError("Input must be a tuple of (text, image, mask) or a callable␣
↪that returns such a tuple.")

    @input.setter
    def input(
        self,
        value: tuple[str | TextArtifact, ImageArtifact, ImageArtifact] |␣
↪Callable[[BaseTask], ListArtifact],
    ) -> None:
        self._input = value

    def try_run(self) -> ImageArtifact:
        prompt_artifact = self.input[0]

        image_artifact = self.input[1]
        if not isinstance(image_artifact, ImageArtifact):
            raise ValueError("Image must be an ImageArtifact.")

        mask_artifact = self.input[2]
        if not isinstance(mask_artifact, ImageArtifact):
            raise ValueError("Mask must be an ImageArtifact.")

        output_image_artifact = self.image_generation_driver.run_image_inpainting(
            prompts=self._get_prompts(prompt_artifact.to_text()),
            negative_prompts=self._get_negative_prompts(),
            image=image_artifact,
            mask=mask_artifact,
        )

        if self.output_dir or self.output_file:
            self._write_to_file(output_image_artifact)

        return output_image_artifact
```

- `_input = field(default=None, alias='input')` class-attribute instance-attribute
- `input` property writable

try_run()

Source Code in griptape/tasks/inpainting_image_generation_task.py

```
def try_run(self) -> ImageArtifact:
    prompt_artifact = self.input[0]

    image_artifact = self.input[1]
    if not isinstance(image_artifact, ImageArtifact):
        raise ValueError("Image must be an ImageArtifact.")

    mask_artifact = self.input[2]
    if not isinstance(mask_artifact, ImageArtifact):
        raise ValueError("Mask must be an ImageArtifact.")

    output_image_artifact = self.image_generation_driver.run_image_inpainting(
        prompts=self._get_prompts(prompt_artifact.to_text()),
        negative_prompts=self._get_negative_prompts(),
        image=image_artifact,
        mask=mask_artifact,
    )

    if self.output_dir or self.output_file:
        self._write_to_file(output_image_artifact)

    return output_image_artifact
```

outpainting_image_generation_task

Bases: *BaseImageGenerationTask* “)

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">BaseImageGenerationDrive	The engine used to generate the image.
<code class="language-python">	<code class="language-python">list[Ruleset]</code>	List of negatively-weighted rulesets applied to the text prompt, if supported by the driver.
<code class="language-python">	<code class="language-python">list[Rule]</code>	List of negatively-weighted rules applied to the text prompt, if supported by the driver.

continues on next page

Table 377 – continued from previous page

Name	Type	Description
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.outpainting_image_generation_task.OutpaintingImageGenerationTask.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk in output_dir.
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.outpainting_image_generation_task.OutpaintingImageGenerationTask.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk as output_file.

Source Code in griptape/tasks/outpainting_image_generation_task.py

```
@define
class OutpaintingImageGenerationTask(BaseImageGenerationTask):
    """A task that modifies an image outside the bounds of a mask.

    Accepts a text prompt, image, and mask as
    input in one of the following formats:
    - tuple of (template string, ImageArtifact, ImageArtifact)
```

(continues on next page)

(continued from previous page)

```
    - tuple of (TextArtifact, ImageArtifact, ImageArtifact)
    - Callable that returns a tuple of (TextArtifact, ImageArtifact, ImageArtifact).

    Attributes:
        image_generation_driver: The engine used to generate the image.
        negative_rulesets: List of negatively-weighted rulesets applied to the text␣
↪prompt, if supported by the driver.
        negative_rules: List of negatively-weighted rules applied to the text prompt,␣
↪if supported by the driver.
        output_dir: If provided, the generated image will be written to disk in␣
↪output_dir.
        output_file: If provided, the generated image will be written to disk as␣
↪output_file.
    """

    _input: Union[
        tuple[Union[str, TextArtifact], ImageArtifact, ImageArtifact],␣
↪Callable[[BaseTask], ListArtifact], ListArtifact
    ] = field(default=None, alias="input")

    @property
    def input(self) -> ListArtifact:
        if isinstance(self._input, ListArtifact):
            return self._input
        if isinstance(self._input, tuple):
            if isinstance(self._input[0], TextArtifact):
                input_text = self._input[0]
            else:
                input_text = TextArtifact(J2().render_from_string(self._input[0],␣
↪**self.full_context))

            return ListArtifact([input_text, self._input[1], self._input[2]])
        if isinstance(self._input, Callable):
            return self._input(self)
        raise ValueError("Input must be a tuple of (text, image, mask) or a callable␣
↪that returns such a tuple.")

    @input.setter
    def input(
        self,
        value: tuple[str | TextArtifact, ImageArtifact, ImageArtifact] |␣
↪Callable[[BaseTask], ListArtifact],
    ) -> None:
        self._input = value

    def try_run(self) -> ImageArtifact:
        prompt_artifact = self.input[0]

        image_artifact = self.input[1]
        if not isinstance(image_artifact, ImageArtifact):
            raise ValueError("Image must be an ImageArtifact.")
```

(continues on next page)

(continued from previous page)

```
        mask_artifact = self.input[2]
        if not isinstance(mask_artifact, ImageArtifact):
            raise ValueError("Mask must be an ImageArtifact.")

        output_image_artifact = self.image_generation_driver.run_image_outpainting(
            prompts=self._get_prompts(prompt_artifact.to_text()),
            negative_prompts=self._get_negative_prompts(),
            image=image_artifact,
            mask=mask_artifact,
        )

        if self.output_dir or self.output_file:
            self._write_to_file(output_image_artifact)

        return output_image_artifact
```

- `_input = field(default=None, alias='input')` class-attribute instance-attribute
- `input` property writable

try_run()

Source Code in griptape/tasks/outpainting_image_generation_task.py

```
def try_run(self) -> ImageArtifact:
    prompt_artifact = self.input[0]

    image_artifact = self.input[1]
    if not isinstance(image_artifact, ImageArtifact):
        raise ValueError("Image must be an ImageArtifact.")

    mask_artifact = self.input[2]
    if not isinstance(mask_artifact, ImageArtifact):
        raise ValueError("Mask must be an ImageArtifact.")

    output_image_artifact = self.image_generation_driver.run_image_outpainting(
        prompts=self._get_prompts(prompt_artifact.to_text()),
        negative_prompts=self._get_negative_prompts(),
        image=image_artifact,
        mask=mask_artifact,
    )

    if self.output_dir or self.output_file:
        self._write_to_file(output_image_artifact)

    return output_image_artifact
```

output_schema_validation_subtask

- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *BaseSubtask* “)

Source Code in griptape/tasks/output_schema_validation_subtask.py

```
@define
class OutputSchemaValidationSubtask(BaseSubtask):
    _input: BaseArtifact = field(alias="input")
    output_schema: Union[Schema, type[BaseModel]] = field(kw_only=True)
    structured_output_strategy: StructuredOutputStrategy = field(
        default="rule", kw_only=True, metadata={"serializable": True}
    )
    generate_assistant_subtask_template: Callable[[OutputSchemaValidationSubtask],␣
↪str] = field(
        default=Factory(lambda self: self.default_generate_assistant_subtask_template,
↪ takes_self=True),
        kw_only=True,
    )
    generate_user_subtask_template: Callable[[OutputSchemaValidationSubtask], str] =␣
↪field(
        default=Factory(lambda self: self.default_generate_user_subtask_template,␣
↪takes_self=True),
        kw_only=True,
    )
    _validation_errors: str | None = field(default=None, init=False)

    @property
    def input(self) -> BaseArtifact:
        return self._input

    @input.setter
    def input(self, value: BaseArtifact) -> None:
        self._input = value

    @property
    def validation_errors(self) -> str | None:
        return self._validation_errors

    def attach_to(self, parent_task: BaseTask) -> None:
        super().attach_to(parent_task)
        try:
            # With `native` or `rule` strategies, the output will be a json string␣
↪that can be parsed.
            # With the `tool` strategy, the output will already be a `JsonArtifact`.
            if self.structured_output_strategy in ("native", "rule"):
                value_to_validate = (
                    self.input.value if isinstance(self.input.value, str) else json.
↪dumps(self.input.value)
                )
                if isinstance(self.output_schema, Schema):
                    self.output_schema.validate(json.loads(value_to_validate))
                    self.output = JsonArtifact(self.input.value)
                else:
                    model = TypeAdapter(self.output_schema).validate_json(value_to_
↪validate)
                    self.output = ModelArtifact(model)
            else:
                self.output = self.input
```

(continues on next page)

(continued from previous page)

```
        except SchemaError as e:
            self._validation_errors = str(e)
        except ValidationError as e:
            self._validation_errors = str(e.errors())

    def before_run(self) -> None:
        logger.info("%s Validating: %s", self.__class__.__name__, self.input.value)

    def try_run(self) -> BaseArtifact:
        if self._validation_errors is None:
            return self._input
        return ErrorArtifact(
            value=f"Validation error: {self._validation_errors}",
        )

    def after_run(self) -> None:
        if self._validation_errors is None:
            logger.info("%s Validation successful", self.__class__.__name__)
        else:
            logger.error("%s Validation error: %s", self.__class__.__name__, self._
↪validation_errors)

    def add_to_prompt_stack(self, stack: PromptStack) -> None:
        if self.output is None:
            return
        stack.add_assistant_message(self.generate_assistant_subtask_template(self))
        stack.add_user_message(self.generate_user_subtask_template(self))

    def default_generate_assistant_subtask_template(self, subtask:␣
↪OutputSchemaValidationSubtask) -> str:
        return J2("tasks/prompt_task/assistant_output_schema_validation_subtask.j2").
↪render(
            subtask=subtask,
        )

    def default_generate_user_subtask_template(self, subtask:␣
↪OutputSchemaValidationSubtask) -> str:
        return J2("tasks/prompt_task/user_output_schema_validation_subtask.j2").
↪render(
            subtask=subtask,
        )
```

- `_input = field(alias='input')` class-attribute instance-attribute
- `_validation_errors = field(default=None, init=False)` class-attribute instance-attribute
- `generate_assistant_subtask_template = field(default=Factory(lambda self: self.default_generate_assistant_subtask_template, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `generate_user_subtask_template = field(default=Factory(lambda self: self.default_generate_user_subtask_template, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `input` property writable

- `output_schema = field(kw_only=True)` class-attribute instance-attribute
- `structured_output_strategy = field(default='rule', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `validation_errors` property

add_to_prompt_stack(stack)

Source Code in griptape/tasks/output_schema_validation_subtask.py

```
def add_to_prompt_stack(self, stack: PromptStack) -> None:
    if self.output is None:
        return
    stack.add_assistant_message(self.generate_assistant_subtask_template(self))
    stack.add_user_message(self.generate_user_subtask_template(self))
```

after_run()

Source Code in griptape/tasks/output_schema_validation_subtask.py

```
def after_run(self) -> None:
    if self._validation_errors is None:
        logger.info("%s Validation successful", self.__class__.__name__)
    else:
        logger.error("%s Validation error: %s", self.__class__.__name__, self._
↪validation_errors)
```

attach_to(parent_task)

Source Code in griptape/tasks/output_schema_validation_subtask.py

```
def attach_to(self, parent_task: BaseTask) -> None:
    super().attach_to(parent_task)
    try:
        # With `native` or `rule` strategies, the output will be a json string that␣
↪can be parsed.
        # With the `tool` strategy, the output will already be a `JsonArtifact`.
        if self.structured_output_strategy in ("native", "rule"):
            value_to_validate = (
                self.input.value if isinstance(self.input.value, str) else json.
↪dumps(self.input.value)
            )
            if isinstance(self.output_schema, Schema):
                self.output_schema.validate(json.loads(value_to_validate))
                self.output = JsonArtifact(self.input.value)
            else:
                model = TypeAdapter(self.output_schema).validate_json(value_to_
↪validate)
                self.output = ModelArtifact(model)
        else:
            self.output = self.input
    except SchemaError as e:
        self._validation_errors = str(e)
```

(continues on next page)

(continued from previous page)

```
    except ValidationError as e:
        self._validation_errors = str(e.errors())
```

before_run()

Source Code in griptape/tasks/output_schema_validation_subtask.py

```
def before_run(self) -> None:
    logger.info("%s Validating: %s", self.__class__.__name__, self.input.value)
```

default_generate_assistant_subtask_template(subtask)

Source Code in griptape/tasks/output_schema_validation_subtask.py

```
def default_generate_assistant_subtask_template(self, subtask:␣
↪OutputSchemaValidationSubtask) -> str:
    return J2("tasks/prompt_task/assistant_output_schema_validation_subtask.j2").
↪render(
        subtask=subtask,
    )
```

default_generate_user_subtask_template(subtask)

Source Code in griptape/tasks/output_schema_validation_subtask.py

```
def default_generate_user_subtask_template(self, subtask:␣
↪OutputSchemaValidationSubtask) -> str:
    return J2("tasks/prompt_task/user_output_schema_validation_subtask.j2").render(
        subtask=subtask,
    )
```

try_run()

Source Code in griptape/tasks/output_schema_validation_subtask.py

```
def try_run(self) -> BaseArtifact:
    if self._validation_errors is None:
        return self._input
    return ErrorArtifact(
        value=f"Validation error: {self._validation_errors}",
    )
```

prompt_image_generation_task

Bases: *BaseImageGenerationTask* “)

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">BaseImageGenerationDrive	The engine used to generate the image.
<code class="language-python">	<code class="language-python">list[Ruleset]</code>	List of negatively-weighted rulesets applied to the text prompt, if supported by the driver.
<code class="language-python">	<code class="language-python">list[Rule]</code>	List of negatively-weighted rules applied to the text prompt, if supported by the driver.

continues on next page

Table 378 – continued from previous page

Name	Type	Description
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.prompt_image_generation_task.PromptImageGenerationTask.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk in output_dir.
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.prompt_image_generation_task.PromptImageGenerationTask.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk as output_file.

Source Code in griptape/tasks/prompt_image_generation_task.py

```
@define
class PromptImageGenerationTask(BaseImageGenerationTask):
    """Used to generate an image from a text prompt.

    Accepts prompt as input in one of the following formats:
    - template string
    - TextArtifact
    - Callable that returns a TextArtifact.
```

(continues on next page)

(continued from previous page)

```
    Attributes:
        image_generation_driver: The engine used to generate the image.
        negative_rulesets: List of negatively-weighted rulesets applied to the text␣
↪prompt, if supported by the driver.
        negative_rules: List of negatively-weighted rules applied to the text prompt,␣
↪if supported by the driver.
        output_dir: If provided, the generated image will be written to disk in␣
↪output_dir.
        output_file: If provided, the generated image will be written to disk as␣
↪output_file.
    """

    DEFAULT_INPUT_TEMPLATE = "{{ args[0] }}"

    _input: Union[str, TextArtifact, Callable[[BaseTask], TextArtifact]] = field(
        default=DEFAULT_INPUT_TEMPLATE, alias="input"
    )

    @property
    def input(self) -> TextArtifact:
        if isinstance(self._input, TextArtifact):
            return self._input
        if isinstance(self._input, Callable):
            return self._input(self)
        return TextArtifact(J2().render_from_string(self._input, **self.full_context))

    @input.setter
    def input(self, value: TextArtifact) -> None:
        self._input = value

    def try_run(self) -> ImageArtifact:
        image_artifact = self.image_generation_driver.run_text_to_image(
            prompts=self._get_prompts(self.input.to_text()),
            negative_prompts=self._get_negative_prompts(),
        )

        if self.output_dir or self.output_file:
            self._write_to_file(image_artifact)

        return image_artifact
```

- `DEFAULT_INPUT_TEMPLATE = '{{ args[0] }}'` class-attribute instance-attribute
- `_input = field(default=DEFAULT_INPUT_TEMPLATE, alias='input')` class-attribute instance-attribute
- `input` property writable

try_run()

Source Code in griptape/tasks/prompt_image_generation_task.py

```
def try_run(self) -> ImageArtifact:
    image_artifact = self.image_generation_driver.run_text_to_image(
        prompts=self._get_prompts(self.input.to_text()),
        negative_prompts=self._get_negative_prompts(),
    )

    if self.output_dir or self.output_file:
        self._write_to_file(image_artifact)

    return image_artifact
```

prompt_task

- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *BaseTask[Union[TextArtifact, AudioArtifact, GenericArtifact, JsonArtifact, ListArtifact, ErrorArtifact "), RuleMixin "), ActionsSubtaskOriginMixin ")*

Source Code in griptape/tasks/prompt_task.py

```
@define
class PromptTask(
    BaseTask[Union[TextArtifact, AudioArtifact, GenericArtifact, JsonArtifact,␣
↪ListArtifact, ErrorArtifact]],
    RuleMixin,
    ActionsSubtaskOriginMixin,
):
    DEFAULT_MAX_STEPS = 20
    # Stop sequence for chain-of-thought in the framework. Using this "token-like"␣
↪string to make it more unique,
    # so that it doesnt trigger on accident.
    RESPONSE_STOP_SEQUENCE = "<|Response|>"
    prompt_driver: BasePromptDriver = field(
        default=Factory(lambda: Defaults.drivers_config.prompt_driver), kw_only=True,␣
↪metadata={"serializable": True}
    )
    output_schema: Optional[Union[Schema, type[BaseModel]]] = field(default=None, kw_
↪only=True)
    generate_system_template: Callable[[PromptTask], str] = field(
        default=Factory(lambda self: self.default_generate_system_template, takes_
↪self=True),
        kw_only=True,
    )
    _conversation_memory: Union[Optional[BaseConversationMemory], NothingType] =␣
↪field(
        default=Factory(lambda: NOTHING), kw_only=True, alias="conversation_memory"
    )
    _input: Union[str, list, tuple, BaseArtifact, Callable[[BaseTask], BaseArtifact]]␣
↪= field(
        default=lambda task: task.full_context["args"][0] if task.full_context["args
↪"] else TextArtifact(value=""),
        alias="input",
    )
```

(continues on next page)

(continued from previous page)

```
    tools: list[BaseTool] = field(factory=list, kw_only=True, metadata={"serializable
↪": True})
    max_subtasks: int = field(default=DEFAULT_MAX_STEPS, kw_only=True, metadata={
↪"serializable": True})
    task_memory: Optional[TaskMemory] = field(default=None, kw_only=True)
    subtasks: list[BaseSubtask] = field(factory=list)
    generate_assistant_subtask_template: Callable[[ActionsSubtask], str] = field(
        default=Factory(lambda self: self.default_generate_assistant_subtask_template,
↪ takes_self=True),
        kw_only=True,
    )
    generate_user_subtask_template: Callable[[ActionsSubtask], str] = field(
        default=Factory(lambda self: self.default_generate_user_subtask_template,␣
↪takes_self=True),
        kw_only=True,
    )
    response_stop_sequence: str = field(default=RESPONSE_STOP_SEQUENCE, kw_only=True)
    reflect_on_tool_use: bool = field(default=True, kw_only=True)
    subtask_runners: list[Callable[[BaseArtifact], BaseArtifact]] = field(
        default=Factory(
            lambda self: [self.default_run_actions_subtasks, self.default_run_output_
↪schema_validation_subtasks],
            takes_self=True,
        ),
        kw_only=True,
    )

    @property
    def rulesets(self) -> list:
        default_rules = self.rules
        rulesets = self._rulesets.copy()

        if self.structure is not None:
            if self.structure._rulesets:
                rulesets = self.structure._rulesets + self._rulesets
            if self.structure.rules:
                default_rules = self.structure.rules + self.rules

        if default_rules:
            rulesets.append(Ruleset(name=self.DEFAULT_RULESET_NAME, rules=default_
↪rules))

        return rulesets

    @property
    def input(self) -> BaseArtifact:
        return self._process_task_input(self._input)

    @input.setter
    def input(self, value: str | list | tuple | BaseArtifact | Callable[[BaseTask],␣
↪BaseArtifact]) -> None:
        self._input = value
```

(continues on next page)

(continued from previous page)

```
    @property
    def conversation_memory(self) -> Optional[BaseConversationMemory]:
        if self._conversation_memory is NOTHING:
            if self.structure is None:
                return None
            return self.structure.conversation_memory
        return self._conversation_memory

    @conversation_memory.setter
    def conversation_memory(self, value: Optional[BaseConversationMemory]) -> None:
        self._conversation_memory = value

    @property
    def prompt_stack(self) -> PromptStack:
        stack = PromptStack(tools=self.tools, output_schema=self.output_schema)
        memory = self.conversation_memory

        system_template = self.generate_system_template(self)
        if system_template:
            stack.add_system_message(system_template)

        stack.add_user_message(self.input)

        if self.output:
            stack.add_assistant_message(self.output.to_text())
        else:
            for s in self.subtasks:
                s.add_to_prompt_stack(stack)

        if memory is not None:
            # inserting at index 1 to place memory right after system prompt
            memory.add_to_prompt_stack(self.prompt_driver, stack, 1 if system_
↪template else 0)

        return stack

    @property
    def tool_output_memory(self) -> list[TaskMemory]:
        unique_memory_dict = {}

        for memories in [tool.output_memory for tool in self.tools if tool.output_
↪memory]:
            for memory_list in memories.values():
                for memory in memory_list:
                    if memory.name not in unique_memory_dict:
                        unique_memory_dict[memory.name] = memory

        return list(unique_memory_dict.values())

    @tools.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_tools(self, _: Attribute, tools: list[BaseTool]) -> None:
```

(continues on next page)

(continued from previous page)

```
        tool_names = [t.name for t in tools]

        if len(tool_names) > len(set(tool_names)):
            raise ValueError("tools names have to be unique in task")

    @output_schema.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_output_schema(self, _: Attribute, output_schema:␣
↪Optional[Union[Schema, type[BaseModel]]]) -> None:
        if (
            output_schema is None
            or isinstance(self.output_schema, Schema)
            or (isinstance(self.output_schema, type) and issubclass(self.output_
↪schema, BaseModel))
        ):
            return
        raise ValueError(f"Unsupported output schema type: {type(self.output_schema)}
↪")

    def __attrs_post_init__(self) -> None:
        super().__attrs_post_init__()
        if self.task_memory:
            self.set_default_tools_memory(self.task_memory)

    output: Optional[BaseArtifact] = field(default=None, init=False)

    def before_run(self) -> None:
        super().before_run()

        logger.info("%s %s\nInput: %s", self.__class__.__name__, self.id, self.input.
↪to_text())

    def after_run(self) -> None:
        super().after_run()

        logger.info(
            "%s %s\nOutput: %s",
            self.__class__.__name__,
            self.id,
            self.output.to_text() if self.output is not None else "",
        )
        conversation_memory = self.conversation_memory
        if (
            (self.structure is None or self.structure.conversation_memory_strategy ==
↪"per_task")
            and conversation_memory is not None
            and self.output is not None
        ):
            run = Run(input=self.input, output=self.output)

            conversation_memory.add_run(run)
```

(continues on next page)

(continued from previous page)

```
    def try_run(self) -> ListArtifact | TextArtifact | AudioArtifact |␣
↪GenericArtifact | JsonArtifact | ErrorArtifact:
        self.subtasks.clear()
        if self.response_stop_sequence not in self.prompt_driver.tokenizer.stop_
↪sequences:
            self.prompt_driver.tokenizer.stop_sequences.extend([self.response_stop_
↪sequence])

        output = self.prompt_driver.run(self.prompt_stack).to_artifact(
            meta={"is_react_prompt": not self.prompt_driver.use_native_tools}
        )
        for subtask_runner in self.subtask_runners:
            output = subtask_runner(output)

        if isinstance(output, (ListArtifact, TextArtifact, AudioArtifact,␣
↪JsonArtifact, ModelArtifact, ErrorArtifact)):
            return output
        raise ValueError(f"Unsupported output type: {type(output)}")

    def preprocess(self, structure: Structure) -> BaseTask:
        super().preprocess(structure)

        if self.task_memory is None and structure.task_memory:
            self.set_default_tools_memory(structure.task_memory)

        return self

    def default_generate_system_template(self, _: PromptTask) -> str:
        schema = self.actions_schema().json_schema("Actions Schema")
        schema["minItems"] = 1  # The `schema` library doesnt support `minItems` so␣
↪we must add it manually.

        return J2("tasks/prompt_task/system.j2").render(
            rulesets=J2("rulesets/rulesets.j2").render(rulesets=self.rulesets),
            action_names=str.join(", ", [tool.name for tool in self.tools]),
            actions_schema=utils.minify_json(json.dumps(schema)),
            meta_memory=J2("memory/meta/meta_memory.j2").render(meta_memories=self.
↪meta_memories),
            use_native_tools=self.prompt_driver.use_native_tools,
            stop_sequence=self.response_stop_sequence,
            reflect_on_tool_use=self.reflect_on_tool_use,
        )

    def default_generate_assistant_subtask_template(self, subtask: ActionsSubtask) ->␣
↪str:
        return J2("tasks/prompt_task/assistant_actions_subtask.j2").render(
            stop_sequence=self.response_stop_sequence,
            subtask=subtask,
        )

    def default_generate_user_subtask_template(self, subtask: ActionsSubtask) -> str:
        return J2("tasks/prompt_task/user_actions_subtask.j2").render(
```

(continues on next page)

(continued from previous page)

```
            stop_sequence=self.response_stop_sequence,
            subtask=subtask,
        )

    def actions_schema(self) -> Schema:
        return self._actions_schema_for_tools(self.tools)

    def set_default_tools_memory(self, memory: TaskMemory) -> None:
        self.task_memory = memory

        for tool in self.tools:
            if self.task_memory:
                if tool.input_memory is None:
                    tool.input_memory = [self.task_memory]
                if tool.output_memory is None and tool.off_prompt:
                    tool.output_memory = {getattr(a, "name"): [self.task_memory] for␣
↪a in tool.activities()}

    def find_subtask(self, subtask_id: str) -> BaseSubtask:
        for subtask in self.subtasks:
            if subtask.id == subtask_id:
                return subtask
        raise ValueError(f"Subtask with id {subtask_id} not found.")

    def add_subtask(self, subtask: BaseSubtask) -> BaseSubtask:
        subtask.attach_to(self)
        subtask.structure = self.structure

        if len(self.subtasks) > 0:
            self.subtasks[-1].add_child(subtask)
            subtask.add_parent(self.subtasks[-1])

        self.subtasks.append(subtask)

        return subtask

    def find_tool(self, tool_name: str) -> BaseTool:
        for tool in self.tools:
            if tool.name == tool_name:
                return tool
        raise ValueError(f"Tool with name {tool_name} not found.")

    def find_memory(self, memory_name: str) -> TaskMemory:
        for memory in self.tool_output_memory:
            if memory.name == memory_name:
                return memory
        raise ValueError(f"Memory with name {memory_name} not found.")

    def default_run_actions_subtasks(self, subtask_input: BaseArtifact) ->␣
↪BaseArtifact:
        if not self.tools:
            return subtask_input
```

(continues on next page)

(continued from previous page)

```
        subtask = self.add_subtask(
            ActionsSubtask(
                subtask_input,
                # TODO: Remove these fields in Prompt Task in Gen AI Builder 2.0
                generate_user_subtask_template=self.generate_user_subtask_template,
                generate_assistant_subtask_template=self.generate_assistant_subtask_
↪template,
                response_stop_sequence=self.response_stop_sequence,
            )
        )

        while subtask.output is None:
            if len(self.subtasks) >= self.max_subtasks:
                subtask.output = ErrorArtifact(f"Exceeded tool limit of {self.max_
↪subtasks} subtasks per task")
            else:
                subtask.run()

                if self.reflect_on_tool_use:
                    output = self.prompt_driver.run(self.prompt_stack).to_artifact(
                        meta={"is_react_prompt": not self.prompt_driver.use_native_
↪tools}
                    )
                    subtask = self.add_subtask(ActionsSubtask(output))

        return subtask.output

    def default_run_output_schema_validation_subtasks(self, subtask_input:␣
↪BaseArtifact) -> BaseArtifact:
        if self.output_schema is None:
            return subtask_input
        subtask = self.add_subtask(OutputSchemaValidationSubtask(subtask_input,␣
↪output_schema=self.output_schema))

        while subtask.output is None:
            if len(self.subtasks) >= self.max_subtasks:
                subtask.output = ErrorArtifact(f"Exceeded tool limit of {self.max_
↪subtasks} subtasks per task")
            else:
                subtask.run()

                output = subtask.output
                output = self.prompt_driver.run(self.prompt_stack).to_artifact(
                    meta={"is_react_prompt": not self.prompt_driver.use_native_tools}
                )
                subtask = self.add_subtask(OutputSchemaValidationSubtask(output,␣
↪output_schema=self.output_schema))

        return subtask.output

    def _process_task_input(
        self,
```

(continues on next page)

(continued from previous page)

```
        task_input: str | tuple | list | BaseArtifact | Callable[[BaseTask],␣
↪BaseArtifact],
    ) -> BaseArtifact:
        if isinstance(task_input, TextArtifact):
            return TextArtifact(J2().render_from_string(task_input.value, **self.full_
↪context), meta=task_input.meta)
        if isinstance(task_input, Callable):
            return self._process_task_input(task_input(self))
        if isinstance(task_input, ListArtifact):
            return ListArtifact([self._process_task_input(elem) for elem in task_
↪input.value])
        if isinstance(task_input, BaseArtifact):
            return task_input
        if isinstance(task_input, (list, tuple)):
            return ListArtifact([self._process_task_input(elem) for elem in task_
↪input])
        return self._process_task_input(TextArtifact(task_input))
```

- `DEFAULT_MAX_STEPS = 20` class-attribute instance-attribute
- `RESPONSE_STOP_SEQUENCE = '<|Response|>'` class-attribute instance-attribute
- `_conversation_memory = field(default=Factory(lambda: NOTHING), kw_only=True, alias='conversation_memory')` class-attribute instance-attribute
- `_input = field(default=lambda task: task.full_context['args'][0] if task.full_context['args'] else TextArtifact(value=''), alias='input')` class-attribute instance-attribute
- `conversation_memory` property writable
- `generate_assistant_subtask_template = field(default=Factory(lambda self: self.default_generate_assistant_subtask_template, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `generate_system_template = field(default=Factory(lambda self: self.default_generate_system_template, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `generate_user_subtask_template = field(default=Factory(lambda self: self.default_generate_user_subtask_template, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `input` property writable
- `max_subtasks = field(default=DEFAULT_MAX_STEPS, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output = field(default=None, init=False)` class-attribute instance-attribute
- `output_schema = field(default=None, kw_only=True)` class-attribute instance-attribute
- `prompt_driver = field(default=Factory(lambda: Defaults.drivers_config.prompt_driver), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `prompt_stack` property
- `reflect_on_tool_use = field(default=True, kw_only=True)` class-attribute instance-attribute

- `response_stop_sequence = field(default=RESPONSE_STOP_SEQUENCE, kw_only=True)` class-attribute instance-attribute
- `rulesets` property
- `subtask_runners = field(default=Factory(lambda self: [self.default_run_actions_subtasks, self.default_run_output_schema_validation_subtasks], takes_self=True), kw_only=True)` class-attribute instance-attribute
- `subtasks = field(factory=list)` class-attribute instance-attribute
- `task_memory = field(default=None, kw_only=True)` class-attribute instance-attribute
- `tool_output_memory` property
- `tools = field(factory=list, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/tasks/prompt_task.py

```
def __attrs_post_init__(self) -> None:
    super().__attrs_post_init__()
    if self.task_memory:
        self.set_default_tools_memory(self.task_memory)
```

_process_task_input(task_input)

Source Code in griptape/tasks/prompt_task.py

```
def _process_task_input(
    self,
    task_input: str | tuple | list | BaseArtifact | Callable[[BaseTask],␣
↪BaseArtifact],
) -> BaseArtifact:
    if isinstance(task_input, TextArtifact):
        return TextArtifact(J2().render_from_string(task_input.value, **self.full_
↪context), meta=task_input.meta)
    if isinstance(task_input, Callable):
        return self._process_task_input(task_input(self))
    if isinstance(task_input, ListArtifact):
        return ListArtifact([self._process_task_input(elem) for elem in task_input.
↪value])
    if isinstance(task_input, BaseArtifact):
        return task_input
    if isinstance(task_input, (list, tuple)):
        return ListArtifact([self._process_task_input(elem) for elem in task_input])
    return self._process_task_input(TextArtifact(task_input))
```

actions_schema()

Source Code in griptape/tasks/prompt_task.py

```
def actions_schema(self) -> Schema:
    return self._actions_schema_for_tools(self.tools)
```

add_subtask(subtask)

Source Code in griptape/tasks/prompt_task.py

```
def add_subtask(self, subtask: BaseSubtask) -> BaseSubtask:
    subtask.attach_to(self)
    subtask.structure = self.structure

    if len(self.subtasks) > 0:
        self.subtasks[-1].add_child(subtask)
        subtask.add_parent(self.subtasks[-1])

    self.subtasks.append(subtask)

    return subtask
```

after_run()

Source Code in griptape/tasks/prompt_task.py

```
def after_run(self) -> None:
    super().after_run()

    logger.info(
        "%s %s\nOutput: %s",
        self.__class__.__name__,
        self.id,
        self.output.to_text() if self.output is not None else "",
    )
    conversation_memory = self.conversation_memory
    if (
        (self.structure is None or self.structure.conversation_memory_strategy ==
↪"per_task")
        and conversation_memory is not None
        and self.output is not None
    ):
        run = Run(input=self.input, output=self.output)

        conversation_memory.add_run(run)
```

before_run()

Source Code in griptape/tasks/prompt_task.py

```
def before_run(self) -> None:
    super().before_run()

    logger.info("%s %s\nInput: %s", self.__class__.__name__, self.id, self.input.to_
↪text())
```

default_generate_assistant_subtask_template(subtask)

Source Code in griptape/tasks/prompt_task.py

```
def default_generate_assistant_subtask_template(self, subtask: ActionsSubtask) -> str:
    return J2("tasks/prompt_task/assistant_actions_subtask.j2").render(
        stop_sequence=self.response_stop_sequence,
        subtask=subtask,
    )
```

default*generate_system_template()*

Source Code in griptape/tasks/prompt_task.py

```
def default_generate_system_template(self, _: PromptTask) -> str:
    schema = self.actions_schema().json_schema("Actions Schema")
    schema["minItems"] = 1  # The `schema` library doesnt support `minItems` so we
↪must add it manually.

    return J2("tasks/prompt_task/system.j2").render(
        rulesets=J2("rulesets/rulesets.j2").render(rulesets=self.rulesets),
        action_names=str.join(", ", [tool.name for tool in self.tools]),
        actions_schema=utils.minify_json(json.dumps(schema)),
        meta_memory=J2("memory/meta/meta_memory.j2").render(meta_memories=self.meta_
↪memories),
        use_native_tools=self.prompt_driver.use_native_tools,
        stop_sequence=self.response_stop_sequence,
        reflect_on_tool_use=self.reflect_on_tool_use,
    )
```

default_generate_user_subtask_template(subtask)

Source Code in griptape/tasks/prompt_task.py

```
def default_generate_user_subtask_template(self, subtask: ActionsSubtask) -> str:
    return J2("tasks/prompt_task/user_actions_subtask.j2").render(
        stop_sequence=self.response_stop_sequence,
        subtask=subtask,
    )
```

default_run_actions_subtasks(subtask_input)

Source Code in griptape/tasks/prompt_task.py

```
def default_run_actions_subtasks(self, subtask_input: BaseArtifact) -> BaseArtifact:
    if not self.tools:
        return subtask_input
    subtask = self.add_subtask(
        ActionsSubtask(
            subtask_input,
            # TODO: Remove these fields in Prompt Task in Gen AI Builder 2.0
            generate_user_subtask_template=self.generate_user_subtask_template,
            generate_assistant_subtask_template=self.generate_assistant_subtask_
```

(continues on next page)

(continued from previous page)

```
↪template,
            response_stop_sequence=self.response_stop_sequence,
        )
    )

    while subtask.output is None:
        if len(self.subtasks) >= self.max_subtasks:
            subtask.output = ErrorArtifact(f"Exceeded tool limit of {self.max_
↪subtasks} subtasks per task")
        else:
            subtask.run()

            if self.reflect_on_tool_use:
                output = self.prompt_driver.run(self.prompt_stack).to_artifact(
                    meta={"is_react_prompt": not self.prompt_driver.use_native_tools}
                )
                subtask = self.add_subtask(ActionsSubtask(output))

    return subtask.output
```

default_run_output_schema_validation_subtasks(subtask_input)

Source Code in griptape/tasks/prompt_task.py

```
def default_run_output_schema_validation_subtasks(self, subtask_input: BaseArtifact) -
↪> BaseArtifact:
    if self.output_schema is None:
        return subtask_input
    subtask = self.add_subtask(OutputSchemaValidationSubtask(subtask_input, output_
↪schema=self.output_schema))

    while subtask.output is None:
        if len(self.subtasks) >= self.max_subtasks:
            subtask.output = ErrorArtifact(f"Exceeded tool limit of {self.max_
↪subtasks} subtasks per task")
        else:
            subtask.run()

            output = subtask.output
            output = self.prompt_driver.run(self.prompt_stack).to_artifact(
                meta={"is_react_prompt": not self.prompt_driver.use_native_tools}
            )
            subtask = self.add_subtask(OutputSchemaValidationSubtask(output, output_
↪schema=self.output_schema))

    return subtask.output
```

find_memory(memory_name)

Source Code in griptape/tasks/prompt_task.py

```
def find_memory(self, memory_name: str) -> TaskMemory:
    for memory in self.tool_output_memory:
        if memory.name == memory_name:
            return memory
    raise ValueError(f"Memory with name {memory_name} not found.")
```

find_subtask(subtask_id)

Source Code in griptape/tasks/prompt_task.py

```
def find_subtask(self, subtask_id: str) -> BaseSubtask:
    for subtask in self.subtasks:
        if subtask.id == subtask_id:
            return subtask
    raise ValueError(f"Subtask with id {subtask_id} not found.")
```

find_tool(tool_name)

Source Code in griptape/tasks/prompt_task.py

```
def find_tool(self, tool_name: str) -> BaseTool:
    for tool in self.tools:
        if tool.name == tool_name:
            return tool
    raise ValueError(f"Tool with name {tool_name} not found.")
```

preprocess(structure)

Source Code in griptape/tasks/prompt_task.py

```
def preprocess(self, structure: Structure) -> BaseTask:
    super().preprocess(structure)

    if self.task_memory is None and structure.task_memory:
        self.set_default_tools_memory(structure.task_memory)

    return self
```

set_default_tools_memory(memory)

Source Code in griptape/tasks/prompt_task.py

```
def set_default_tools_memory(self, memory: TaskMemory) -> None:
    self.task_memory = memory

    for tool in self.tools:
        if self.task_memory:
            if tool.input_memory is None:
```

(continues on next page)

(continued from previous page)

```
                tool.input_memory = [self.task_memory]
            if tool.output_memory is None and tool.off_prompt:
                tool.output_memory = {getattr(a, "name"): [self.task_memory] for a in␣
↪tool.activities()}
```

try_run()

Source Code in griptape/tasks/prompt_task.py

```
def try_run(self) -> ListArtifact | TextArtifact | AudioArtifact | GenericArtifact |␣
↪JsonArtifact | ErrorArtifact:
    self.subtasks.clear()
    if self.response_stop_sequence not in self.prompt_driver.tokenizer.stop_sequences:
        self.prompt_driver.tokenizer.stop_sequences.extend([self.response_stop_
↪sequence])

    output = self.prompt_driver.run(self.prompt_stack).to_artifact(
        meta={"is_react_prompt": not self.prompt_driver.use_native_tools}
    )
    for subtask_runner in self.subtask_runners:
        output = subtask_runner(output)

    if isinstance(output, (ListArtifact, TextArtifact, AudioArtifact, JsonArtifact,␣
↪ModelArtifact, ErrorArtifact)):
        return output
    raise ValueError(f"Unsupported output type: {type(output)}")
```

validate*output_schema(*, output_schema)

Source Code in griptape/tasks/prompt_task.py

```
@output_schema.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_output_schema(self, _: Attribute, output_schema: Optional[Union[Schema,␣
↪type[BaseModel]]]) -> None:
    if (
        output_schema is None
        or isinstance(self.output_schema, Schema)
        or (isinstance(self.output_schema, type) and issubclass(self.output_schema,␣
↪BaseModel))
    ):
        return
    raise ValueError(f"Unsupported output schema type: {type(self.output_schema)}")
```

validate*tools(*, tools)

Source Code in griptape/tasks/prompt_task.py

```
@tools.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_tools(self, _: Attribute, tools: list[BaseTool]) -> None:
    tool_names = [t.name for t in tools]
```

(continues on next page)

(continued from previous page)

```
    if len(tool_names) > len(set(tool_names)):
        raise ValueError("tools names have to be unique in task")
```

rag_task

Bases: *BaseTextInputTask[Union[ListArtifact, ErrorArtifact* “)

Source Code in griptape/tasks/rag_task.py

```
@define
class RagTask(BaseTextInputTask[Union[ListArtifact, ErrorArtifact]]):
    rag_engine: RagEngine = field(kw_only=True, default=Factory(lambda: RagEngine()))

    def try_run(self) -> ListArtifact | ErrorArtifact:
        outputs = self.rag_engine.process_query(self.input.to_text()).outputs

        if len(outputs) > 0:
            return ListArtifact(outputs)
        return ErrorArtifact("empty output")
```

- `rag_engine = field(kw_only=True, default=Factory(lambda: RagEngine()))` class-attribute instance-attribute

try_run()

Source Code in griptape/tasks/rag_task.py

```
def try_run(self) -> ListArtifact | ErrorArtifact:
    outputs = self.rag_engine.process_query(self.input.to_text()).outputs

    if len(outputs) > 0:
        return ListArtifact(outputs)
    return ErrorArtifact("empty output")
```

structure_run_task

Bases: *BaseTaskEvent* “)

Attributes

Name	Type	Description
<a href="structure_run_task.mdx#griptape.tasks.structure_run_task.StructureRunTask.structure_run_driver" title="structure_run_driver = field(kw_only=True) class-attribute instance-attribute (griptape.tasks.structure_run_task.StructureRunTask.structure_run_driver)"><code class="language-python">	<a href="../drivers/structure_run/base_structure_run_driver.mdx#griptape.drivers.structure_run.base_structure_run_driver.BaseStructureRunDriver" title="BaseStructureRunDriver (griptape.drivers.structure_run.base_structure_run_driver.BaseStructureRunDriver)"><code class="language-python">BaseStructureRunDriver</c	Driver to run the Structure.

Source Code in griptape/tasks/structure_run_task.py

```
@define
class StructureRunTask(BaseTask):
    """Task to run a Structure.

    Attributes:
        structure_run_driver: Driver to run the Structure.
    """

    _input: Union[str, list, tuple, BaseArtifact, Callable[[BaseTask], BaseArtifact]]␣
↪= field(
        default=lambda task: task.full_context["args"][0] if task.full_context["args
↪"] else TextArtifact(value=""),
    )

    @property
    def input(self) -> BaseArtifact:
        return self._process_task_input(self._input)

    @input.setter
    def input(self, value: str | list | tuple | BaseArtifact | Callable[[BaseTask],␣
↪BaseArtifact]) -> None:
        self._input = value

    structure_run_driver: BaseStructureRunDriver = field(kw_only=True)

    def try_run(self) -> BaseArtifact:
        if isinstance(self.input, ListArtifact):
            return self.structure_run_driver.run(*self.input.value)
        return self.structure_run_driver.run(self.input)

    def _process_task_input(
        self,
```

(continues on next page)

(continued from previous page)

```
        task_input: str | tuple | list | BaseArtifact | Callable[[BaseTask],␣
↪BaseArtifact],
    ) -> BaseArtifact:
        if isinstance(task_input, TextArtifact):
            task_input.value = J2().render_from_string(task_input.value, **self.full_
↪context)

            return task_input
        if isinstance(task_input, Callable):
            return self._process_task_input(task_input(self))
        if isinstance(task_input, ListArtifact):
            return ListArtifact([self._process_task_input(elem) for elem in task_
↪input.value])
        if isinstance(task_input, BaseArtifact):
            return task_input
        if isinstance(task_input, (list, tuple)):
            return ListArtifact([self._process_task_input(elem) for elem in task_
↪input])
        return self._process_task_input(TextArtifact(task_input))
```

- `_input = field(default=lambda task: task.full_context['args'][0] if task.full_context['args'] else TextArtifact(value=''))` class-attribute instance-attribute
- `input` property writable
- `structure_run_driver = field(kw_only=True)` class-attribute instance-attribute

_process_task_input(task_input)

Source Code in griptape/tasks/structure_run_task.py

```
def _process_task_input(
    self,
    task_input: str | tuple | list | BaseArtifact | Callable[[BaseTask],␣
↪BaseArtifact],
) -> BaseArtifact:
    if isinstance(task_input, TextArtifact):
        task_input.value = J2().render_from_string(task_input.value, **self.full_
↪context)

        return task_input
    if isinstance(task_input, Callable):
        return self._process_task_input(task_input(self))
    if isinstance(task_input, ListArtifact):
        return ListArtifact([self._process_task_input(elem) for elem in task_input.
↪value])
    if isinstance(task_input, BaseArtifact):
        return task_input
    if isinstance(task_input, (list, tuple)):
        return ListArtifact([self._process_task_input(elem) for elem in task_input])
    return self._process_task_input(TextArtifact(task_input))
```

try_run()

Source Code in griptape/tasks/structure_run_task.py

```
def try_run(self) -> BaseArtifact:
    if isinstance(self.input, ListArtifact):
        return self.structure_run_driver.run(*self.input.value)
    return self.structure_run_driver.run(self.input)
```

text_summary_task

Bases: *BaseTextInputTask[TextArtifact* “)

Source Code in griptape/tasks/text_summary_task.py

```
@define
class TextSummaryTask(BaseTextInputTask[TextArtifact]):
    summary_engine: BaseSummaryEngine = field(default=Factory(lambda:␣
↪PromptSummaryEngine()), kw_only=True)

    def try_run(self) -> TextArtifact:
        return TextArtifact(self.summary_engine.summarize_text(self.input.to_text(),␣
↪rulesets=self.rulesets))
```

- `summary_engine = field(default=Factory(lambda: PromptSummaryEngine()), kw_only=True)` class-attribute instance-attribute

try_run()

Source Code in griptape/tasks/text_summary_task.py

```
def try_run(self) -> TextArtifact:
    return TextArtifact(self.summary_engine.summarize_text(self.input.to_text(),␣
↪rulesets=self.rulesets))
```

text_to_speech_task

Bases: *BaseAudioGenerationTask* “)

Source Code in griptape/tasks/text_to_speech_task.py

```
@define
class TextToSpeechTask(BaseAudioGenerationTask):
    DEFAULT_INPUT_TEMPLATE = "{{ args[0] }}"

    _input: Union[str, TextArtifact, Callable[[BaseTask], TextArtifact]] =␣
↪field(default=DEFAULT_INPUT_TEMPLATE)
    text_to_speech_driver: BaseTextToSpeechDriver = field(
        default=Factory(lambda: Defaults.drivers_config.text_to_speech_driver), kw_
↪only=True
    )

    @property
    def input(self) -> TextArtifact:
```

(continues on next page)

(continued from previous page)

```
        if isinstance(self._input, TextArtifact):
            return self._input
        if isinstance(self._input, Callable):
            return self._input(self)
        return TextArtifact(J2().render_from_string(self._input, **self.full_context))

    @input.setter
    def input(self, value: TextArtifact) -> None:
        self._input = value

    def try_run(self) -> AudioArtifact:
        audio_artifact = self.text_to_speech_driver.run_text_to_audio(prompts=[self.
↪input.to_text()])

        if self.output_dir or self.output_file:
            self._write_to_file(audio_artifact)

        return audio_artifact
```

- `DEFAULT_INPUT_TEMPLATE = '{{ args[0] }}'` class-attribute instance-attribute
- `_input = field(default=DEFAULT_INPUT_TEMPLATE)` class-attribute instance-attribute
- `input` property writable
- `text_to_speech_driver = field(default=Factory(lambda: Defaults.drivers_config.text_to_speech_driver), kw_only=True)` class-attribute instance-attribute

try_run()

Source Code in griptape/tasks/text_to_speech_task.py

```
def try_run(self) -> AudioArtifact:
    audio_artifact = self.text_to_speech_driver.run_text_to_audio(prompts=[self.input.
↪to_text()])

    if self.output_dir or self.output_file:
        self._write_to_file(audio_artifact)

    return audio_artifact
```

tool_task

Bases: *PromptTask* “) , *ActionsSubtaskOriginMixin* ”)

Source Code in griptape/tasks/tool_task.py

```
@define
class ToolTask(PromptTask, ActionsSubtaskOriginMixin):
    DEFAULT_MAX_STEPS = 0
    ACTION_PATTERN = r"(?s)[^{]*({.*})"

    tool: BaseTool = field(kw_only=True, metadata={"serializable": True})
    subtask: Optional[ActionsSubtask] = field(default=None, kw_only=True)
```

(continues on next page)

(continued from previous page)

```
    task_memory: Optional[TaskMemory] = field(default=None, kw_only=True)
    tools: list[BaseTool] = field(factory=list, kw_only=True, metadata={"serializable
↪": False})
    max_subtasks: int = field(default=DEFAULT_MAX_STEPS, kw_only=True, metadata={
↪"serializable": False})

    @property
    def prompt_stack(self) -> PromptStack:
        stack = super().prompt_stack
        stack.tools = [self.tool]

        return stack

    def __attrs_post_init__(self) -> None:
        super().__attrs_post_init__()
        if self.task_memory is not None:
            self.set_default_tools_memory(self.task_memory)

    def preprocess(self, structure: Structure) -> ToolTask:
        super().preprocess(structure)

        if self.task_memory is None and structure.task_memory is not None:
            self.set_default_tools_memory(structure.task_memory)

        return self

    def default_generate_system_template(self, _: PromptTask) -> str:
        return J2("tasks/tool_task/system.j2").render(
            rulesets=J2("rulesets/rulesets.j2").render(rulesets=self.rulesets),
            action_schema=utils.minify_json(json.dumps(self.tool.schema())),
            meta_memory=J2("memory/meta/meta_memory.j2").render(meta_memories=self.
↪meta_memories),
            use_native_tools=self.prompt_driver.use_native_tools,
        )

    def actions_schema(self) -> Schema:
        return self._actions_schema_for_tools([self.tool])

    def try_run(self) -> ListArtifact | TextArtifact | ErrorArtifact:
        warnings.warn(
            "`ToolTask` is deprecated and will be removed in a future release. Use␣
↪`PromptTask` with `reflect_on_tool_use=False` instead.",
            DeprecationWarning,
            stacklevel=2,
        )
        result = self.prompt_driver.run(self.prompt_stack)

        if self.prompt_driver.use_native_tools:
            subtask_input = result.to_artifact(meta={"is_react_prompt": False})
        else:
            action_matches = re.findall(self.ACTION_PATTERN, result.to_text(), re.
↪DOTALL)
```

(continues on next page)

(continued from previous page)

```
            if not action_matches:
                return ErrorArtifact("No action found in prompt output.")
            data = action_matches[-1]
            action_dict = json.loads(data)

            action_dict["tag"] = self.tool.name
            subtask_input = TextArtifact(
                J2("tasks/tool_task/subtask.j2").render(action_json=json.dumps(action_
↪dict)),
                meta={"is_react_prompt": True},
            )

        try:
            subtask = self.add_subtask(ActionsSubtask(subtask_input))

            output = subtask.run()

            if isinstance(output, ListArtifact):
                output = output[0]
        except Exception as e:
            output = ErrorArtifact(f"Error processing tool input: {e}", exception=e)
        return output

    def find_tool(self, tool_name: str) -> BaseTool:
        if self.tool.name == tool_name:
            return self.tool
        raise ValueError(f"Tool with name {tool_name} not found.")

    def find_memory(self, memory_name: str) -> TaskMemory:
        raise NotImplementedError("ToolTask does not support Task Memory.")

    def find_subtask(self, subtask_id: str) -> ActionsSubtask:
        if self.subtask and self.subtask.id == subtask_id:
            return self.subtask
        raise ValueError(f"Subtask with id {subtask_id} not found.")

    def add_subtask(self, subtask: BaseSubtask) -> ActionsSubtask:
        if not isinstance(subtask, ActionsSubtask):
            raise TypeError("Subtask must be an instance of ActionsSubtask.")

        self.subtask = subtask
        self.subtask.attach_to(self)

        return self.subtask

    def set_default_tools_memory(self, memory: TaskMemory) -> None:
        self.task_memory = memory

        if self.task_memory:
            if self.tool.input_memory is None:
                self.tool.input_memory = [self.task_memory]
```

(continues on next page)

(continued from previous page)

```
            if self.tool.output_memory is None and self.tool.off_prompt:
                self.tool.output_memory = {getattr(a, "name"): [self.task_memory] for␣
↪a in self.tool.activities()}
```

- `ACTION_PATTERN = '(?s)[^{]*({.*})'` class-attribute instance-attribute
- `DEFAULT_MAX_STEPS = 0` class-attribute instance-attribute
- `max_subtasks = field(default=DEFAULT_MAX_STEPS, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `prompt_stack` property
- `subtask = field(default=None, kw_only=True)` class-attribute instance-attribute
- `task_memory = field(default=None, kw_only=True)` class-attribute instance-attribute
- `tool = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `tools = field(factory=list, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/tasks/tool_task.py

```
def __attrs_post_init__(self) -> None:
    super().__attrs_post_init__()
    if self.task_memory is not None:
        self.set_default_tools_memory(self.task_memory)
```

actions_schema()

Source Code in griptape/tasks/tool_task.py

```
def actions_schema(self) -> Schema:
    return self._actions_schema_for_tools([self.tool])
```

add_subtask(subtask)

Source Code in griptape/tasks/tool_task.py

```
def add_subtask(self, subtask: BaseSubtask) -> ActionsSubtask:
    if not isinstance(subtask, ActionsSubtask):
        raise TypeError("Subtask must be an instance of ActionsSubtask.")

    self.subtask = subtask
    self.subtask.attach_to(self)

    return self.subtask
```

default*generate_system_template()*

Source Code in griptape/tasks/tool_task.py

```
def default_generate_system_template(self, _: PromptTask) -> str:
    return J2("tasks/tool_task/system.j2").render(
        rulesets=J2("rulesets/rulesets.j2").render(rulesets=self.rulesets),
        action_schema=utils.minify_json(json.dumps(self.tool.schema())),
        meta_memory=J2("memory/meta/meta_memory.j2").render(meta_memories=self.meta_
↪memories),
        use_native_tools=self.prompt_driver.use_native_tools,
    )
```

find_memory(memory_name)

Source Code in griptape/tasks/tool_task.py

```
def find_memory(self, memory_name: str) -> TaskMemory:
    raise NotImplementedError("ToolTask does not support Task Memory.")
```

find_subtask(subtask_id)

Source Code in griptape/tasks/tool_task.py

```
def find_subtask(self, subtask_id: str) -> ActionsSubtask:
    if self.subtask and self.subtask.id == subtask_id:
        return self.subtask
    raise ValueError(f"Subtask with id {subtask_id} not found.")
```

find_tool(tool_name)

Source Code in griptape/tasks/tool_task.py

```
def find_tool(self, tool_name: str) -> BaseTool:
    if self.tool.name == tool_name:
        return self.tool
    raise ValueError(f"Tool with name {tool_name} not found.")
```

preprocess(structure)

Source Code in griptape/tasks/tool_task.py

```
def preprocess(self, structure: Structure) -> ToolTask:
    super().preprocess(structure)

    if self.task_memory is None and structure.task_memory is not None:
        self.set_default_tools_memory(structure.task_memory)

    return self
```

set_default_tools_memory(memory)

Source Code in griptape/tasks/tool_task.py

```
def set_default_tools_memory(self, memory: TaskMemory) -> None:
    self.task_memory = memory

    if self.task_memory:
        if self.tool.input_memory is None:
            self.tool.input_memory = [self.task_memory]
        if self.tool.output_memory is None and self.tool.off_prompt:
            self.tool.output_memory = {getattr(a, "name"): [self.task_memory] for a␣
↪in self.tool.activities()}
```

try_run()

Source Code in griptape/tasks/tool_task.py

```
def try_run(self) -> ListArtifact | TextArtifact | ErrorArtifact:
    warnings.warn(
        "`ToolTask` is deprecated and will be removed in a future release. Use␣
↪`PromptTask` with `reflect_on_tool_use=False` instead.",
        DeprecationWarning,
        stacklevel=2,
    )
    result = self.prompt_driver.run(self.prompt_stack)

    if self.prompt_driver.use_native_tools:
        subtask_input = result.to_artifact(meta={"is_react_prompt": False})
    else:
        action_matches = re.findall(self.ACTION_PATTERN, result.to_text(), re.DOTALL)

        if not action_matches:
            return ErrorArtifact("No action found in prompt output.")
        data = action_matches[-1]
        action_dict = json.loads(data)

        action_dict["tag"] = self.tool.name
        subtask_input = TextArtifact(
            J2("tasks/tool_task/subtask.j2").render(action_json=json.dumps(action_
↪dict)),
            meta={"is_react_prompt": True},
        )

    try:
        subtask = self.add_subtask(ActionsSubtask(subtask_input))

        output = subtask.run()

        if isinstance(output, ListArtifact):
            output = output[0]
    except Exception as e:
        output = ErrorArtifact(f"Error processing tool input: {e}", exception=e)
    return output
```

toolkit_task

Bases: *PromptTask* “)

Source Code in griptape/tasks/toolkit_task.py

```
@define
class ToolkitTask(PromptTask):
    def try_run(self) -> ListArtifact | TextArtifact | AudioArtifact |␣
↪GenericArtifact | JsonArtifact | ErrorArtifact:
        warnings.warn(
            "`ToolkitTask` is deprecated and will be removed in a future release.␣
↪`PromptTask` is a drop-in replacement.",
            DeprecationWarning,
            stacklevel=2,
        )

        return super().try_run()
```

try_run()

Source Code in griptape/tasks/toolkit_task.py

```
def try_run(self) -> ListArtifact | TextArtifact | AudioArtifact | GenericArtifact |␣
↪JsonArtifact | ErrorArtifact:
    warnings.warn(
        "`ToolkitTask` is deprecated and will be removed in a future release.␣
↪`PromptTask` is a drop-in replacement.",
        DeprecationWarning,
        stacklevel=2,
    )

    return super().try_run()
```

variation_image_generation_task

Bases: *BaseImageGenerationTask* “)

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">BaseImageGenerationDrive	The engine used to generate the image.
<code class="language-python">	<code class="language-python">list[Ruleset]</code>	List of negatively-weighted rulesets applied to the text prompt, if supported by the driver.

continues on next page

Table 380 – continued from previous page

Name	Type	Description
<code class="language-python">	<code class="language-python">list[Rule]</code>	List of negatively-weighted rules applied to the text prompt, if supported by the driver.
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.variation_image_generation_task.VariationImageGenerationTask.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk in output_dir.

continues on next page

Table 380 – continued from previous page

Name	Type	Description
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tasks.variation_image_generation_task.VariationImageGenerationTask.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk as output_file.

Source Code in griptape/tasks/variation_image_generation_task.py

```
@define
class VariationImageGenerationTask(BaseImageGenerationTask):
    """A task that generates a variation of an image using a prompt.

    Accepts a text prompt and image as
    input in one of the following formats:
    - tuple of (template string, ImageArtifact)
    - tuple of (TextArtifact, ImageArtifact)
    - Callable that returns a tuple of (TextArtifact, ImageArtifact).

    Attributes:
        image_generation_driver: The engine used to generate the image.
        negative_rulesets: List of negatively-weighted rulesets applied to the text␣
↪prompt, if supported by the driver.
        negative_rules: List of negatively-weighted rules applied to the text prompt,␣
↪if supported by the driver.
        output_dir: If provided, the generated image will be written to disk in␣
↪output_dir.
        output_file: If provided, the generated image will be written to disk as␣
↪output_file.
    """

    image_generation_driver: BaseImageGenerationDriver = field(
        default=Factory(lambda: Defaults.drivers_config.image_generation_driver),
        kw_only=True,
    )
    _input: Union[tuple[Union[str, TextArtifact], ImageArtifact], Callable[[BaseTask],
↪ ListArtifact], ListArtifact] = (
```

(continues on next page)

(continued from previous page)

```
        field(default=None, alias="input")
    )

    @property
    def input(self) -> ListArtifact:
        if isinstance(self._input, ListArtifact):
            return self._input
        if isinstance(self._input, tuple):
            if isinstance(self._input[0], TextArtifact):
                input_text = self._input[0]
            else:
                input_text = TextArtifact(J2().render_from_string(self._input[0],␣
↪**self.full_context))

            return ListArtifact([input_text, self._input[1]])
        if isinstance(self._input, Callable):
            return self._input(self)
        raise ValueError("Input must be a tuple of (text, image) or a callable that␣
↪returns such a tuple.")

    @input.setter
    def input(self, value: tuple[str | TextArtifact, ImageArtifact] |␣
↪Callable[[BaseTask], ListArtifact]) -> None:
        self._input = value

    def try_run(self) -> ImageArtifact:
        prompt_artifact = self.input[0]

        image_artifact = self.input[1]
        if not isinstance(image_artifact, ImageArtifact):
            raise ValueError("Image must be an ImageArtifact.")

        output_image_artifact = self.image_generation_driver.run_image_variation(
            prompts=self._get_prompts(prompt_artifact.to_text()),
            negative_prompts=self._get_negative_prompts(),
            image=image_artifact,
        )

        if self.output_dir or self.output_file:
            self._write_to_file(output_image_artifact)

        return output_image_artifact
```

- `_input = field(default=None, alias='input')` class-attribute instance-attribute
- `image_generation_driver = field(default=Factory(lambda: Defaults.drivers_config.image_generation_driver), kw_only=True)` class-attribute instance-attribute
- `input` property writable

try_run()

Source Code in griptape/tasks/variation_image_generation_task.py

```
def try_run(self) -> ImageArtifact:
    prompt_artifact = self.input[0]

    image_artifact = self.input[1]
    if not isinstance(image_artifact, ImageArtifact):
        raise ValueError("Image must be an ImageArtifact.")

    output_image_artifact = self.image_generation_driver.run_image_variation(
        prompts=self._get_prompts(prompt_artifact.to_text()),
        negative_prompts=self._get_negative_prompts(),
        image=image_artifact,
    )

    if self.output_dir or self.output_file:
        self._write_to_file(output_image_artifact)

    return output_image_artifact
```

11.5.19 tokenizers

- `__all__ = ['AmazonBedrockTokenizer', 'AnthropicTokenizer', 'BaseTokenizer', 'CohereTokenizer', 'DummyTokenizer', 'GoogleTokenizer', 'GrokTokenizer', 'HuggingFaceTokenizer', 'OpenAiTokenizer', 'SimpleTokenizer', 'VoyageAiTokenizer']` module-attribute

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/amazon_bedrock_tokenizer.py

```
@define()
class AmazonBedrockTokenizer(BaseTokenizer):
    MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {
        "anthropic.claude-3": 200000,
        "anthropic.claude-v2:1": 200000,
        "anthropic.claude": 100000,
        "cohere.command-r": 128000,
        "cohere.embed": 512,
        "cohere.command": 4000,
        "cohere": 1024,
        "ai21": 8192,
        "meta.llama3-8b-instruct": 8000,
        "meta.llama3-70b-instruct": 8000,
        "meta.llama3-2-1b-instruct": 131000,
        "meta.llama3-2-3b-instruct": 131000,
        "meta.llama3": 128000,
        "mistral.large-2407": 128000,
        "mistral.mistral": 32000,
        "mistral.mixtral": 32000,
        "amazon.nova-micro-v1": 128000,
        "amazon.nova": 300000,
```

(continues on next page)

(continued from previous page)

```
        "amazon.titan-embed-image": 128000,
        "amazon.titan-embed-text": 8000,
        "amazon.titan-text-express-v1": 8000,
        "amazon.titan-text-lite-v1": 4000,
        "amazon.titan-text-premier-v1": 32000,
    }
    MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {
        "anthropic.claude-3-7": 8192,
        "anthropic.claude-3-5": 8192,
        "anthropic.claude": 4096,
        "cohere": 4096,
        "ai21.j2": 8191,
        "meta": 2048,
        "amazon.titan-text-lite": 4096,
        "amazon.titan-text-express": 8192,
        "amazon.titan-text-premier": 3072,
        "amazon.nova": 5000,
        "mistral.mistral": 8192,
        "mistral.mixtral": 4096,
    }

    model: str = field(kw_only=True)
    characters_per_token: int = field(default=4, kw_only=True)

    def count_tokens(self, text: str) -> int:
        return (len(text) + self.characters_per_token - 1) // self.characters_per_
↪token
```

- `MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {'anthropic.claude-3': 200000, 'anthropic.claude-v2:1': 200000, 'anthropic.claude': 100000, 'cohere.command-r': 128000, 'cohere.embed': 512, 'cohere.command': 4000, 'cohere': 1024, 'ai21': 8192, 'meta.llama3-8b-instruct': 8000, 'meta.llama3-70b-instruct': 8000, 'meta.llama3-2-1b-instruct': 131000, 'meta.llama3-2-3b-instruct': 131000, 'meta.llama3': 128000, 'mistral.large-2407': 128000, 'mistral.mistral': 32000, 'mistral.mixtral': 32000, 'amazon.nova-micro-v1': 128000, 'amazon.nova': 300000, 'amazon.titan-embed-image': 128000, 'amazon.titan-embed-text': 8000, 'amazon.titan-text-express-v1': 8000, 'amazon.titan-text-lite-v1': 4000, 'amazon.titan-text-premier-v1': 32000}` class-attribute instance-attribute
- `MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {'anthropic.claude-3-7': 8192, 'anthropic.claude-3-5': 8192, 'anthropic.claude': 4096, 'cohere': 4096, 'ai21.j2': 8191, 'meta': 2048, 'amazon.titan-text-lite': 4096, 'amazon.titan-text-express': 8192, 'amazon.titan-text-premier': 3072, 'amazon.nova': 5000, 'mistral.mistral': 8192, 'mistral.mixtral': 4096}` class-attribute instance-attribute
- `characters_per_token = field(default=4, kw_only=True)` class-attribute instance-attribute
- `model = field(kw_only=True)` class-attribute instance-attribute

count_tokens(text)

Source Code in griptape/tokenizers/amazon_bedrock_tokenizer.py

```
def count_tokens(self, text: str) -> int:
    return (len(text) + self.characters_per_token - 1) // self.characters_per_token
```

AnthropicTokenizer

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/anthropic_tokenizer.py

```
@define()
class AnthropicTokenizer(BaseTokenizer):
    MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {"claude-3": 200000, "claude-2.1": 200000,
↪"claude": 100000}
    MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {"claude": 4096}

    client: Anthropic = field(
        default=Factory(lambda: import_optional_dependency("anthropic").Anthropic()),
        kw_only=True,
    )

    def count_tokens(self, text: str | list[BetaMessageParam]) -> int:
        types = import_optional_dependency("anthropic.types.beta")

        # TODO: Refactor all Tokenizers to support Prompt Stack as an input.
        messages = [types.BetaMessageParam(role="user", content=text)] if␣
↪isinstance(text, str) else text

        usage = self.client.beta.messages.count_tokens(
            model=self.model,
            messages=messages,
        )

        return usage.input_tokens
```

- `MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {'claude-3': 200000, 'claude-2.1': 200000, 'claude': 100000}` class-attribute instance-attribute
- `MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {'claude': 4096}` class-attribute instance-attribute
- `client = field(default=Factory(lambda: import_optional_dependency('anthropic').Anthropic()), kw_only=True)` class-attribute instance-attribute

count_tokens(text)

Source Code in griptape/tokenizers/anthropic_tokenizer.py

```
def count_tokens(self, text: str | list[BetaMessageParam]) -> int:
    types = import_optional_dependency("anthropic.types.beta")

    # TODO: Refactor all Tokenizers to support Prompt Stack as an input.
    messages = [types.BetaMessageParam(role="user", content=text)] if isinstance(text,
```

(continues on next page)

(continued from previous page)

```
↪ str) else text

    usage = self.client.beta.messages.count_tokens(
        model=self.model,
        messages=messages,
    )

    return usage.input_tokens
```

BaseTokenizer

Bases:

ABC, *SerializableMixin* “)

Source Code in griptape/tokenizers/base_tokenizer.py

```
@define()
class BaseTokenizer(ABC, SerializableMixin):
    DEFAULT_MAX_INPUT_TOKENS = 4096
    DEFAULT_MAX_OUTPUT_TOKENS = 1000
    MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {}
    MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {}

    model: str = field(kw_only=True, metadata={"serializable": True})
    stop_sequences: list[str] = field(default=Factory(list), kw_only=True, metadata={
↪"serializable": True})
    _max_input_tokens: Optional[int] = field(
        kw_only=True, default=None, alias="max_input_tokens", metadata={"serializable
↪": True}
    )
    _max_output_tokens: Optional[int] = field(
        kw_only=True, default=None, alias="max_output_tokens", metadata={"serializable
↪": True}
    )

    @lazy_property()
    def max_input_tokens(self) -> int:
        return self._default_max_input_tokens()

    @lazy_property()
    def max_output_tokens(self) -> int:
        return self._default_max_output_tokens()

    def __attrs_post_init__(self) -> None:
        if self.model is not None:
            if self.max_input_tokens is None:
                self.max_input_tokens = self._default_max_input_tokens()

            if self.max_output_tokens is None:
                # TODO(collin): https://github.com/griptape-ai/griptape/issues/1844
                self.max_output_tokens = self._default_max_output_tokens()
```

(continues on next page)

(continued from previous page)

```
    def count_input_tokens_left(self, text: str) -> int:
        diff = self.max_input_tokens - self.count_tokens(text)

        if diff > 0:
            return diff
        return 0

    def count_output_tokens_left(self, text: str) -> int:
        diff = self.max_output_tokens - self.count_tokens(text)

        if diff > 0:
            return diff
        return 0

    @abstractmethod
    def count_tokens(self, text: str) -> int: ...

    def _default_max_input_tokens(self) -> int:
        tokens = next(
            (
                max_tokens
                for model_prefix, max_tokens in self.MODEL_PREFIXES_TO_MAX_INPUT_
↪TOKENS.items()
                if model_prefix in self.model
            ),
            None,
        )

        if tokens is None:
            logging.warning(
                "Model %s not found in MODEL_PREFIXES_TO_MAX_INPUT_TOKENS, using␣
↪default value of %s.",
                self.model,
                self.DEFAULT_MAX_INPUT_TOKENS,
            )
            return self.DEFAULT_MAX_INPUT_TOKENS
        return tokens

    def _default_max_output_tokens(self) -> int:
        tokens = next(
            (
                max_tokens
                for model_prefix, max_tokens in self.MODEL_PREFIXES_TO_MAX_OUTPUT_
↪TOKENS.items()
                if model_prefix in self.model
            ),
            None,
        )

        if tokens is None:
            logging.debug(
                "Model %s not found in MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS, using␣
```

(continues on next page)

(continued from previous page)

```
↪default value of %s.",
                self.model,
                self.DEFAULT_MAX_OUTPUT_TOKENS,
            )
            return self.DEFAULT_MAX_OUTPUT_TOKENS
        return tokens
```

- `DEFAULT_MAX_INPUT_TOKENS = 4096` class-attribute instance-attribute
- `DEFAULT_MAX_OUTPUT_TOKENS = 1000` class-attribute instance-attribute
- `MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {}` class-attribute instance-attribute
- `MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {}` class-attribute instance-attribute
- `_max_input_tokens = field(kw_only=True, default=None, alias='max_input_tokens', metadata={'serializable': True})` class-attribute instance-attribute
- `_max_output_tokens = field(kw_only=True, default=None, alias='max_output_tokens', metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `stop_sequences = field(default=Factory(list), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/tokenizers/base_tokenizer.py

```
def __attrs_post_init__(self) -> None:
    if self.model is not None:
        if self.max_input_tokens is None:
            self.max_input_tokens = self._default_max_input_tokens()

        if self.max_output_tokens is None:
            # TODO(collin): https://github.com/griptape-ai/griptape/issues/1844
            self.max_output_tokens = self._default_max_output_tokens()
```

_default_max_input_tokens()

Source Code in griptape/tokenizers/base_tokenizer.py

```
def _default_max_input_tokens(self) -> int:
    tokens = next(
        (
            max_tokens
            for model_prefix, max_tokens in self.MODEL_PREFIXES_TO_MAX_INPUT_TOKENS.
↪items()
            if model_prefix in self.model
        ),
        None,
    )

    if tokens is None:
```

(continues on next page)

(continued from previous page)

```
        logging.warning(
            "Model %s not found in MODEL_PREFIXES_TO_MAX_INPUT_TOKENS, using default␣
↪value of %s.",
            self.model,
            self.DEFAULT_MAX_INPUT_TOKENS,
        )
        return self.DEFAULT_MAX_INPUT_TOKENS
    return tokens
```

_default_max_output_tokens()

Source Code in griptape/tokenizers/base_tokenizer.py

```
def _default_max_output_tokens(self) -> int:
    tokens = next(
        (
            max_tokens
            for model_prefix, max_tokens in self.MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS.
↪items()
            if model_prefix in self.model
        ),
        None,
    )

    if tokens is None:
        logging.debug(
            "Model %s not found in MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS, using default␣
↪value of %s.",
            self.model,
            self.DEFAULT_MAX_OUTPUT_TOKENS,
        )
        return self.DEFAULT_MAX_OUTPUT_TOKENS
    return tokens
```

count_input_tokens_left(text)

Source Code in griptape/tokenizers/base_tokenizer.py

```
def count_input_tokens_left(self, text: str) -> int:
    diff = self.max_input_tokens - self.count_tokens(text)

    if diff > 0:
        return diff
    return 0
```

count_output_tokens_left(text)

Source Code in griptape/tokenizers/base_tokenizer.py

```
def count_output_tokens_left(self, text: str) -> int:
    diff = self.max_output_tokens - self.count_tokens(text)

```

(continues on next page)

(continued from previous page)

```
    if diff > 0:
        return diff
    return 0
```

count_tokens(text)abstractmethod

Source Code in griptape/tokenizers/base_tokenizer.py

```
@abstractmethod
def count_tokens(self, text: str) -> int: ...
```

max_input_tokens()

Source Code in griptape/tokenizers/base_tokenizer.py

```
@lazy_property()
def max_input_tokens(self) -> int:
    return self._default_max_input_tokens()
```

max_output_tokens()

Source Code in griptape/tokenizers/base_tokenizer.py

```
@lazy_property()
def max_output_tokens(self) -> int:
    return self._default_max_output_tokens()
```

CohereTokenizer

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/cohere_tokenizer.py

```
@define()
class CohereTokenizer(BaseTokenizer):
    MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {"command-r": 128000, "command": 4096, "embed
↪": 512}
    MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {"command": 4096, "embed": 512}

    client: Client = field(kw_only=True)

    def count_tokens(self, text: str) -> int:
        return len(self.client.tokenize(text=text, model=self.model).tokens)
```

- `MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {'command-r': 128000, 'command': 4096, 'embed': 512}` class-attribute instance-attribute
- `MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {'command': 4096, 'embed': 512}` class-attribute instance-attribute
- `client = field(kw_only=True)` class-attribute instance-attribute

count_tokens(text)

Source Code in griptape/tokenizers/cohere_tokenizer.py

```
def count_tokens(self, text: str) -> int:
    return len(self.client.tokenize(text=text, model=self.model).tokens)
```

DummyTokenizer

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/dummy_tokenizer.py

```
@define
class DummyTokenizer(BaseTokenizer):
    model: Optional[str] = field(default=None, kw_only=True)
    _max_input_tokens: int = field(init=False, default=0, kw_only=True, alias="max_
↪input_tokens")
    _max_output_tokens: int = field(init=False, default=0, kw_only=True, alias="max_
↪output_tokens")

    def count_tokens(self, text: str) -> int:
        raise DummyError(__class__.__name__, "count_tokens")
```

- `_max_input_tokens = field(init=False, default=0, kw_only=True, alias='max_input_tokens')` class-attribute instance-attribute
- `_max_output_tokens = field(init=False, default=0, kw_only=True, alias='max_output_tokens')` class-attribute instance-attribute
- `model = field(default=None, kw_only=True)` class-attribute instance-attribute

count_tokens(text)

Source Code in griptape/tokenizers/dummy_tokenizer.py

```
def count_tokens(self, text: str) -> int:
    raise DummyError(__class__.__name__, "count_tokens")
```

GoogleTokenizer

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/google_tokenizer.py

```
@define()
class GoogleTokenizer(BaseTokenizer):
    MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {"gemini-1.5-pro": 2097152, "gemini":␣
↪1048576}
    MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {"gemini": 8192}

    api_key: str = field(kw_only=True, metadata={"serializable": True})
    _client: Optional[GenerativeModel] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )
```

(continues on next page)

(continued from previous page)

```
    @lazy_property()
    def client(self) -> GenerativeModel:
        genai = import_optional_dependency("google.generativeai")
        genai.configure(api_key=self.api_key)

        return genai.GenerativeModel(self.model)

    def count_tokens(self, text: str) -> int:
        return self.client.count_tokens(text).total_tokens
```

- `MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {'gemini-1.5-pro': 2097152, 'gemini': 1048576}` class-attribute instance-attribute
- `MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {'gemini': 8192}` class-attribute instance-attribute
- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/tokenizers/google_tokenizer.py

```
@lazy_property()
def client(self) -> GenerativeModel:
    genai = import_optional_dependency("google.generativeai")
    genai.configure(api_key=self.api_key)

    return genai.GenerativeModel(self.model)
```

count_tokens(text)

Source Code in griptape/tokenizers/google_tokenizer.py

```
def count_tokens(self, text: str) -> int:
    return self.client.count_tokens(text).total_tokens
```

GrokTokenizer

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/grok_tokenizer.py

```
@define()
class GrokTokenizer(BaseTokenizer):
    # https://docs.x.ai/docs/models?cluster=us-east-1#model-constraints
    MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {
        "grok-2-vision": 32768,
        "grok-2": 131072,
        "grok-vision-beta": 8192,
        "grok-beta": 131072,
    }
```

(continues on next page)

(continued from previous page)

```
    MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {"grok": 4096}

    base_url: str = field(default="https://api.x.ai", kw_only=True, metadata={
↪"serializable": True})
    api_key: str = field(kw_only=True, default=None)
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True), kw_only=True
    )

    def count_tokens(self, text: str) -> int:
        response = requests.post(
            urljoin(self.base_url, "/v1/tokenize-text"),
            headers=self.headers,
            json={"text": text, "model": self.model},
        )
        response.raise_for_status()
        return len(response.json()["token_ids"])
```

- `MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {'grok-2-vision': 32768, 'grok-2': 131072, 'grok-vision-beta': 8192, 'grok-beta': 131072}` class-attribute instance-attribute
- `MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {'grok': 4096}` class-attribute instance-attribute
- `api_key = field(kw_only=True, default=None)` class-attribute instance-attribute
- `base_url = field(default='https://api.x.ai', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute

count_tokens(text)

Source Code in griptape/tokenizers/grok_tokenizer.py

```
def count_tokens(self, text: str) -> int:
    response = requests.post(
        urljoin(self.base_url, "/v1/tokenize-text"),
        headers=self.headers,
        json={"text": text, "model": self.model},
    )
    response.raise_for_status()
    return len(response.json()["token_ids"])
```

HuggingFaceTokenizer

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/huggingface_tokenizer.py

```
@define()
class HuggingFaceTokenizer(BaseTokenizer):
    tokenizer: PreTrainedTokenizerBase = field(
```

(continues on next page)

(continued from previous page)

```
        default=Factory(
            lambda self: import_optional_dependency("transformers").AutoTokenizer.
↪from_pretrained(self.model),
            takes_self=True,
        ),
        kw_only=True,
    )
    _max_input_tokens: int = field(
        default=Factory(lambda self: self.tokenizer.model_max_length, takes_
↪self=True),
        kw_only=True,
        alias="max_input_tokens",
    )
    _max_output_tokens: int = field(default=4096, kw_only=True, alias="max_output_
↪tokens")

    def count_tokens(self, text: str) -> int:
        return len(self.tokenizer.encode(text))  # pyright: ignore[reportArgumentType]
```

- `_max_input_tokens = field(default=Factory(lambda self: self.tokenizer.model_max_length, takes_self=True), kw_only=True, alias='max_input_tokens')`
  class-attribute instance-attribute
- `_max_output_tokens = field(default=4096, kw_only=True, alias='max_output_tokens')`
  class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: import_optional_dependency('transformers').AutoTokenizer.from_pretrained(self.model), takes_self=True), kw_only=True)`
  class-attribute instance-attribute

count_tokens(text)

Source Code in griptape/tokenizers/huggingface_tokenizer.py

```
def count_tokens(self, text: str) -> int:
    return len(self.tokenizer.encode(text))  # pyright: ignore[reportArgumentType]
```

OpenAiTokenizer

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/openai_tokenizer.py

```
@define()
class OpenAiTokenizer(BaseTokenizer):
    DEFAULT_OPENAI_GPT_3_COMPLETION_MODEL = "gpt-3.5-turbo-instruct"
    DEFAULT_OPENAI_GPT_3_CHAT_MODEL = "gpt-3.5-turbo"
    DEFAULT_OPENAI_GPT_4_MODEL = "gpt-4o"
    DEFAULT_ENCODING = "cl100k_base"
    DEFAULT_MAX_TOKENS = 2049
    DEFAULT_MAX_OUTPUT_TOKENS = 4096
    TOKEN_OFFSET = 8

    # https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo
```

(continues on next page)

(continued from previous page)

```
    MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {
        "gpt-4.1": 1_000_000,
        "gpt-4o": 128000,
        "gpt-4-1106": 128000,
        "gpt-4-32k": 32768,
        "gpt-4": 8192,
        "gpt-3.5-turbo-16k": 16384,
        "gpt-3.5-turbo": 4096,
        "gpt-35-turbo-16k": 16384,
        "gpt-35-turbo": 4096,
        "text-embedding-ada-002": 8191,
        "text-embedding-ada-001": 2046,
        "text-embedding-3-small": 8191,
        "text-embedding-3-large": 8191,
    }

    MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {"gpt": 4096}

    EMBEDDING_MODELS = [
        "text-embedding-ada-002",
        "text-embedding-ada-001",
        "text-embedding-3-small",
        "text-embedding-3-large",
    ]

    _max_input_tokens: int = field(
        kw_only=True,
        default=Factory(lambda self: self._default_max_input_tokens(), takes_
↪self=True),
        alias="max_input_tokens",
    )
    _max_output_tokens: int = field(
        kw_only=True,
        default=Factory(lambda self: self._default_max_output_tokens(), takes_
↪self=True),
        alias="max_output_tokens",
    )

    @property
    def encoding(self) -> tiktoken.Encoding:
        try:
            return tiktoken.encoding_for_model(self.model)
        except KeyError:
            return tiktoken.get_encoding(self.DEFAULT_ENCODING)

    def _default_max_input_tokens(self) -> int:
        tokens = next((v for k, v in self.MODEL_PREFIXES_TO_MAX_INPUT_TOKENS.items()␣
↪if self.model.startswith(k)), None)
        offset = 0 if self.model in self.EMBEDDING_MODELS else self.TOKEN_OFFSET

        return (tokens or self.DEFAULT_MAX_TOKENS) - offset
```

(continues on next page)

(continued from previous page)

```
    def _default_max_output_tokens(self) -> int:
        tokens = next(
            (v for k, v in self.MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS.items() if self.
↪model.startswith(k)),
            None,
        )

        if tokens is None:
            return self.DEFAULT_MAX_OUTPUT_TOKENS
        return tokens

    def count_tokens(self, text: str | list[dict], model: Optional[str] = None) ->␣
↪int:  # noqa: C901
        """Handles the special case of ChatML.

        Implementation adopted from the official OpenAI notebook:
        https://github.com/openai/openai-cookbook/blob/main/examples/How_to_count_
↪tokens_with_tiktoken.ipynb.
        """
        if isinstance(text, list):
            model = model or self.model

            try:
                encoding = tiktoken.encoding_for_model(model)
            except KeyError:
                logging.warning("model not found. Using cl100k_base encoding.")

                encoding = tiktoken.get_encoding("cl100k_base")

            if model in {
                "gpt-3.5-turbo-0613",
                "gpt-3.5-turbo-16k-0613",
                "gpt-4-0314",
                "gpt-4-32k-0314",
                "gpt-4-0613",
                "gpt-4-32k-0613",
                "gpt-4o-2024-05-13",
            }:
                tokens_per_message = 3
                tokens_per_name = 1
            elif model == "gpt-3.5-turbo-0301":
                # every message follows <|start|>{role/name}\n{content}<|end|>\n
                tokens_per_message = 4
                # if theres a name, the role is omitted
                tokens_per_name = -1
            elif "gpt-3.5-turbo" in model or "gpt-35-turbo" in model:
                logging.info("gpt-3.5-turbo may update over time. Returning num␣
↪tokens assuming gpt-3.5-turbo-0613.")
                return self.count_tokens(text, model="gpt-3.5-turbo-0613")
            elif "gpt-4o" in model:
                logging.info("gpt-4o may update over time. Returning num tokens␣
↪assuming gpt-4o-2024-05-13.")
```

(continues on next page)

(continued from previous page)

```
                return self.count_tokens(text, model="gpt-4o-2024-05-13")
            elif "gpt-4" in model:
                logging.info("gpt-4 may update over time. Returning num tokens␣
↪assuming gpt-4-0613.")
                return self.count_tokens(text, model="gpt-4-0613")
            else:
                raise NotImplementedError(
                    f"token_count() is not implemented for model {model}. "
                    "See https://github.com/openai/openai-python/blob/main/chatml.md␣
↪for "
                    "information on how messages are converted to tokens."
                )

            num_tokens = 0

            for message in text:
                num_tokens += tokens_per_message
                for key, value in message.items():
                    num_tokens += len(encoding.encode(value))
                    if key == "name":
                        num_tokens += tokens_per_name

            # every reply is primed with <|start|>assistant<|message|>
            num_tokens += 3

            return num_tokens
        return len(self.encoding.encode(text, allowed_special=set(self.stop_
↪sequences)))
```

- `DEFAULT_ENCODING = 'cl100k_base'` class-attribute instance-attribute
- `DEFAULT_MAX_OUTPUT_TOKENS = 4096` class-attribute instance-attribute
- `DEFAULT_MAX_TOKENS = 2049` class-attribute instance-attribute
- `DEFAULT_OPENAI_GPT_3_CHAT_MODEL = 'gpt-3.5-turbo'` class-attribute instance-attribute
- `DEFAULT_OPENAI_GPT_3_COMPLETION_MODEL = 'gpt-3.5-turbo-instruct'` class-attribute instance-attribute
- `DEFAULT_OPENAI_GPT_4_MODEL = 'gpt-4o'` class-attribute instance-attribute
- `EMBEDDING_MODELS = ['text-embedding-ada-002', 'text-embedding-ada-001', 'text-embedding-3-small', 'text-embedding-3-large']` class-attribute instance-attribute
- `MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {'gpt-4.1': 1000000, 'gpt-4o': 128000, 'gpt-4-1106': 128000, 'gpt-4-32k': 32768, 'gpt-4': 8192, 'gpt-3.5-turbo-16k': 16384, 'gpt-3.5-turbo': 4096, 'gpt-35-turbo-16k': 16384, 'gpt-35-turbo': 4096, 'text-embedding-ada-002': 8191, 'text-embedding-ada-001': 2046, 'text-embedding-3-small': 8191, 'text-embedding-3-large': 8191}` class-attribute instance-attribute
- `MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {'gpt': 4096}` class-attribute instance-attribute
- `TOKEN_OFFSET = 8` class-attribute instance-attribute
- `_max_input_tokens = field(kw_only=True, default=Factory(lambda self: self._default_max_input_tokens(), takes_self=True), alias='max_input_tokens')` class-attribute instance-attribute

- `_max_output_tokens = field(kw_only=True, default=Factory(lambda self: self._default_max_output_tokens(), takes_self=True), alias='max_output_tokens')` class-attribute instance-attribute
- `encoding` property

_default_max_input_tokens()

Source Code in griptape/tokenizers/openai_tokenizer.py

```
def _default_max_input_tokens(self) -> int:
    tokens = next((v for k, v in self.MODEL_PREFIXES_TO_MAX_INPUT_TOKENS.items() if self.model.startswith(k)), None)
    offset = 0 if self.model in self.EMBEDDING_MODELS else self.TOKEN_OFFSET

    return (tokens or self.DEFAULT_MAX_TOKENS) - offset
```

_default_max_output_tokens()

Source Code in griptape/tokenizers/openai_tokenizer.py

```
def _default_max_output_tokens(self) -> int:
    tokens = next(
        (v for k, v in self.MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS.items() if self.model.startswith(k)),
        None,
    )

    if tokens is None:
        return self.DEFAULT_MAX_OUTPUT_TOKENS
    return tokens
```

count_tokens(text, model=None)

Source Code in griptape/tokenizers/openai_tokenizer.py

```
def count_tokens(self, text: str | list[dict], model: Optional[str] = None) -> int:
# noqa: C901
    """Handles the special case of ChatML.

    Implementation adopted from the official OpenAI notebook:
    https://github.com/openai/openai-cookbook/blob/main/examples/How_to_count_tokens_with_tiktoken.ipynb.
    """
    if isinstance(text, list):
        model = model or self.model

        try:
            encoding = tiktoken.encoding_for_model(model)
        except KeyError:
            logging.warning("model not found. Using cl100k_base encoding.")

            encoding = tiktoken.get_encoding("cl100k_base")
```

(continues on next page)

(continued from previous page)

```
        if model in {
            "gpt-3.5-turbo-0613",
            "gpt-3.5-turbo-16k-0613",
            "gpt-4-0314",
            "gpt-4-32k-0314",
            "gpt-4-0613",
            "gpt-4-32k-0613",
            "gpt-4o-2024-05-13",
        }:
            tokens_per_message = 3
            tokens_per_name = 1
        elif model == "gpt-3.5-turbo-0301":
            # every message follows <|start|>{role/name}\n{content}<|end|>\n
            tokens_per_message = 4
            # if theres a name, the role is omitted
            tokens_per_name = -1
        elif "gpt-3.5-turbo" in model or "gpt-35-turbo" in model:
            logging.info("gpt-3.5-turbo may update over time. Returning num tokens␣
↪assuming gpt-3.5-turbo-0613.")
            return self.count_tokens(text, model="gpt-3.5-turbo-0613")
        elif "gpt-4o" in model:
            logging.info("gpt-4o may update over time. Returning num tokens assuming␣
↪gpt-4o-2024-05-13.")
            return self.count_tokens(text, model="gpt-4o-2024-05-13")
        elif "gpt-4" in model:
            logging.info("gpt-4 may update over time. Returning num tokens assuming␣
↪gpt-4-0613.")
            return self.count_tokens(text, model="gpt-4-0613")
        else:
            raise NotImplementedError(
                f"token_count() is not implemented for model {model}. "
                "See https://github.com/openai/openai-python/blob/main/chatml.md for "
                "information on how messages are converted to tokens."
            )

        num_tokens = 0

        for message in text:
            num_tokens += tokens_per_message
            for key, value in message.items():
                num_tokens += len(encoding.encode(value))
                if key == "name":
                    num_tokens += tokens_per_name

        # every reply is primed with <|start|>assistant<|message|>
        num_tokens += 3

        return num_tokens
    return len(self.encoding.encode(text, allowed_special=set(self.stop_sequences)))
```

SimpleTokenizer

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/simple_tokenizer.py

```
@define()
class SimpleTokenizer(BaseTokenizer):
    model: str | None = field(init=False, default=None, kw_only=True)
    characters_per_token: int = field(kw_only=True)

    def count_tokens(self, text: str) -> int:
        return (len(text) + self.characters_per_token - 1) // self.characters_per_
↪token
```

- `characters_per_token = field(kw_only=True)` class-attribute instance-attribute
- `model = field(init=False, default=None, kw_only=True)` class-attribute instance-attribute

count_tokens(text)

Source Code in griptape/tokenizers/simple_tokenizer.py

```
def count_tokens(self, text: str) -> int:
    return (len(text) + self.characters_per_token - 1) // self.characters_per_token
```

VoyageAiTokenizer

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/voyageai_tokenizer.py

```
@define()
class VoyageAiTokenizer(BaseTokenizer):
    MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {
        "voyage-3": 32000,
        "voyage-large-2": 16000,
        "voyage-code-2": 16000,
        "voyage-2": 4000,
        "voyage-lite-02-instruct": 4000,
    }
    MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {"voyage": 0}

    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    client: Client = field(
        default=Factory(
            lambda self: import_optional_dependency("voyageai").Client(api_key=self.
↪api_key),
            takes_self=True,
        ),
        kw_only=True,
    )

    def count_tokens(self, text: str) -> int:
        return self.client.count_tokens([text], model=self.model)
```

- `MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {'voyage-3': 32000, 'voyage-large-2': 16000, 'voyage-code-2': 16000, 'voyage-2': 4000, 'voyage-lite-02-instruct': 4000}` class-attribute instance-attribute
- `MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {'voyage': 0}` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `client = field(default=Factory(lambda self: import_optional_dependency('voyageai').Client(api_key=self.api_key), takes_self=True), kw_only=True)` class-attribute instance-attribute

count_tokens(text)

Source Code in griptape/tokenizers/voyageai_tokenizer.py

```
def count_tokens(self, text: str) -> int:
    return self.client.count_tokens([text], model=self.model)
```

amazon_bedrock_tokenizer

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/amazon_bedrock_tokenizer.py

```
@define()
class AmazonBedrockTokenizer(BaseTokenizer):
    MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {
        "anthropic.claude-3": 200000,
        "anthropic.claude-v2:1": 200000,
        "anthropic.claude": 100000,
        "cohere.command-r": 128000,
        "cohere.embed": 512,
        "cohere.command": 4000,
        "cohere": 1024,
        "ai21": 8192,
        "meta.llama3-8b-instruct": 8000,
        "meta.llama3-70b-instruct": 8000,
        "meta.llama3-2-1b-instruct": 131000,
        "meta.llama3-2-3b-instruct": 131000,
        "meta.llama3": 128000,
        "mistral.large-2407": 128000,
        "mistral.mistral": 32000,
        "mistral.mixtral": 32000,
        "amazon.nova-micro-v1": 128000,
        "amazon.nova": 300000,
        "amazon.titan-embed-image": 128000,
        "amazon.titan-embed-text": 8000,
        "amazon.titan-text-express-v1": 8000,
        "amazon.titan-text-lite-v1": 4000,
        "amazon.titan-text-premier-v1": 32000,
    }
    MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {
        "anthropic.claude-3-7": 8192,
```

(continues on next page)

(continued from previous page)

```
        "anthropic.claude-3-5": 8192,
        "anthropic.claude": 4096,
        "cohere": 4096,
        "ai21.j2": 8191,
        "meta": 2048,
        "amazon.titan-text-lite": 4096,
        "amazon.titan-text-express": 8192,
        "amazon.titan-text-premier": 3072,
        "amazon.nova": 5000,
        "mistral.mistral": 8192,
        "mistral.mixtral": 4096,
    }

    model: str = field(kw_only=True)
    characters_per_token: int = field(default=4, kw_only=True)

    def count_tokens(self, text: str) -> int:
        return (len(text) + self.characters_per_token - 1) // self.characters_per_
↪token
```

- `MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {'anthropic.claude-3': 200000, 'anthropic.claude-v2:1': 200000, 'anthropic.claude': 100000, 'cohere.command-r': 128000, 'cohere.embed': 512, 'cohere.command': 4000, 'cohere': 1024, 'ai21': 8192, 'meta.llama3-8b-instruct': 8000, 'meta.llama3-70b-instruct': 8000, 'meta.llama3-2-1b-instruct': 131000, 'meta.llama3-2-3b-instruct': 131000, 'meta.llama3': 128000, 'mistral.large-2407': 128000, 'mistral.mistral': 32000, 'mistral.mixtral': 32000, 'amazon.nova-micro-v1': 128000, 'amazon.nova': 300000, 'amazon.titan-embed-image': 128000, 'amazon.titan-embed-text': 8000, 'amazon.titan-text-express-v1': 8000, 'amazon.titan-text-lite-v1': 4000, 'amazon.titan-text-premier-v1': 32000}` class-attribute instance-attribute
- `MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {'anthropic.claude-3-7': 8192, 'anthropic.claude-3-5': 8192, 'anthropic.claude': 4096, 'cohere': 4096, 'ai21.j2': 8191, 'meta': 2048, 'amazon.titan-text-lite': 4096, 'amazon.titan-text-express': 8192, 'amazon.titan-text-premier': 3072, 'amazon.nova': 5000, 'mistral.mistral': 8192, 'mistral.mixtral': 4096}` class-attribute instance-attribute
- `characters_per_token = field(default=4, kw_only=True)` class-attribute instance-attribute
- `model = field(kw_only=True)` class-attribute instance-attribute

count_tokens(text)

Source Code in griptape/tokenizers/amazon_bedrock_tokenizer.py

```
def count_tokens(self, text: str) -> int:
    return (len(text) + self.characters_per_token - 1) // self.characters_per_token
```

anthropic_tokenizer

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/anthropic_tokenizer.py

```
@define()
class AnthropicTokenizer(BaseTokenizer):
    MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {"claude-3": 200000, "claude-2.1": 200000,
↪"claude": 100000}
    MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {"claude": 4096}

    client: Anthropic = field(
        default=Factory(lambda: import_optional_dependency("anthropic").Anthropic()),
        kw_only=True,
    )

    def count_tokens(self, text: str | list[BetaMessageParam]) -> int:
        types = import_optional_dependency("anthropic.types.beta")

        # TODO: Refactor all Tokenizers to support Prompt Stack as an input.
        messages = [types.BetaMessageParam(role="user", content=text)] if␣
↪isinstance(text, str) else text

        usage = self.client.beta.messages.count_tokens(
            model=self.model,
            messages=messages,
        )

        return usage.input_tokens
```

- `MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {'claude-3': 200000, 'claude-2.1': 200000, 'claude': 100000}` class-attribute instance-attribute
- `MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {'claude': 4096}` class-attribute instance-attribute
- `client = field(default=Factory(lambda: import_optional_dependency('anthropic').Anthropic()), kw_only=True)` class-attribute instance-attribute

count_tokens(text)

Source Code in griptape/tokenizers/anthropic_tokenizer.py

```
def count_tokens(self, text: str | list[BetaMessageParam]) -> int:
    types = import_optional_dependency("anthropic.types.beta")

    # TODO: Refactor all Tokenizers to support Prompt Stack as an input.
    messages = [types.BetaMessageParam(role="user", content=text)] if isinstance(text,
↪ str) else text

    usage = self.client.beta.messages.count_tokens(
        model=self.model,
        messages=messages,
    )

    return usage.input_tokens
```

base_tokenizer

Bases:

ABC, *SerializableMixin* “)

Source Code in griptape/tokenizers/base_tokenizer.py

```
@define()
class BaseTokenizer(ABC, SerializableMixin):
    DEFAULT_MAX_INPUT_TOKENS = 4096
    DEFAULT_MAX_OUTPUT_TOKENS = 1000
    MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {}
    MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {}

    model: str = field(kw_only=True, metadata={"serializable": True})
    stop_sequences: list[str] = field(default=Factory(list), kw_only=True, metadata={
↪"serializable": True})
    _max_input_tokens: Optional[int] = field(
        kw_only=True, default=None, alias="max_input_tokens", metadata={"serializable
↪": True}
    )
    _max_output_tokens: Optional[int] = field(
        kw_only=True, default=None, alias="max_output_tokens", metadata={"serializable
↪": True}
    )

    @lazy_property()
    def max_input_tokens(self) -> int:
        return self._default_max_input_tokens()

    @lazy_property()
    def max_output_tokens(self) -> int:
        return self._default_max_output_tokens()

    def __attrs_post_init__(self) -> None:
        if self.model is not None:
            if self.max_input_tokens is None:
                self.max_input_tokens = self._default_max_input_tokens()

            if self.max_output_tokens is None:
                # TODO(collin): https://github.com/griptape-ai/griptape/issues/1844
                self.max_output_tokens = self._default_max_output_tokens()

    def count_input_tokens_left(self, text: str) -> int:
        diff = self.max_input_tokens - self.count_tokens(text)

        if diff > 0:
            return diff
        return 0

    def count_output_tokens_left(self, text: str) -> int:
        diff = self.max_output_tokens - self.count_tokens(text)

        if diff > 0:
```

(continues on next page)

(continued from previous page)

```
            return diff
        return 0

    @abstractmethod
    def count_tokens(self, text: str) -> int: ...

    def _default_max_input_tokens(self) -> int:
        tokens = next(
            (
                max_tokens
                for model_prefix, max_tokens in self.MODEL_PREFIXES_TO_MAX_INPUT_
↪TOKENS.items()
                if model_prefix in self.model
            ),
            None,
        )

        if tokens is None:
            logging.warning(
                "Model %s not found in MODEL_PREFIXES_TO_MAX_INPUT_TOKENS, using␣
↪default value of %s.",
                self.model,
                self.DEFAULT_MAX_INPUT_TOKENS,
            )
            return self.DEFAULT_MAX_INPUT_TOKENS
        return tokens

    def _default_max_output_tokens(self) -> int:
        tokens = next(
            (
                max_tokens
                for model_prefix, max_tokens in self.MODEL_PREFIXES_TO_MAX_OUTPUT_
↪TOKENS.items()
                if model_prefix in self.model
            ),
            None,
        )

        if tokens is None:
            logging.debug(
                "Model %s not found in MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS, using␣
↪default value of %s.",
                self.model,
                self.DEFAULT_MAX_OUTPUT_TOKENS,
            )
            return self.DEFAULT_MAX_OUTPUT_TOKENS
        return tokens
```

- `DEFAULT_MAX_INPUT_TOKENS = 4096` class-attribute instance-attribute
- `DEFAULT_MAX_OUTPUT_TOKENS = 1000` class-attribute instance-attribute
- `MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {}` class-attribute instance-attribute

- `MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {}` class-attribute instance-attribute
- `_max_input_tokens = field(kw_only=True, default=None, alias='max_input_tokens', metadata={'serializable': True})` class-attribute instance-attribute
- `_max_output_tokens = field(kw_only=True, default=None, alias='max_output_tokens', metadata={'serializable': True})` class-attribute instance-attribute
- `model = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `stop_sequences = field(default=Factory(list), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/tokenizers/base_tokenizer.py

```
def __attrs_post_init__(self) -> None:
    if self.model is not None:
        if self.max_input_tokens is None:
            self.max_input_tokens = self._default_max_input_tokens()

        if self.max_output_tokens is None:
            # TODO(collin): https://github.com/griptape-ai/griptape/issues/1844
            self.max_output_tokens = self._default_max_output_tokens()
```

_default_max_input_tokens()

Source Code in griptape/tokenizers/base_tokenizer.py

```
def _default_max_input_tokens(self) -> int:
    tokens = next(
        (
            max_tokens
            for model_prefix, max_tokens in self.MODEL_PREFIXES_TO_MAX_INPUT_TOKENS.
↪items()
            if model_prefix in self.model
        ),
        None,
    )

    if tokens is None:
        logging.warning(
            "Model %s not found in MODEL_PREFIXES_TO_MAX_INPUT_TOKENS, using default␣
↪value of %s.",
            self.model,
            self.DEFAULT_MAX_INPUT_TOKENS,
        )
        return self.DEFAULT_MAX_INPUT_TOKENS
    return tokens
```

_default_max_output_tokens()

Source Code in griptape/tokenizers/base_tokenizer.py

```
def _default_max_output_tokens(self) -> int:
    tokens = next(
        (
            max_tokens
            for model_prefix, max_tokens in self.MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS.
↪items()
            if model_prefix in self.model
        ),
        None,
    )

    if tokens is None:
        logging.debug(
            "Model %s not found in MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS, using default␣
↪value of %s.",
            self.model,
            self.DEFAULT_MAX_OUTPUT_TOKENS,
        )
        return self.DEFAULT_MAX_OUTPUT_TOKENS
    return tokens
```

count_input_tokens_left(text)

Source Code in griptape/tokenizers/base_tokenizer.py

```
def count_input_tokens_left(self, text: str) -> int:
    diff = self.max_input_tokens - self.count_tokens(text)

    if diff > 0:
        return diff
    return 0
```

count_output_tokens_left(text)

Source Code in griptape/tokenizers/base_tokenizer.py

```
def count_output_tokens_left(self, text: str) -> int:
    diff = self.max_output_tokens - self.count_tokens(text)

    if diff > 0:
        return diff
    return 0
```

count_tokens(text)abstractmethod

Source Code in griptape/tokenizers/base_tokenizer.py

```
@abstractmethod
def count_tokens(self, text: str) -> int: ...
```

max_input_tokens()

Source Code in griptape/tokenizers/base_tokenizer.py

```
@lazy_property()
def max_input_tokens(self) -> int:
    return self._default_max_input_tokens()
```

max_output_tokens()

Source Code in griptape/tokenizers/base_tokenizer.py

```
@lazy_property()
def max_output_tokens(self) -> int:
    return self._default_max_output_tokens()
```

cohere_tokenizer

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/cohere_tokenizer.py

```
@define()
class CohereTokenizer(BaseTokenizer):
    MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {"command-r": 128000, "command": 4096, "embed
↪": 512}
    MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {"command": 4096, "embed": 512}

    client: Client = field(kw_only=True)

    def count_tokens(self, text: str) -> int:
        return len(self.client.tokenize(text=text, model=self.model).tokens)
```

- `MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {'command-r': 128000, 'command': 4096, 'embed': 512}` class-attribute instance-attribute
- `MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {'command': 4096, 'embed': 512}` class-attribute instance-attribute
- `client = field(kw_only=True)` class-attribute instance-attribute

count_tokens(text)

Source Code in griptape/tokenizers/cohere_tokenizer.py

```
def count_tokens(self, text: str) -> int:
    return len(self.client.tokenize(text=text, model=self.model).tokens)
```

dummy_tokenizer

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/dummy_tokenizer.py

```
@define
class DummyTokenizer(BaseTokenizer):
    model: Optional[str] = field(default=None, kw_only=True)
    _max_input_tokens: int = field(init=False, default=0, kw_only=True, alias="max_
↪input_tokens")
    _max_output_tokens: int = field(init=False, default=0, kw_only=True, alias="max_
↪output_tokens")

    def count_tokens(self, text: str) -> int:
        raise DummyError(__class__.__name__, "count_tokens")
```

- `_max_input_tokens = field(init=False, default=0, kw_only=True, alias='max_input_tokens')` class-attribute instance-attribute
- `_max_output_tokens = field(init=False, default=0, kw_only=True, alias='max_output_tokens')` class-attribute instance-attribute
- `model = field(default=None, kw_only=True)` class-attribute instance-attribute

count_tokens(text)

Source Code in griptape/tokenizers/dummy_tokenizer.py

```
def count_tokens(self, text: str) -> int:
    raise DummyError(__class__.__name__, "count_tokens")
```

google_tokenizer

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/google_tokenizer.py

```
@define()
class GoogleTokenizer(BaseTokenizer):
    MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {"gemini-1.5-pro": 2097152, "gemini":␣
↪1048576}
    MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {"gemini": 8192}

    api_key: str = field(kw_only=True, metadata={"serializable": True})
    _client: Optional[GenerativeModel] = field(
        default=None, kw_only=True, alias="client", metadata={"serializable": False}
    )

    @lazy_property()
    def client(self) -> GenerativeModel:
        genai = import_optional_dependency("google.generativeai")
        genai.configure(api_key=self.api_key)

        return genai.GenerativeModel(self.model)

    def count_tokens(self, text: str) -> int:
        return self.client.count_tokens(text).total_tokens
```

- `MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {'gemini-1.5-pro': 2097152, 'gemini': 1048576}` class-attribute instance-attribute

- `MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {'gemini': 8192}` class-attribute instance-attribute
- `_client = field(default=None, kw_only=True, alias='client', metadata={'serializable': False})` class-attribute instance-attribute
- `api_key = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

client()

Source Code in griptape/tokenizers/google_tokenizer.py

```
@lazy_property()
def client(self) -> GenerativeModel:
    genai = import_optional_dependency("google.generativeai")
    genai.configure(api_key=self.api_key)

    return genai.GenerativeModel(self.model)
```

count_tokens(text)

Source Code in griptape/tokenizers/google_tokenizer.py

```
def count_tokens(self, text: str) -> int:
    return self.client.count_tokens(text).total_tokens
```

grok_tokenizer

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/grok_tokenizer.py

```
@define()
class GrokTokenizer(BaseTokenizer):
    # https://docs.x.ai/docs/models?cluster=us-east-1#model-constraints
    MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {
        "grok-2-vision": 32768,
        "grok-2": 131072,
        "grok-vision-beta": 8192,
        "grok-beta": 131072,
    }

    MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {"grok": 4096}

    base_url: str = field(default="https://api.x.ai", kw_only=True, metadata={
↪"serializable": True})
    api_key: str = field(kw_only=True, default=None)
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True), kw_only=True
    )

    def count_tokens(self, text: str) -> int:
        response = requests.post(
```

(continues on next page)

(continued from previous page)

```
            urljoin(self.base_url, "/v1/tokenize-text"),
            headers=self.headers,
            json={"text": text, "model": self.model},
        )
        response.raise_for_status()
        return len(response.json()["token_ids"])
```

- `MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {'grok-2-vision': 32768, 'grok-2': 131072, 'grok-vision-beta': 8192, 'grok-beta': 131072}` class-attribute instance-attribute
- `MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {'grok': 4096}` class-attribute instance-attribute
- `api_key = field(kw_only=True, default=None)` class-attribute instance-attribute
- `base_url = field(default='https://api.x.ai', kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute

count_tokens(text)

Source Code in griptape/tokenizers/grok_tokenizer.py

```
def count_tokens(self, text: str) -> int:
    response = requests.post(
        urljoin(self.base_url, "/v1/tokenize-text"),
        headers=self.headers,
        json={"text": text, "model": self.model},
    )
    response.raise_for_status()
    return len(response.json()["token_ids"])
```

huggingface_tokenizer

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/huggingface_tokenizer.py

```
@define()
class HuggingFaceTokenizer(BaseTokenizer):
    tokenizer: PreTrainedTokenizerBase = field(
        default=Factory(
            lambda self: import_optional_dependency("transformers").AutoTokenizer.
↪from_pretrained(self.model),
            takes_self=True,
        ),
        kw_only=True,
    )
    _max_input_tokens: int = field(
        default=Factory(lambda self: self.tokenizer.model_max_length, takes_
↪self=True),
        kw_only=True,
        alias="max_input_tokens",
```

(continues on next page)

(continued from previous page)

```
    )
    _max_output_tokens: int = field(default=4096, kw_only=True, alias="max_output_
↪tokens")

    def count_tokens(self, text: str) -> int:
        return len(self.tokenizer.encode(text))  # pyright: ignore[reportArgumentType]
```

- `_max_input_tokens = field(default=Factory(lambda self: self.tokenizer.model_max_length, takes_self=True), kw_only=True, alias='max_input_tokens')`
  class-attribute instance-attribute
- `_max_output_tokens = field(default=4096, kw_only=True, alias='max_output_tokens')`
  class-attribute instance-attribute
- `tokenizer = field(default=Factory(lambda self: import_optional_dependency('transformers').AutoTokenizer.from_pretrained(self.model), takes_self=True), kw_only=True)`
  class-attribute instance-attribute

count_tokens(text)

Source Code in griptape/tokenizers/huggingface_tokenizer.py

```
def count_tokens(self, text: str) -> int:
    return len(self.tokenizer.encode(text))  # pyright: ignore[reportArgumentType]
```

openai_tokenizer

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/openai_tokenizer.py

```
@define()
class OpenAiTokenizer(BaseTokenizer):
    DEFAULT_OPENAI_GPT_3_COMPLETION_MODEL = "gpt-3.5-turbo-instruct"
    DEFAULT_OPENAI_GPT_3_CHAT_MODEL = "gpt-3.5-turbo"
    DEFAULT_OPENAI_GPT_4_MODEL = "gpt-4o"
    DEFAULT_ENCODING = "cl100k_base"
    DEFAULT_MAX_TOKENS = 2049
    DEFAULT_MAX_OUTPUT_TOKENS = 4096
    TOKEN_OFFSET = 8

    # https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo
    MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {
        "gpt-4.1": 1_000_000,
        "gpt-4o": 128000,
        "gpt-4-1106": 128000,
        "gpt-4-32k": 32768,
        "gpt-4": 8192,
        "gpt-3.5-turbo-16k": 16384,
        "gpt-3.5-turbo": 4096,
        "gpt-35-turbo-16k": 16384,
        "gpt-35-turbo": 4096,
        "text-embedding-ada-002": 8191,
```

(continues on next page)

(continued from previous page)

```
        "text-embedding-ada-001": 2046,
        "text-embedding-3-small": 8191,
        "text-embedding-3-large": 8191,
    }

    MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {"gpt": 4096}

    EMBEDDING_MODELS = [
        "text-embedding-ada-002",
        "text-embedding-ada-001",
        "text-embedding-3-small",
        "text-embedding-3-large",
    ]

    _max_input_tokens: int = field(
        kw_only=True,
        default=Factory(lambda self: self._default_max_input_tokens(), takes_
↪self=True),
        alias="max_input_tokens",
    )
    _max_output_tokens: int = field(
        kw_only=True,
        default=Factory(lambda self: self._default_max_output_tokens(), takes_
↪self=True),
        alias="max_output_tokens",
    )

    @property
    def encoding(self) -> tiktoken.Encoding:
        try:
            return tiktoken.encoding_for_model(self.model)
        except KeyError:
            return tiktoken.get_encoding(self.DEFAULT_ENCODING)

    def _default_max_input_tokens(self) -> int:
        tokens = next((v for k, v in self.MODEL_PREFIXES_TO_MAX_INPUT_TOKENS.items()␣
↪if self.model.startswith(k)), None)
        offset = 0 if self.model in self.EMBEDDING_MODELS else self.TOKEN_OFFSET

        return (tokens or self.DEFAULT_MAX_TOKENS) - offset

    def _default_max_output_tokens(self) -> int:
        tokens = next(
            (v for k, v in self.MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS.items() if self.
↪model.startswith(k)),
            None,
        )

        if tokens is None:
            return self.DEFAULT_MAX_OUTPUT_TOKENS
        return tokens
```

(continues on next page)

(continued from previous page)

```
    def count_tokens(self, text: str | list[dict], model: Optional[str] = None) ->␣
↪int:  # noqa: C901
        """Handles the special case of ChatML.

        Implementation adopted from the official OpenAI notebook:
        https://github.com/openai/openai-cookbook/blob/main/examples/How_to_count_
↪tokens_with_tiktoken.ipynb.
        """
        if isinstance(text, list):
            model = model or self.model

            try:
                encoding = tiktoken.encoding_for_model(model)
            except KeyError:
                logging.warning("model not found. Using cl100k_base encoding.")

                encoding = tiktoken.get_encoding("cl100k_base")

            if model in {
                "gpt-3.5-turbo-0613",
                "gpt-3.5-turbo-16k-0613",
                "gpt-4-0314",
                "gpt-4-32k-0314",
                "gpt-4-0613",
                "gpt-4-32k-0613",
                "gpt-4o-2024-05-13",
            }:
                tokens_per_message = 3
                tokens_per_name = 1
            elif model == "gpt-3.5-turbo-0301":
                # every message follows <|start|>{role/name}\n{content}<|end|>\n
                tokens_per_message = 4
                # if theres a name, the role is omitted
                tokens_per_name = -1
            elif "gpt-3.5-turbo" in model or "gpt-35-turbo" in model:
                logging.info("gpt-3.5-turbo may update over time. Returning num␣
↪tokens assuming gpt-3.5-turbo-0613.")
                return self.count_tokens(text, model="gpt-3.5-turbo-0613")
            elif "gpt-4o" in model:
                logging.info("gpt-4o may update over time. Returning num tokens␣
↪assuming gpt-4o-2024-05-13.")
                return self.count_tokens(text, model="gpt-4o-2024-05-13")
            elif "gpt-4" in model:
                logging.info("gpt-4 may update over time. Returning num tokens␣
↪assuming gpt-4-0613.")
                return self.count_tokens(text, model="gpt-4-0613")
            else:
                raise NotImplementedError(
                    f"token_count() is not implemented for model {model}. "
                    "See https://github.com/openai/openai-python/blob/main/chatml.md␣
↪for "
                    "information on how messages are converted to tokens."
```

(continues on next page)

(continued from previous page)

```
                )

            num_tokens = 0

            for message in text:
                num_tokens += tokens_per_message
                for key, value in message.items():
                    num_tokens += len(encoding.encode(value))
                    if key == "name":
                        num_tokens += tokens_per_name

            # every reply is primed with <|start|>assistant<|message|>
            num_tokens += 3

            return num_tokens
        return len(self.encoding.encode(text, allowed_special=set(self.stop_
↪sequences)))
```

- `DEFAULT_ENCODING = 'cl100k_base'` class-attribute instance-attribute
- `DEFAULT_MAX_OUTPUT_TOKENS = 4096` class-attribute instance-attribute
- `DEFAULT_MAX_TOKENS = 2049` class-attribute instance-attribute
- `DEFAULT_OPENAI_GPT_3_CHAT_MODEL = 'gpt-3.5-turbo'` class-attribute instance-attribute
- `DEFAULT_OPENAI_GPT_3_COMPLETION_MODEL = 'gpt-3.5-turbo-instruct'` class-attribute instance-attribute
- `DEFAULT_OPENAI_GPT_4_MODEL = 'gpt-4o'` class-attribute instance-attribute
- `EMBEDDING_MODELS = ['text-embedding-ada-002', 'text-embedding-ada-001', 'text-embedding-3-small', 'text-embedding-3-large']` class-attribute instance-attribute
- `MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {'gpt-4.1': 1000000, 'gpt-4o': 128000, 'gpt-4-1106': 128000, 'gpt-4-32k': 32768, 'gpt-4': 8192, 'gpt-3.5-turbo-16k': 16384, 'gpt-3.5-turbo': 4096, 'gpt-35-turbo-16k': 16384, 'gpt-35-turbo': 4096, 'text-embedding-ada-002': 8191, 'text-embedding-ada-001': 2046, 'text-embedding-3-small': 8191, 'text-embedding-3-large': 8191}` class-attribute instance-attribute
- `MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {'gpt': 4096}` class-attribute instance-attribute
- `TOKEN_OFFSET = 8` class-attribute instance-attribute
- `_max_input_tokens = field(kw_only=True, default=Factory(lambda self: self._default_max_input_tokens(), takes_self=True), alias='max_input_tokens')` class-attribute instance-attribute
- `_max_output_tokens = field(kw_only=True, default=Factory(lambda self: self._default_max_output_tokens(), takes_self=True), alias='max_output_tokens')` class-attribute instance-attribute
- `encoding` property

_default_max_input_tokens()

Source Code in griptape/tokenizers/openai_tokenizer.py

```
def _default_max_input_tokens(self) -> int:
    tokens = next((v for k, v in self.MODEL_PREFIXES_TO_MAX_INPUT_TOKENS.items() if␣
↪self.model.startswith(k)), None)
    offset = 0 if self.model in self.EMBEDDING_MODELS else self.TOKEN_OFFSET

    return (tokens or self.DEFAULT_MAX_TOKENS) - offset
```

_default_max_output_tokens()

Source Code in griptape/tokenizers/openai_tokenizer.py

```
def _default_max_output_tokens(self) -> int:
    tokens = next(
        (v for k, v in self.MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS.items() if self.model.
↪startswith(k)),
        None,
    )

    if tokens is None:
        return self.DEFAULT_MAX_OUTPUT_TOKENS
    return tokens
```

count_tokens(text, model=None)

Source Code in griptape/tokenizers/openai_tokenizer.py

```
def count_tokens(self, text: str | list[dict], model: Optional[str] = None) -> int:
↪# noqa: C901
    """Handles the special case of ChatML.

    Implementation adopted from the official OpenAI notebook:
    https://github.com/openai/openai-cookbook/blob/main/examples/How_to_count_tokens_
↪with_tiktoken.ipynb.
    """
    if isinstance(text, list):
        model = model or self.model

        try:
            encoding = tiktoken.encoding_for_model(model)
        except KeyError:
            logging.warning("model not found. Using cl100k_base encoding.")

            encoding = tiktoken.get_encoding("cl100k_base")

        if model in {
            "gpt-3.5-turbo-0613",
            "gpt-3.5-turbo-16k-0613",
            "gpt-4-0314",
            "gpt-4-32k-0314",
```

(continues on next page)

(continued from previous page)

```
            "gpt-4-0613",
            "gpt-4-32k-0613",
            "gpt-4o-2024-05-13",
        }:
            tokens_per_message = 3
            tokens_per_name = 1
        elif model == "gpt-3.5-turbo-0301":
            # every message follows <|start|>{role/name}\n{content}<|end|>\n
            tokens_per_message = 4
            # if theres a name, the role is omitted
            tokens_per_name = -1
        elif "gpt-3.5-turbo" in model or "gpt-35-turbo" in model:
            logging.info("gpt-3.5-turbo may update over time. Returning num tokens␣
↪assuming gpt-3.5-turbo-0613.")
            return self.count_tokens(text, model="gpt-3.5-turbo-0613")
        elif "gpt-4o" in model:
            logging.info("gpt-4o may update over time. Returning num tokens assuming␣
↪gpt-4o-2024-05-13.")
            return self.count_tokens(text, model="gpt-4o-2024-05-13")
        elif "gpt-4" in model:
            logging.info("gpt-4 may update over time. Returning num tokens assuming␣
↪gpt-4-0613.")
            return self.count_tokens(text, model="gpt-4-0613")
        else:
            raise NotImplementedError(
                f"token_count() is not implemented for model {model}. "
                "See https://github.com/openai/openai-python/blob/main/chatml.md for "
                "information on how messages are converted to tokens."
            )

        num_tokens = 0

        for message in text:
            num_tokens += tokens_per_message
            for key, value in message.items():
                num_tokens += len(encoding.encode(value))
                if key == "name":
                    num_tokens += tokens_per_name

        # every reply is primed with <|start|>assistant<|message|>
        num_tokens += 3

        return num_tokens
    return len(self.encoding.encode(text, allowed_special=set(self.stop_sequences)))
```

simple_tokenizer

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/simple_tokenizer.py

```
@define()
```

(continues on next page)

(continued from previous page)

```
class SimpleTokenizer(BaseTokenizer):
    model: str | None = field(init=False, default=None, kw_only=True)
    characters_per_token: int = field(kw_only=True)

    def count_tokens(self, text: str) -> int:
        return (len(text) + self.characters_per_token - 1) // self.characters_per_
↪token
```

- `characters_per_token = field(kw_only=True)` class-attribute instance-attribute
- `model = field(init=False, default=None, kw_only=True)` class-attribute instance-attribute

count_tokens(text)

Source Code in griptape/tokenizers/simple_tokenizer.py

```
def count_tokens(self, text: str) -> int:
    return (len(text) + self.characters_per_token - 1) // self.characters_per_token
```

voyageai_tokenizer

Bases: *BaseTokenizer* “)

Source Code in griptape/tokenizers/voyageai_tokenizer.py

```
@define()
class VoyageAiTokenizer(BaseTokenizer):
    MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {
        "voyage-3": 32000,
        "voyage-large-2": 16000,
        "voyage-code-2": 16000,
        "voyage-2": 4000,
        "voyage-lite-02-instruct": 4000,
    }
    MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {"voyage": 0}

    api_key: Optional[str] = field(default=None, kw_only=True, metadata={"serializable
↪": False})
    client: Client = field(
        default=Factory(
            lambda self: import_optional_dependency("voyageai").Client(api_key=self.
↪api_key),
            takes_self=True,
        ),
        kw_only=True,
    )

    def count_tokens(self, text: str) -> int:
        return self.client.count_tokens([text], model=self.model)
```

- `MODEL_PREFIXES_TO_MAX_INPUT_TOKENS = {'voyage-3': 32000, 'voyage-large-2': 16000, 'voyage-code-2': 16000, 'voyage-2': 4000, 'voyage-lite-02-instruct': 4000}` class-attribute instance-attribute

- `MODEL_PREFIXES_TO_MAX_OUTPUT_TOKENS = {'voyage': 0}` class-attribute instance-attribute
- `api_key = field(default=None, kw_only=True, metadata={'serializable': False})` class-attribute instance-attribute
- `client = field(default=Factory(lambda self: import_optional_dependency('voyageai').Client(api_key=self.api_key), takes_self=True), kw_only=True)` class-attribute instance-attribute

count_tokens(text)

Source Code in griptape/tokenizers/voyageai_tokenizer.py

```
def count_tokens(self, text: str) -> int:
    return self.client.count_tokens([text], model=self.model)
```

11.5.20 tools

- `__all__ = ['AudioTranscriptionTool', 'BaseImageGenerationTool', 'BaseTool', 'CalculatorTool', 'ComputerTool', 'DateTimeTool', 'EmailTool', 'ExtractionTool', 'FileManagerTool', 'GriptapeCloudToolTool', 'ImageQueryTool', 'InpaintingImageGenerationTool', 'OutpaintingImageGenerationTool', 'PromptImageGenerationTool', 'PromptSummaryTool', 'QueryTool', 'RagTool', 'RestApiTool', 'SqlTool', 'StructureRunTool', 'StructuredOutputTool', 'TextToSpeechTool', 'VariationImageGenerationTool', 'VectorStoreTool', 'WebScraperTool', 'WebSearchTool']` module-attribute

Bases: *BaseTool* “)

Source Code in griptape/tools/audio_transcription/tool.py

```
@define
class AudioTranscriptionTool(BaseTool):
    """A tool that can be used to generate transcriptions from input audio."""

    audio_transcription_driver: BaseAudioTranscriptionDriver = field(kw_only=True)
    audio_loader: AudioLoader = field(default=Factory(lambda: AudioLoader()), kw_
↪only=True)

    @activity(
        config={
            "description": "This tool can be used to generate transcriptions of audio␣
↪files on disk.",
            "schema": Schema({Literal("path", description="The paths to an audio file␣
↪on disk."): str}),
        },
    )
    def transcribe_audio_from_disk(self, params: dict) -> TextArtifact |␣
↪ErrorArtifact:
        audio_path = params["values"]["path"]

        audio_artifact = self.audio_loader.load(audio_path)

        return self.audio_transcription_driver.run(audio_artifact)
```

(continues on next page)

(continued from previous page)

```
    @activity(
        config={
            "description": "This tool can be used to generate the transcription of an␣
↪audio artifact in memory.",
            "schema": Schema({"schema": Schema({"memory_name": str, "artifact_
↪namespace": str, "artifact_name": str})}),
        },
    )
    def transcribe_audio_from_memory(self, params: dict[str, Any]) -> TextArtifact |␣
↪ErrorArtifact:
        memory = self.find_input_memory(params["values"]["memory_name"])
        artifact_namespace = params["values"]["artifact_namespace"]
        artifact_name = params["values"]["artifact_name"]

        if memory is None:
            return ErrorArtifact("memory not found")

        audio_artifact = cast(
            "AudioArtifact",
            load_artifact_from_memory(memory, artifact_namespace, artifact_name,␣
↪AudioArtifact),
        )

        return self.audio_transcription_driver.run(audio_artifact)
```

- `audio_loader = field(default=Factory(lambda: AudioLoader()), kw_only=True)` class-attribute instance-attribute
- `audio_transcription_driver = field(kw_only=True)` class-attribute instance-attribute

transcribe_audio_from_disk(params)

Source Code in griptape/tools/audio_transcription/tool.py

```
@activity(
    config={
        "description": "This tool can be used to generate transcriptions of audio␣
↪files on disk.",
        "schema": Schema({Literal("path", description="The paths to an audio file on␣
↪disk."): str}),
    },
)
def transcribe_audio_from_disk(self, params: dict) -> TextArtifact | ErrorArtifact:
    audio_path = params["values"]["path"]

    audio_artifact = self.audio_loader.load(audio_path)

    return self.audio_transcription_driver.run(audio_artifact)
```

transcribe_audio_from_memory(params)

Source Code in griptape/tools/audio_transcription/tool.py

```
@activity(
    config={
        "description": "This tool can be used to generate the transcription of an␣
↪audio artifact in memory.",
        "schema": Schema({"schema": Schema({"memory_name": str, "artifact_namespace":␣
↪str, "artifact_name": str})}),
    },
)
def transcribe_audio_from_memory(self, params: dict[str, Any]) -> TextArtifact |␣
↪ErrorArtifact:
    memory = self.find_input_memory(params["values"]["memory_name"])
    artifact_namespace = params["values"]["artifact_namespace"]
    artifact_name = params["values"]["artifact_name"]

    if memory is None:
        return ErrorArtifact("memory not found")

    audio_artifact = cast(
        "AudioArtifact",
        load_artifact_from_memory(memory, artifact_namespace, artifact_name,␣
↪AudioArtifact),
    )

    return self.audio_transcription_driver.run(audio_artifact)
```

BaseImageGenerationTool

Bases: *ArtifactFileOutputMixin* “) , *BaseTool* ”)

Source Code in griptape/tools/base_image_generation_tool.py

```
@define
class BaseImageGenerationTool(ArtifactFileOutputMixin, BaseTool):
    """A base class for tools that generate images from text prompts."""

    PROMPT_DESCRIPTION = "Features and qualities to include in the generated image,␣
↪descriptive and succinct."
    NEGATIVE_PROMPT_DESCRIPTION = (
        "Features and qualities to avoid in the generated image. Affirmatively␣
↪describe "
        "what to avoid, for example: to avoid the color red, include red "
        "rather than no red."
    )
```

- `NEGATIVE_PROMPT_DESCRIPTION = "Features and qualities to avoid in the generated image. Affirmatively describe what to avoid, for example: to avoid the color red, include 'red' rather than 'no red'."` class-attribute instance-attribute
- `PROMPT_DESCRIPTION = 'Features and qualities to include in the generated image, descriptive and succinct.'` class-attribute instance-attribute

BaseTool

Bases: *ActivityMixin* “) , *SerializableMixin* ”) , *RunnableMixin['BaseTool'* “) , ABC

Attributes

Name	Type	Description
<code class="language-python">	*str*	Tool name.
<code class="language-python">	<code class="language-python">Optional[list[TaskMemory]]	TaskMemory available in tool activities. Gets automatically set if None.
<code class="language-python">	<code class="language-python">Optional[dict[str, list[TaskMemory]]]</code>	TaskMemory that activities write to be default. Gets automatically set if None.

continues on next page

Table 381 – continued from previous page

Name	Type	Description
<a href="./#griptape.tools.BaseTool.install_dependencies_on_init" title="install_dependencies_on_init = field(default=True, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.tools.BaseTool.install_dependencies_on_init)"><code class="language-python">	*bool*	Determines whether dependencies from the tool requirements.txt file are installed in init.
<a href="./#griptape.tools.BaseTool.dependencies_install_directory" title="dependencies_install_directory = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.tools.BaseTool.dependencies_install_directory)"><code class="language-python">	*Optional[str]*	Custom dependency install directory.
<a href="./#griptape.tools.BaseTool.verbose" title="verbose = field(default=False, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.tools.BaseTool.verbose)"><code class="language-python">	*bool*	Determines whether tool operations (such as dependency installation) should be verbose.

continues on next page

Table 381 – continued from previous page

Name	Type	Description
<a href="./#griptape.tools.BaseTool.off_prompt" title="off_prompt = field(default=False, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.tools.BaseTool.off_prompt)"><code class="language-python">	*bool*	Determines whether tool activity output goes to the output memory.

Source Code in griptape/tools/base_tool.py

```
@define
class BaseTool(ActivityMixin, SerializableMixin, RunnableMixin["BaseTool"], ABC):
    """Abstract class for all tools to inherit from for.

    Attributes:
        name: Tool name.
        input_memory: TaskMemory available in tool activities. Gets automatically set␣
↪if None.
        output_memory: TaskMemory that activities write to be default. Gets␣
↪automatically set if None.
        install_dependencies_on_init: Determines whether dependencies from the tool␣
↪requirements.txt file are installed in init.
        dependencies_install_directory: Custom dependency install directory.
        verbose: Determines whether tool operations (such as dependency installation)␣
↪should be verbose.
        off_prompt: Determines whether tool activity output goes to the output memory.
    """

    REQUIREMENTS_FILE = "requirements.txt"

    name: str = field(
        default=Factory(lambda self: self.__class__.__name__, takes_self=True),
        kw_only=True,
        metadata={"serializable": True},
    )
    input_memory: Optional[list[TaskMemory]] = field(default=None, kw_only=True,␣
↪metadata={"serializable": True})
    output_memory: Optional[dict[str, list[TaskMemory]]] = field(
        default=None, kw_only=True, metadata={"serializable": True}
    )
    install_dependencies_on_init: bool = field(default=True, kw_only=True, metadata={
↪"serializable": True})
    dependencies_install_directory: Optional[str] = field(default=None, kw_only=True,␣
↪metadata={"serializable": True})
    verbose: bool = field(default=False, kw_only=True, metadata={"serializable": True}
```

(continues on next page)

(continued from previous page)

```
↪)
    off_prompt: bool = field(default=False, kw_only=True, metadata={"serializable":␣
↪True})

    def __attrs_post_init__(self) -> None:
        if (
            self.install_dependencies_on_init
            and self.has_requirements
            and not self.are_requirements_met(self.requirements_path)
        ):
            self.install_dependencies(os.environ.copy())

    @output_memory.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_output_memory(self, _: Attribute, output_memory: dict[str,␣
↪Optional[list[TaskMemory]]]) -> None:
        if output_memory:
            for activity_name, memory_list in output_memory.items():
                if not self.find_activity(activity_name):
                    raise ValueError(f"activity {activity_name} doesnt exist")
                if memory_list is None:
                    raise ValueError(f"memory list for activity {activity_name} cant␣
↪be None")

                output_memory_names = [memory.name for memory in memory_list]

                if len(output_memory_names) > len(set(output_memory_names)):
                    raise ValueError(f"memory names have to be unique in activity
↪{activity_name} output")

    @property
    def requirements_path(self) -> str:
        return os.path.join(self.abs_dir_path, self.REQUIREMENTS_FILE)

    @property
    def abs_file_path(self) -> str:
        return os.path.abspath(inspect.getfile(self.__class__))

    @property
    def abs_dir_path(self) -> str:
        return os.path.dirname(self.abs_file_path)

    @property
    def has_requirements(self) -> bool:
        return os.path.exists(self.requirements_path)

    # This method has to remain a method and cant be decorated with @property because
    # of the max depth recursion issue in `self.activities`.
    def schema(self) -> dict:
        full_schema = Schema(Or(*self.activity_schemas()), description=f"{self.name}␣
↪action schema.")
```

(continues on next page)

(continued from previous page)

```
        return full_schema.json_schema(f"{self.name} ToolAction Schema")

    def activity_schemas(self) -> list[Schema]:
        schemas = []

        for activity in self.activities():
            schema_dict: dict[Literal | schema.Optional, Any] = {
                Literal("name"): self.name,
                Literal("path", description=self.activity_description(activity)):␣
↪self.activity_name(activity),
            }

            activity_schema = self.activity_schema(activity)
            # If no schema is defined, we just make `input` optional instead of␣
↪omitting it.
            # This works better with lower-end models that may accidentally pass in␣
↪an empty dict.
            if activity_schema is None:
                schema_dict[schema.Optional("input")] = {}
            else:
                schema_dict[Literal("input")] = activity_schema.schema

            schemas.append(Schema(schema_dict))

        return schemas

    def run(self, activity: Callable, subtask: ActionsSubtask, action: ToolAction) ->␣
↪BaseArtifact:
        try:
            output = self.before_run(activity, subtask, action)

            output = self.try_run(activity, subtask, action, output)

            output = self.after_run(activity, subtask, action, output)
        except Exception as e:
            logging.debug(traceback.format_exc())
            output = ErrorArtifact(str(e), exception=e)

        return output

    def before_run(self, activity: Callable, subtask: ActionsSubtask, action:␣
↪ToolAction) -> Optional[dict]:
        super().before_run()

        return action.input

    @observable(tags=["Tool.run()"])
    def try_run(
        self,
        activity: Callable,
        subtask: ActionsSubtask,
        action: ToolAction,
```

(continues on next page)

(continued from previous page)

```
        value: Optional[dict],
    ) -> BaseArtifact:
        activity_result = activity(deepcopy(value))

        if isinstance(activity_result, BaseArtifact):
            result = activity_result
        else:
            logging.warning("Activity result is not an artifact; converting result to␣
↪InfoArtifact")

            if activity_result is None:
                result = InfoArtifact("Tool returned an empty value")
            else:
                result = InfoArtifact(activity_result)

        return result

    def after_run(
        self,
        activity: Callable,
        subtask: ActionsSubtask,
        action: ToolAction,
        value: BaseArtifact,
    ) -> BaseArtifact:
        super().after_run()

        if self.output_memory:
            output_memories = self.output_memory[getattr(activity, "name")] or []
            for memory in output_memories:
                value = memory.process_output(activity, subtask, value)

            return value
        return value

    def validate(self) -> bool:
        if not os.path.exists(self.requirements_path):
            raise Exception(f"{self.REQUIREMENTS_FILE} not found")
        return True

    def tool_dir(self) -> str:
        class_file = inspect.getfile(self.__class__)

        return os.path.dirname(os.path.abspath(class_file))

    def install_dependencies(self, env: Optional[dict[str, str]] = None) -> None:
        env = env or {}

        command = [sys.executable, "-m", "pip", "install", "-r", "requirements.txt"]

        if self.dependencies_install_directory is None:
            command.extend(["-U"])
        else:
```

(continues on next page)

(continued from previous page)

```
            command.extend(["-t", self.dependencies_install_directory])

        subprocess.run(
            command,
            env=env,
            cwd=self.tool_dir(),
            stdout=None if self.verbose else subprocess.DEVNULL,
            stderr=None if self.verbose else subprocess.DEVNULL,
            check=False,
        )

    def find_input_memory(self, memory_name: str) -> Optional[TaskMemory]:
        if self.input_memory:
            return next((m for m in self.input_memory if m.name == memory_name), None)
        return None

    def to_native_tool_name(self, activity: Callable) -> str:
        """Converts a Tools name and an Activity into to a native tool name.

        The native tool name is a combination of the Tools name and the Activitys␣
↪name.
        The Tools name may only contain letters and numbers, and the Activitys name␣
↪may only contain letters, numbers, and underscores.

        Args:
            activity: Activity to convert

        Returns:
            str: Native tool name.
        """
        tool_name = self.name
        if re.match(r"^[a-zA-Z0-9]+$", tool_name) is None:
            raise ValueError("Tool name can only contain letters and numbers.")

        activity_name = self.activity_name(activity)
        if re.match(r"^[a-zA-Z0-9_]+$", activity_name) is None:
            raise ValueError("Activity name can only contain letters, numbers, and␣
↪underscores.")

        return f"{tool_name}_{activity_name}"

    def are_requirements_met(self, requirements_path: str) -> bool:
        requirements = Path(requirements_path).read_text().splitlines()

        try:
            for requirement in requirements:
                importlib.metadata.version(requirement)
            return True
        except importlib.metadata.PackageNotFoundError:
            return False
```

- `REQUIREMENTS_FILE = 'requirements.txt'` class-attribute instance-attribute

- `abs_dir_path` property
- `abs_file_path` property
- `dependencies_install_directory = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `has_requirements` property
- `input_memory = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `install_dependencies_on_init = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `name = field(default=Factory(lambda self: self.__class__.__name__, takes_self=True), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `off_prompt = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output_memory = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `requirements_path` property
- `verbose = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/tools/base_tool.py

```
def __attrs_post_init__(self) -> None:
    if (
        self.install_dependencies_on_init
        and self.has_requirements
        and not self.are_requirements_met(self.requirements_path)
    ):
        self.install_dependencies(os.environ.copy())
```

activity_schemas()

Source Code in griptape/tools/base_tool.py

```
def activity_schemas(self) -> list[Schema]:
    schemas = []

    for activity in self.activities():
        schema_dict: dict[Literal | schema.Optional, Any] = {
            Literal("name"): self.name,
            Literal("path", description=self.activity_description(activity)): self.
↪activity_name(activity),
        }

        activity_schema = self.activity_schema(activity)
        # If no schema is defined, we just make `input` optional instead of omitting␣
```

(continues on next page)

(continued from previous page)

```
↪it.
        # This works better with lower-end models that may accidentally pass in an␣
↪empty dict.
        if activity_schema is None:
            schema_dict[schema.Optional("input")] = {}
        else:
            schema_dict[Literal("input")] = activity_schema.schema

        schemas.append(Schema(schema_dict))

    return schemas
```

after_run(activity, subtask, action, value)

Source Code in griptape/tools/base_tool.py

```
def after_run(
    self,
    activity: Callable,
    subtask: ActionsSubtask,
    action: ToolAction,
    value: BaseArtifact,
) -> BaseArtifact:
    super().after_run()

    if self.output_memory:
        output_memories = self.output_memory[getattr(activity, "name")] or []
        for memory in output_memories:
            value = memory.process_output(activity, subtask, value)

        return value
    return value
```

are_requirements_met(requirements_path)

Source Code in griptape/tools/base_tool.py

```
def are_requirements_met(self, requirements_path: str) -> bool:
    requirements = Path(requirements_path).read_text().splitlines()

    try:
        for requirement in requirements:
            importlib.metadata.version(requirement)
        return True
    except importlib.metadata.PackageNotFoundError:
        return False
```

before_run(activity, subtask, action)

Source Code in griptape/tools/base_tool.py

```
def before_run(self, activity: Callable, subtask: ActionsSubtask, action: ToolAction)␣
↪-> Optional[dict]:
    super().before_run()

    return action.input
```

find_input_memory(memory_name)

Source Code in griptape/tools/base_tool.py

```
def find_input_memory(self, memory_name: str) -> Optional[TaskMemory]:
    if self.input_memory:
        return next((m for m in self.input_memory if m.name == memory_name), None)
    return None
```

install_dependencies(env=None)

Source Code in griptape/tools/base_tool.py

```
def install_dependencies(self, env: Optional[dict[str, str]] = None) -> None:
    env = env or {}

    command = [sys.executable, "-m", "pip", "install", "-r", "requirements.txt"]

    if self.dependencies_install_directory is None:
        command.extend(["-U"])
    else:
        command.extend(["-t", self.dependencies_install_directory])

    subprocess.run(
        command,
        env=env,
        cwd=self.tool_dir(),
        stdout=None if self.verbose else subprocess.DEVNULL,
        stderr=None if self.verbose else subprocess.DEVNULL,
        check=False,
    )
```

run(activity, subtask, action)

Source Code in griptape/tools/base_tool.py

```
def run(self, activity: Callable, subtask: ActionsSubtask, action: ToolAction) ->␣
↪BaseArtifact:
    try:
        output = self.before_run(activity, subtask, action)

        output = self.try_run(activity, subtask, action, output)

        output = self.after_run(activity, subtask, action, output)
    except Exception as e:
        logging.debug(traceback.format_exc())
```

(continues on next page)

(continued from previous page)

```
        output = ErrorArtifact(str(e), exception=e)

    return output
```

schema()

Source Code in griptape/tools/base_tool.py

```
def schema(self) -> dict:
    full_schema = Schema(Or(*self.activity_schemas()), description=f"{self.name}␣
↪action schema.")

    return full_schema.json_schema(f"{self.name} ToolAction Schema")
```

to_native_tool_name(activity)

Converts a Tool's name and an Activity into to a native tool name.

The native tool name is a combination of the Tool's name and the Activity's name. The Tool's name may only contain letters and numbers, and the Activity's name may only contain letters, numbers, and underscores.

Parameters

Name	Type	Description	Default
activity	*Callable*	Activity to convert 	*required*

Returns

Name	Type	Description
str	*str*	Native tool name.

Source Code in griptape/tools/base_tool.py

```
def to_native_tool_name(self, activity: Callable) -> str:
    """Converts a Tools name and an Activity into to a native tool name.

    The native tool name is a combination of the Tools name and the Activitys name.
    The Tools name may only contain letters and numbers, and the Activitys name may␣
↪only contain letters, numbers, and underscores.

    Args:
        activity: Activity to convert

    Returns:
        str: Native tool name.
    """
    tool_name = self.name
    if re.match(r"^[a-zA-Z0-9]+$", tool_name) is None:
```

(continues on next page)

(continued from previous page)

```
        raise ValueError("Tool name can only contain letters and numbers.")

    activity_name = self.activity_name(activity)
    if re.match(r"^[a-zA-Z0-9_]+$", activity_name) is None:
        raise ValueError("Activity name can only contain letters, numbers, and␣
↪underscores.")

    return f"{tool_name}_{activity_name}"
```

tool_dir()

Source Code in griptape/tools/base_tool.py

```
def tool_dir(self) -> str:
    class_file = inspect.getfile(self.__class__)

    return os.path.dirname(os.path.abspath(class_file))
```

try_run(activity, subtask, action, value)

Source Code in griptape/tools/base_tool.py

```
@observable(tags=["Tool.run()"])
def try_run(
    self,
    activity: Callable,
    subtask: ActionsSubtask,
    action: ToolAction,
    value: Optional[dict],
) -> BaseArtifact:
    activity_result = activity(deepcopy(value))

    if isinstance(activity_result, BaseArtifact):
        result = activity_result
    else:
        logging.warning("Activity result is not an artifact; converting result to␣
↪InfoArtifact")

        if activity_result is None:
            result = InfoArtifact("Tool returned an empty value")
        else:
            result = InfoArtifact(activity_result)

    return result
```

validate()

Source Code in griptape/tools/base_tool.py

```
def validate(self) -> bool:
    if not os.path.exists(self.requirements_path):
```

(continues on next page)

(continued from previous page)

```
        raise Exception(f"{self.REQUIREMENTS_FILE} not found")
    return True
```

validate*output_memory(*, output_memory)

Source Code in griptape/tools/base_tool.py

```
@output_memory.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_output_memory(self, _: Attribute, output_memory: dict[str,␣
↪Optional[list[TaskMemory]]]) -> None:
    if output_memory:
        for activity_name, memory_list in output_memory.items():
            if not self.find_activity(activity_name):
                raise ValueError(f"activity {activity_name} doesnt exist")
            if memory_list is None:
                raise ValueError(f"memory list for activity {activity_name} cant be␣
↪None")

            output_memory_names = [memory.name for memory in memory_list]

            if len(output_memory_names) > len(set(output_memory_names)):
                raise ValueError(f"memory names have to be unique in activity
↪{activity_name} output")
```

CalculatorTool

Bases: *BaseTool* “)

Source Code in griptape/tools/calculator/tool.py

```
class CalculatorTool(BaseTool):
    @activity(
        config={
            "description": "Can be used for computing simple numerical or algebraic␣
↪calculations in Python",
            "schema": Schema(
                {
                    Literal(
                        "expression",
                        description="Arithmetic expression parsable in pure Python.␣
↪Single line only. "
                        "Dont use variables. Dont use any imports or external␣
↪libraries",
                    ): str,
                },
            ),
        },
    )
    def calculate(self, params: dict) -> BaseArtifact:
        import numexpr  # pyright: ignore[reportMissingImports]
```

(continues on next page)

(continued from previous page)

```
        try:
            expression = params["values"]["expression"]

            return TextArtifact(numexpr.evaluate(expression))
        except Exception as e:
            return ErrorArtifact(f"error calculating: {e}")
```

calculate(params)

Source Code in griptape/tools/calculator/tool.py

```
@activity(
    config={
        "description": "Can be used for computing simple numerical or algebraic␣
↪calculations in Python",
        "schema": Schema(
            {
                Literal(
                    "expression",
                    description="Arithmetic expression parsable in pure Python.␣
↪Single line only. "
                    "Dont use variables. Dont use any imports or external libraries",
                ): str,
            },
        ),
    },
)
def calculate(self, params: dict) -> BaseArtifact:
    import numexpr  # pyright: ignore[reportMissingImports]

    try:
        expression = params["values"]["expression"]

        return TextArtifact(numexpr.evaluate(expression))
    except Exception as e:
        return ErrorArtifact(f"error calculating: {e}")
```

ComputerTool

Bases: *BaseTool* “)

Source Code in griptape/tools/computer/tool.py

```
@define
class ComputerTool(BaseTool):
    local_workdir: Optional[str] = field(default=None, kw_only=True)
    container_workdir: str = field(default="/griptape", kw_only=True)
    env_vars: dict = field(factory=dict, kw_only=True)
    dockerfile_path: str = field(
        default=Factory(lambda self: f"{os.path.join(self.tool_dir(), resources/
↪Dockerfile)}", takes_self=True),
        kw_only=True,
```

(continues on next page)

(continued from previous page)

```
    )
    requirements_txt_path: str = field(
        default=Factory(lambda self: f"{os.path.join(self.tool_dir(), resources/
↪requirements.txt)}", takes_self=True),
        kw_only=True,
    )
    docker_client: DockerClient = field(
        default=Factory(lambda self: self.default_docker_client(), takes_self=True),
        kw_only=True,
    )

    _tempdir: Optional[tempfile.TemporaryDirectory] = field(default=None, kw_
↪only=True)

    def __attrs_post_init__(self) -> None:
        super().__attrs_post_init__()

        if self.local_workdir:
            Path(self.local_workdir).mkdir(parents=True, exist_ok=True)
        else:
            self._tempdir = tempfile.TemporaryDirectory()
            self.local_workdir = self._tempdir.name

    @docker_client.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_docker_client(self, _: Attribute, docker_client: DockerClient) ->␣
↪None:
        if not docker_client:
            raise ValueError("Docker client cant be initialized: make sure the Docker␣
↪daemon is running")

    def install_dependencies(self, env: Optional[dict[str, str]] = None) -> None:
        super().install_dependencies(env)

        self.remove_existing_container(self.container_name(self))
        self.build_image(self)

    @activity(
        config={
            "description": "Can be used to execute Python code to solve any␣
↪programmatic tasks and access and analyze"
            " files in the file system. If you need to use code output use `print`␣
↪statements. "
            "You have access to the following external Python libraries: "
            "{{ _self.dependencies() }}",
            "schema": Schema(
                {
                    Literal("code", description="Python code to execute"): str,
                    Literal(
                        "filename",
                        description="name of the file to put the Python code in␣
↪before executing it",
                    ): str,
```

(continues on next page)

(continued from previous page)

```
                },
            ),
        },
    )
    def execute_code(self, params: dict) -> BaseArtifact:
        code = params["values"]["code"]
        filename = params["values"]["filename"]

        return self.execute_code_in_container(filename, code)

    @activity(
        config={
            "description": "Can be used to execute shell commands in Linux",
            "schema": Schema({Literal("command", description="shell command to execute
↪"): str}),
        },
    )
    def execute_command(self, params: dict) -> BaseArtifact:
        command = params["values"]["command"]

        return self.execute_command_in_container(command)

    def execute_command_in_container(self, command: str) -> BaseArtifact:
        from docker.models.containers import Container

        try:
            binds = {self.local_workdir: {"bind": self.container_workdir, "mode": "rw
↪"}} if self.local_workdir else None

            container = self.docker_client.containers.run(  # pyright:␣
↪ignore[reportCallIssue]
                self.image_name(self),
                environment=self.env_vars,
                command=command,
                name=self.container_name(self),
                volumes=binds,  # pyright: ignore[reportArgumentType] According to␣
↪the [docs](https://docker-py.readthedocs.io/en/stable/containers.html), the type of␣
↪`volumes` is dict[str, dict[str, str]].
                stdout=True,
                stderr=True,
                detach=True,
            )

            if isinstance(container, Container):
                container.wait()

                stderr = container.logs(stdout=False, stderr=True).decode().strip()
                stdout = container.logs(stdout=True, stderr=False).decode().strip()

                container.stop()
                container.remove()
```

(continues on next page)

(continued from previous page)

```
                if stderr:
                    return ErrorArtifact(stderr)
                return TextArtifact(stdout)
            return ErrorArtifact("error running container")
        except Exception as e:
            return ErrorArtifact(f"error executing command: {e}")

    def execute_code_in_container(self, filename: str, code: str) -> BaseArtifact:
        container_file_path = os.path.join(self.container_workdir, filename)

        if self.local_workdir:
            tempdir = None
            local_workdir = self.local_workdir
        else:
            tempdir = tempfile.TemporaryDirectory()
            local_workdir = tempdir.name

        local_file_path = os.path.join(local_workdir, filename)

        try:
            Path(local_file_path).write_text(code)

            return self.execute_command_in_container(f"python {container_file_path}")
        except Exception as e:
            return ErrorArtifact(f"error executing code: {e}")
        finally:
            if tempdir:
                tempdir.cleanup()

    def default_docker_client(self) -> Optional[DockerClient]:
        import docker

        try:
            return docker.from_env()
        except Exception as e:
            logging.exception(e)

            return None

    def image_name(self, tool: BaseTool) -> str:
        import stringcase  # pyright: ignore[reportMissingImports]

        return f"{stringcase.snakecase(tool.name)}_image"

    def container_name(self, tool: BaseTool) -> str:
        import stringcase  # pyright: ignore[reportMissingImports]

        return f"{stringcase.snakecase(tool.name)}_container"

    def remove_existing_container(self, name: str) -> None:
        from docker.errors import NotFound
        from docker.models.containers import Container
```

(continues on next page)

(continued from previous page)

```
        try:
            existing_container = self.docker_client.containers.get(name)
            if isinstance(existing_container, Container):
                existing_container.remove(force=True)

                logging.info("Removed existing container %s", name)
        except NotFound:
            pass

    def build_image(self, tool: BaseTool) -> None:
        with tempfile.TemporaryDirectory() as temp_dir:
            shutil.copy(self.dockerfile_path, temp_dir)
            shutil.copy(self.requirements_txt_path, temp_dir)

            image = self.docker_client.images.build(path=temp_dir, tag=self.image_
↪name(tool), rm=True, forcerm=True)

            if isinstance(image, tuple):
                logging.info("Built image: %s", image[0].short_id)

    def dependencies(self) -> list[str]:
        with open(self.requirements_txt_path) as file:
            return [line.strip() for line in file]

    def __del__(self) -> None:
        if self._tempdir:
            self._tempdir.cleanup()
```

- `_tempdir = field(default=None, kw_only=True)` class-attribute instance-attribute
- `container_workdir = field(default='/griptape', kw_only=True)` class-attribute instance-attribute
- `docker_client = field(default=Factory(lambda self: self.default_docker_client(), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `dockerfile_path = field(default=Factory(lambda self: f'{os.path.join(self.tool_dir(), 'resources/Dockerfile')}', takes_self=True), kw_only=True)` class-attribute instance-attribute
- `env_vars = field(factory=dict, kw_only=True)` class-attribute instance-attribute
- `local_workdir = field(default=None, kw_only=True)` class-attribute instance-attribute
- `requirements_txt_path = field(default=Factory(lambda self: f'{os.path.join(self.tool_dir(), 'resources/requirements.txt')}', takes_self=True), kw_only=True)` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/tools/computer/tool.py

```
def __attrs_post_init__(self) -> None:
    super().__attrs_post_init__()
```

(continues on next page)

(continued from previous page)

```
    if self.local_workdir:
        Path(self.local_workdir).mkdir(parents=True, exist_ok=True)
    else:
        self._tempdir = tempfile.TemporaryDirectory()
        self.local_workdir = self._tempdir.name
```

del()

Source Code in griptape/tools/computer/tool.py

```
def __del__(self) -> None:
    if self._tempdir:
        self._tempdir.cleanup()
```

build_image(tool)

Source Code in griptape/tools/computer/tool.py

```
def build_image(self, tool: BaseTool) -> None:
    with tempfile.TemporaryDirectory() as temp_dir:
        shutil.copy(self.dockerfile_path, temp_dir)
        shutil.copy(self.requirements_txt_path, temp_dir)

        image = self.docker_client.images.build(path=temp_dir, tag=self.image_
↪name(tool), rm=True, forcerm=True)

        if isinstance(image, tuple):
            logging.info("Built image: %s", image[0].short_id)
```

container_name(tool)

Source Code in griptape/tools/computer/tool.py

```
def container_name(self, tool: BaseTool) -> str:
    import stringcase  # pyright: ignore[reportMissingImports]

    return f"{stringcase.snakecase(tool.name)}_container"
```

default_docker_client()

Source Code in griptape/tools/computer/tool.py

```
def default_docker_client(self) -> Optional[DockerClient]:
    import docker

    try:
        return docker.from_env()
    except Exception as e:
        logging.exception(e)

        return None
```

dependencies()

Source Code in griptape/tools/computer/tool.py

```
def dependencies(self) -> list[str]:
    with open(self.requirements_txt_path) as file:
        return [line.strip() for line in file]
```

execute_code(params)

Source Code in griptape/tools/computer/tool.py

```
@activity(
    config={
        "description": "Can be used to execute Python code to solve any programmatic␣
↪tasks and access and analyze"
        " files in the file system. If you need to use code output use `print`␣
↪statements. "
        "You have access to the following external Python libraries: "
        "{{ _self.dependencies() }}",
        "schema": Schema(
            {
                Literal("code", description="Python code to execute"): str,
                Literal(
                    "filename",
                    description="name of the file to put the Python code in before␣
↪executing it",
                ): str,
            },
        ),
    },
)
def execute_code(self, params: dict) -> BaseArtifact:
    code = params["values"]["code"]
    filename = params["values"]["filename"]

    return self.execute_code_in_container(filename, code)
```

execute_code_in_container(filename, code)

Source Code in griptape/tools/computer/tool.py

```
def execute_code_in_container(self, filename: str, code: str) -> BaseArtifact:
    container_file_path = os.path.join(self.container_workdir, filename)

    if self.local_workdir:
        tempdir = None
        local_workdir = self.local_workdir
    else:
        tempdir = tempfile.TemporaryDirectory()
        local_workdir = tempdir.name

    local_file_path = os.path.join(local_workdir, filename)
```

(continues on next page)

(continued from previous page)

```
    try:
        Path(local_file_path).write_text(code)

        return self.execute_command_in_container(f"python {container_file_path}")
    except Exception as e:
        return ErrorArtifact(f"error executing code: {e}")
    finally:
        if tempdir:
            tempdir.cleanup()
```

execute_command(params)

Source Code in griptape/tools/computer/tool.py

```
@activity(
    config={
        "description": "Can be used to execute shell commands in Linux",
        "schema": Schema({Literal("command", description="shell command to execute"):␣
↪str}),
    },
)
def execute_command(self, params: dict) -> BaseArtifact:
    command = params["values"]["command"]

    return self.execute_command_in_container(command)
```

execute_command_in_container(command)

Source Code in griptape/tools/computer/tool.py

```
def execute_command_in_container(self, command: str) -> BaseArtifact:
    from docker.models.containers import Container

    try:
        binds = {self.local_workdir: {"bind": self.container_workdir, "mode": "rw"}}␣
↪if self.local_workdir else None

        container = self.docker_client.containers.run(  # pyright:␣
↪ignore[reportCallIssue]
            self.image_name(self),
            environment=self.env_vars,
            command=command,
            name=self.container_name(self),
            volumes=binds,  # pyright: ignore[reportArgumentType] According to the␣
↪[docs](https://docker-py.readthedocs.io/en/stable/containers.html), the type of␣
↪`volumes` is dict[str, dict[str, str]].
            stdout=True,
            stderr=True,
            detach=True,
        )

        if isinstance(container, Container):
```

(continues on next page)

(continued from previous page)

```
            container.wait()

            stderr = container.logs(stdout=False, stderr=True).decode().strip()
            stdout = container.logs(stdout=True, stderr=False).decode().strip()

            container.stop()
            container.remove()

            if stderr:
                return ErrorArtifact(stderr)
            return TextArtifact(stdout)
        return ErrorArtifact("error running container")
    except Exception as e:
        return ErrorArtifact(f"error executing command: {e}")
```

image_name(tool)

Source Code in griptape/tools/computer/tool.py

```
def image_name(self, tool: BaseTool) -> str:
    import stringcase  # pyright: ignore[reportMissingImports]

    return f"{stringcase.snakecase(tool.name)}_image"
```

install_dependencies(env=None)

Source Code in griptape/tools/computer/tool.py

```
def install_dependencies(self, env: Optional[dict[str, str]] = None) -> None:
    super().install_dependencies(env)

    self.remove_existing_container(self.container_name(self))
    self.build_image(self)
```

remove_existing_container(name)

Source Code in griptape/tools/computer/tool.py

```
def remove_existing_container(self, name: str) -> None:
    from docker.errors import NotFound
    from docker.models.containers import Container

    try:
        existing_container = self.docker_client.containers.get(name)
        if isinstance(existing_container, Container):
            existing_container.remove(force=True)

            logging.info("Removed existing container %s", name)
    except NotFound:
        pass
```

validate*docker_client*(, docker_client)

Source Code in griptape/tools/computer/tool.py

```
@docker_client.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_docker_client(self, _: Attribute, docker_client: DockerClient) -> None:
    if not docker_client:
        raise ValueError("Docker client cant be initialized: make sure the Docker␣
↪daemon is running")
```

DateTimeTool

Bases: *BaseTool* “)

Source Code in griptape/tools/date_time/tool.py

```
@define
class DateTimeTool(BaseTool):
    denylist: list[str] = field(default=Factory(lambda: ["get_relative_datetime"]),␣
↪kw_only=True)

    @activity(config={"description": "Can be used to return current date and time."})
    def get_current_datetime(self) -> BaseArtifact:
        return TextArtifact(str(datetime.now()))

    @activity(
        config={
            "description": "Can be used to return a relative date and time.",
            "schema": schema.Schema(
                {
                    schema.Literal(
                        "relative_date_string",
                        description=Relative date in English compatible with the␣
↪dateparser library. For example, "now EST", "20 minutes ago",
                        "in 2 days", "3 months, 1 week and 1 day ago", or "yesterday␣
↪at 2pm",
                    ): str,
                },
            ),
        },
    )
    def get_relative_datetime(self, params: dict) -> BaseArtifact:
        from dateparser import parse

        try:
            date_string = params["values"]["relative_date_string"]
            relative_datetime = parse(date_string)

            if relative_datetime:
                return TextArtifact(str(relative_datetime))
            return ErrorArtifact("invalid date string")
        except Exception as e:
            return ErrorArtifact(f"error getting current datetime: {e}")
```

(continues on next page)

(continued from previous page)

```
    @activity(
        config={
            "description": "Can be used to add a timedelta to a datetime.",
            "schema": schema.Schema(
                {
                    schema.Literal(
                        "timedelta_kwargs",
                        description=A dictionary of keyword arguments to pass to the␣
↪timedelta function. For example, {"days": -1, "hours": 2},
                    ): dict,
                    schema.Optional(
                        schema.Literal(
                            "iso_datetime",
                            description=Datetime represented as a string in ISO 8601␣
↪format. For example, "2021-01-01T00:00:00". Defaults to the current datetime if not␣
↪provided.,
                        )
                    ): str,
                },
            ),
        },
    )
    def add_timedelta(self, timedelta_kwargs: dict, iso_datetime: Optional[str] =␣
↪None) -> BaseArtifact:
        if iso_datetime is None:
            iso_datetime = datetime.now().isoformat()
        return TextArtifact((datetime.fromisoformat(iso_datetime) +␣
↪timedelta(**timedelta_kwargs)).isoformat())

    @activity(
        config={
            "description": "Can be used to calculate the difference between two␣
↪datetimes. The difference is calculated as end_datetime - start_datetime.",
            "schema": schema.Schema(
                {
                    schema.Literal(
                        "start_datetime",
                        description=Datetime represented as a string in ISO 8601␣
↪format. For example, "2021-01-01T00:00:00",
                    ): str,
                    schema.Literal(
                        "end_datetime",
                        description=Datetime represented as a string in ISO 8601␣
↪format. For example, "2021-01-02T00:00:00",
                    ): str,
                }
            ),
        }
    )
    def get_datetime_diff(self, start_datetime: str, end_datetime: str) ->␣
↪BaseArtifact:
        return TextArtifact(str(datetime.fromisoformat(end_datetime) - datetime.
```

(continues on next page)

(continued from previous page)

```
↪fromisoformat(start_datetime)))
```

- `denylist = field(default=Factory(lambda: ['get_relative_datetime']), kw_only=True)` class-attribute instance-attribute

add_timedelta(timedelta_kwargs, iso_datetime=None)

Source Code in griptape/tools/date_time/tool.py

```
@activity(
    config={
        "description": "Can be used to add a timedelta to a datetime.",
        "schema": schema.Schema(
            {
                schema.Literal(
                    "timedelta_kwargs",
                    description=A dictionary of keyword arguments to pass to the␣
↪timedelta function. For example, {"days": -1, "hours": 2},
                ): dict,
                schema.Optional(
                    schema.Literal(
                        "iso_datetime",
                        description=Datetime represented as a string in ISO 8601␣
↪format. For example, "2021-01-01T00:00:00". Defaults to the current datetime if not␣
↪provided.,
                    )
                ): str,
            },
        ),
    },
)
def add_timedelta(self, timedelta_kwargs: dict, iso_datetime: Optional[str] = None) ->
↪ BaseArtifact:
    if iso_datetime is None:
        iso_datetime = datetime.now().isoformat()
    return TextArtifact((datetime.fromisoformat(iso_datetime) + timedelta(**timedelta_
↪kwargs)).isoformat())
```

get_current_datetime()

Source Code in griptape/tools/date_time/tool.py

```
@activity(config={"description": "Can be used to return current date and time."})
def get_current_datetime(self) -> BaseArtifact:
    return TextArtifact(str(datetime.now()))
```

get_datetime_diff(start_datetime, end_datetime)

Source Code in griptape/tools/date_time/tool.py

```
@activity(
    config={
```

(continues on next page)

(continued from previous page)

```
        "description": "Can be used to calculate the difference between two datetimes.
↪ The difference is calculated as end_datetime - start_datetime.",
        "schema": schema.Schema(
            {
                schema.Literal(
                    "start_datetime",
                    description=Datetime represented as a string in ISO 8601 format.␣
↪For example, "2021-01-01T00:00:00",
                ): str,
                schema.Literal(
                    "end_datetime",
                    description=Datetime represented as a string in ISO 8601 format.␣
↪For example, "2021-01-02T00:00:00",
                ): str,
            }
        ),
    }
)
def get_datetime_diff(self, start_datetime: str, end_datetime: str) -> BaseArtifact:
    return TextArtifact(str(datetime.fromisoformat(end_datetime) - datetime.
↪fromisoformat(start_datetime)))
```

get_relative_datetime(params)

Source Code in griptape/tools/date_time/tool.py

```
@activity(
    config={
        "description": "Can be used to return a relative date and time.",
        "schema": schema.Schema(
            {
                schema.Literal(
                    "relative_date_string",
                    description=Relative date in English compatible with the␣
↪dateparser library. For example, "now EST", "20 minutes ago",
                    "in 2 days", "3 months, 1 week and 1 day ago", or "yesterday at␣
↪2pm",
                ): str,
            },
        ),
    },
)
def get_relative_datetime(self, params: dict) -> BaseArtifact:
    from dateparser import parse

    try:
        date_string = params["values"]["relative_date_string"]
        relative_datetime = parse(date_string)

        if relative_datetime:
            return TextArtifact(str(relative_datetime))
        return ErrorArtifact("invalid date string")
```

(continues on next page)

(continued from previous page)

```
    except Exception as e:
        return ErrorArtifact(f"error getting current datetime: {e}")
```

EmailTool

Bases: *BaseTool* “)

Attributes

Name	Type	Description
<a href="./#griptape.tools.EmailTool.username" title="username = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.EmailTool.username)"><code class="language-python">	*Optional[str]*	Username/email address used to send email via the SMTP protocol and retrieve email via the IMAP protocol. Example: bender@futurama.com.
<a href="./#griptape.tools.EmailTool.password" title="password = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.EmailTool.password)"><code class="language-python">	*Optional[str]*	Password used to send email via the SMTP protocol and retrieve email via the IMAP protocol. If using gmail, this would be an App Password.
<a href="./#griptape.tools.EmailTool.email_max_retrieve_count" title="email_max_retrieve_count = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.EmailTool.email_max_retrieve_count)"><code class="language-python">	*Optional[int]*	Used to limit the number of messages retrieved during any given activities.

continues on next page

Table 384 – continued from previous page

Name	Type	Description
<a href="./#griptape.tools.EmailTool.smtp_host" title="smtp_host = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.EmailTool.smtp_host)"><code class="language-python">	*Optional[str]*	Hostname or url of the SMTP server. Example: smtp.gmail.com. Required when using the *send* activity.
<a href="./#griptape.tools.EmailTool.smtp_port" title="smtp_port = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.EmailTool.smtp_port)"><code class="language-python">	*Optional[int]*	Port of the SMTP server. Example: 465. Required when using the *send* activity.
<a href="./#griptape.tools.EmailTool.smtp_use_ssl" title="smtp_use_ssl = field(default=True, kw_only=True) class-attribute instance-attribute (griptape.tools.EmailTool.smtp_use_ssl)"><co class="language-python">	*bool*	Whether to use SSL when sending email via the SMTP protocol.
<a href="./#griptape.tools.EmailTool.smtp_user" title="smtp_user = field(default=Factory(lambda self: self.username, takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.tools.EmailTool.smtp_user)"><code class="language-python">	*Optional[str]*	Username/email address used to send email via the SMTP protocol. Overrides username for SMTP only. Required when using the *send* activity.

continues on next page

Table 384 – continued from previous page

Name	Type	Description
<a href="./#griptape.tools.EmailTool.smtp_password" title="smtp_password = field(default=Factory(lambda self: self.password, takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.tools.EmailTool.smtp_password)"><code class="language-python">	*Optional[str]*	Password to send email via the SMTP protocol. Overrides password for SMTP only. Required when using the *send* activity.
<a href="./#griptape.tools.EmailTool.imap_url" title="imap_url = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.EmailTool.imap_url)"><code class="language-python">	*Optional[str]*	Hostname or url of the IMAP server. Example: imap.gmail.com. Required when using the *retrieve* activity.
<code class="language-python">	*Optional[str]*	Username/email address used to retrieve email via the IMAP protocol. Overrides username for IMAP only. Required when using the *retrieve* activity.

continues on next page

Table 384 – continued from previous page

Name	Type	Description
<code class="language-python">	*Optional[str]*	Password to retrieve email via the IMAP protocol. Overrides password for IMAP only. Required when using the *retrieve* activity.
<code class="language-python">	*Optional[dict[str, Optional[str]]]*	Descriptions of mailboxes available for retrieving email via the IMAP protocol. Required when using the *retrieve* activity. Example: {'INBOX': 'default mailbox for incoming email', 'SENT': 'default mailbox for sent email'}
<co class="language-python">	<code class="language-python">EmailLoader</code>	Instance of *EmailLoader*.

Source Code in griptape/tools/email/tool.py

```
@define
class EmailTool(BaseTool):
    """Tool for working with email.

    Attributes:
        username: Username/email address used to send email via the SMTP protocol and␣
↪retrieve email via the IMAP protocol.
            Example: bender@futurama.com.
        password: Password used to send email via the SMTP protocol and retrieve␣
↪email via the IMAP protocol. If using gmail,
            this would be an App Password.
        email_max_retrieve_count: Used to limit the number of messages retrieved␣
↪during any given activities.
        smtp_host: Hostname or url of the SMTP server. Example: smtp.gmail.com.␣
↪Required when using the `send` activity.
        smtp_port: Port of the SMTP server. Example: 465. Required when using the␣
↪`send` activity.
        smtp_use_ssl: Whether to use SSL when sending email via the SMTP protocol.
        smtp_user: Username/email address used to send email via the SMTP protocol.␣
↪Overrides username for SMTP only. Required when using the `send` activity.
        smtp_password: Password to send email via the SMTP protocol. Overrides␣
↪password for SMTP only. Required when using the `send` activity.
        imap_url: Hostname or url of the IMAP server. Example: imap.gmail.com.␣
↪Required when using the `retrieve` activity.
        imap_user: Username/email address used to retrieve email via the IMAP␣
↪protocol. Overrides username for IMAP only. Required when using the `retrieve`␣
↪activity.
        imap_password: Password to retrieve email via the IMAP protocol.  Overrides␣
↪password for IMAP only. Required when using the `retrieve` activity.
        mailboxes: Descriptions of mailboxes available for retrieving email via the␣
↪IMAP protocol. Required when using the `retrieve` activity.
            Example: {INBOX: default mailbox for incoming email, SENT: default␣
↪mailbox for sent email}
        email_loader: Instance of `EmailLoader`.
    """

    username: Optional[str] = field(default=None, kw_only=True)
    password: Optional[str] = field(default=None, kw_only=True)
    email_max_retrieve_count: Optional[int] = field(default=None, kw_only=True)
    smtp_host: Optional[str] = field(default=None, kw_only=True)
    smtp_port: Optional[int] = field(default=None, kw_only=True)
    smtp_use_ssl: bool = field(default=True, kw_only=True)
    smtp_user: Optional[str] = field(default=Factory(lambda self: self.username,␣
↪takes_self=True), kw_only=True)
    smtp_password: Optional[str] = field(default=Factory(lambda self: self.password,␣
↪takes_self=True), kw_only=True)
    imap_url: Optional[str] = field(default=None, kw_only=True)
    imap_user: Optional[str] = field(default=Factory(lambda self: self.username,␣
↪takes_self=True), kw_only=True)
    imap_password: Optional[str] = field(default=Factory(lambda self: self.password,␣
↪takes_self=True), kw_only=True)
    mailboxes: Optional[dict[str, Optional[str]]] = field(default=None, kw_only=True)
    email_loader: EmailLoader = field(
```

(continues on next page)

(continued from previous page)

```
        default=Factory(
            lambda self: EmailLoader(imap_url=self.imap_url, username=self.imap_user,␣
↪password=self.imap_password),
            takes_self=True,
        ),
        kw_only=True,
    )

    @activity(
        config={
            "description": "Can be used to retrieve emails."
            "{% if _self.mailboxes %} Available mailboxes: {{ _self.mailboxes }}{%␣
↪endif %}",
            "schema": Schema(
                {
                    Literal("label", description="Label to retrieve emails from such␣
↪as INBOX or SENT"): str,
                    schema.Optional(
                        Literal("key", description="Optional key for filtering such␣
↪as FROM or SUBJECT"),
                    ): str,
                    schema.Optional(
                        Literal("search_criteria", description="Optional search␣
↪criteria to filter emails by key"),
                    ): str,
                    schema.Optional(Literal("max_count", description="Optional max␣
↪email count")): int,
                },
            ),
        },
    )
    def retrieve(self, params: dict) -> ListArtifact | ErrorArtifact:
        if self.mailboxes is None:
            return ErrorArtifact("mailboxes is required")

        values = params["values"]
        max_count = int(values["max_count"]) if values.get("max_count") is not None␣
↪else self.email_max_retrieve_count

        return self.email_loader.load(
            EmailLoader.EmailQuery(
                label=values["label"],
                key=values.get("key"),
                search_criteria=values.get("search_criteria"),
                max_count=max_count,
            ),
        )

    @activity(
        config={
            "description": "Can be used to send emails",
            "schema": Schema(
```

(continues on next page)

(continued from previous page)

```
                {
                    Literal("to", description="Recipients email address"): str,
                    Literal("subject", description="Email subject"): str,
                    Literal("body", description="Email body"): str,
                },
            ),
        },
    )
    def send(self, params: dict) -> InfoArtifact | ErrorArtifact:
        values = params["values"]
        if self.smtp_user is None:
            return ErrorArtifact("smtp_user is required")
        if self.smtp_password is None:
            return ErrorArtifact("smtp_password is required")
        if self.smtp_host is None:
            return ErrorArtifact("smtp_host is required")
        if self.smtp_port is None:
            return ErrorArtifact("smtp_port is required")

        msg = MIMEText(values["body"])
        msg["Subject"] = values["subject"]
        msg["From"] = self.smtp_user
        msg["To"] = values["to"]

        try:
            with self._create_smtp_client(self.smtp_host, self.smtp_port) as client:
                client.login(self.smtp_user, self.smtp_password)
                client.sendmail(msg["From"], [msg["To"]], msg.as_string())
                return InfoArtifact("email was successfully sent")
        except Exception as e:
            logging.exception(e)
            return ErrorArtifact(f"error sending email: {e}")

    def _create_smtp_client(self, smtp_host: str, smtp_port: int) -> smtplib.SMTP |␣
↪smtplib.SMTP_SSL:
        if self.smtp_use_ssl:
            return smtplib.SMTP_SSL(smtp_host, smtp_port)
        return smtplib.SMTP(smtp_host, smtp_port)
```

- `email_loader = field(default=Factory(lambda self: EmailLoader(imap_url=self.imap_url, username=self.imap_user, password=self.imap_password), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `email_max_retrieve_count = field(default=None, kw_only=True)` class-attribute instance-attribute
- `imap_password = field(default=Factory(lambda self: self.password, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `imap_url = field(default=None, kw_only=True)` class-attribute instance-attribute
- `imap_user = field(default=Factory(lambda self: self.username, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `mailboxes = field(default=None, kw_only=True)` class-attribute instance-attribute

- `password = field(default=None, kw_only=True)` class-attribute instance-attribute
- `smtp_host = field(default=None, kw_only=True)` class-attribute instance-attribute
- `smtp_password = field(default=Factory(lambda self: self.password, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `smtp_port = field(default=None, kw_only=True)` class-attribute instance-attribute
- `smtp_use_ssl = field(default=True, kw_only=True)` class-attribute instance-attribute
- `smtp_user = field(default=Factory(lambda self: self.username, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `username = field(default=None, kw_only=True)` class-attribute instance-attribute

_create_smtp_client(smtp_host, smtp_port)

Source Code in griptape/tools/email/tool.py

```
def _create_smtp_client(self, smtp_host: str, smtp_port: int) -> smtplib.SMTP |␣
↪smtplib.SMTP_SSL:
    if self.smtp_use_ssl:
        return smtplib.SMTP_SSL(smtp_host, smtp_port)
    return smtplib.SMTP(smtp_host, smtp_port)
```

retrieve(params)

Source Code in griptape/tools/email/tool.py

```
@activity(
    config={
        "description": "Can be used to retrieve emails."
        "{% if _self.mailboxes %} Available mailboxes: {{ _self.mailboxes }}{% endif
↪%}",
        "schema": Schema(
            {
                Literal("label", description="Label to retrieve emails from such as␣
↪INBOX or SENT"): str,
                schema.Optional(
                    Literal("key", description="Optional key for filtering such as␣
↪FROM or SUBJECT"),
                ): str,
                schema.Optional(
                    Literal("search_criteria", description="Optional search criteria␣
↪to filter emails by key"),
                ): str,
                schema.Optional(Literal("max_count", description="Optional max email␣
↪count")): int,
            },
        ),
    },
)
def retrieve(self, params: dict) -> ListArtifact | ErrorArtifact:
    if self.mailboxes is None:
        return ErrorArtifact("mailboxes is required")
```

(continues on next page)

(continued from previous page)

```
    values = params["values"]
    max_count = int(values["max_count"]) if values.get("max_count") is not None else␣
↪self.email_max_retrieve_count

    return self.email_loader.load(
        EmailLoader.EmailQuery(
            label=values["label"],
            key=values.get("key"),
            search_criteria=values.get("search_criteria"),
            max_count=max_count,
        ),
    )
```

send(params)

Source Code in griptape/tools/email/tool.py

```
@activity(
    config={
        "description": "Can be used to send emails",
        "schema": Schema(
            {
                Literal("to", description="Recipients email address"): str,
                Literal("subject", description="Email subject"): str,
                Literal("body", description="Email body"): str,
            },
        ),
    },
)
def send(self, params: dict) -> InfoArtifact | ErrorArtifact:
    values = params["values"]
    if self.smtp_user is None:
        return ErrorArtifact("smtp_user is required")
    if self.smtp_password is None:
        return ErrorArtifact("smtp_password is required")
    if self.smtp_host is None:
        return ErrorArtifact("smtp_host is required")
    if self.smtp_port is None:
        return ErrorArtifact("smtp_port is required")

    msg = MIMEText(values["body"])
    msg["Subject"] = values["subject"]
    msg["From"] = self.smtp_user
    msg["To"] = values["to"]

    try:
        with self._create_smtp_client(self.smtp_host, self.smtp_port) as client:
            client.login(self.smtp_user, self.smtp_password)
            client.sendmail(msg["From"], [msg["To"]], msg.as_string())
            return InfoArtifact("email was successfully sent")
    except Exception as e:
        logging.exception(e)
```

(continues on next page)

(continued from previous page)

```
        return ErrorArtifact(f"error sending email: {e}")
```

ExtractionTool

Bases: *BaseTool* “) , *RuleMixin* ”)

Attributes

Name	Type	Description
<a href="./#griptape.tools.ExtractionTool.extraction_engine" title="extraction_engine = field() class-attribute instance-attribute (griptape.tools.ExtractionTool.extraction_engine)"><code class="language-python">	<a href="../engines#griptape.engines.BaseExtractionEngine" title="BaseExtractionEngine (griptape.engines.BaseExtractionEngine)"><code class="language-python">BaseExtractionEngine</cod	*ExtractionEngine*.

Source Code in griptape/tools/extraction/tool.py

```
@define(kw_only=True)
class ExtractionTool(BaseTool, RuleMixin):
    """Tool for using an Extraction Engine.

    Attributes:
        extraction_engine: `ExtractionEngine`.
    """

    extraction_engine: BaseExtractionEngine = field()

    @activity(
        config={
            "description": "Can be used extract structured text from data.",
            "schema": Schema(
                {
                    Literal("data"): Or(
                        str,
                        Schema(
                            {
                                "memory_name": str,
                                "artifact_namespace": str,
                            }
                        ),
                    ),
                }
            ),
        },
```

(continues on next page)

(continued from previous page)

```
    )
    def extract(self, params: dict) -> ListArtifact | InfoArtifact | ErrorArtifact:
        data = params["values"]["data"]

        if isinstance(data, str):
            artifacts = ListArtifact([TextArtifact(data)])
        else:
            memory = self.find_input_memory(data["memory_name"])
            artifact_namespace = data["artifact_namespace"]

            if memory is not None:
                artifacts = memory.load_artifacts(artifact_namespace)
            else:
                return ErrorArtifact("memory not found")

        return self.extraction_engine.extract_artifacts(artifacts)
```

- `extraction_engine = field()` class-attribute instance-attribute

extract(params)

Source Code in griptape/tools/extraction/tool.py

```
@activity(
    config={
        "description": "Can be used extract structured text from data.",
        "schema": Schema(
            {
                Literal("data"): Or(
                    str,
                    Schema(
                        {
                            "memory_name": str,
                            "artifact_namespace": str,
                        }
                    ),
                ),
            }
        ),
    },
)
def extract(self, params: dict) -> ListArtifact | InfoArtifact | ErrorArtifact:
    data = params["values"]["data"]

    if isinstance(data, str):
        artifacts = ListArtifact([TextArtifact(data)])
    else:
        memory = self.find_input_memory(data["memory_name"])
        artifact_namespace = data["artifact_namespace"]

        if memory is not None:
            artifacts = memory.load_artifacts(artifact_namespace)
```

(continues on next page)

(continued from previous page)

```
        else:
            return ErrorArtifact("memory not found")

    return self.extraction_engine.extract_artifacts(artifacts)
```

FileManagerTool

Bases: *BaseTool* “)

Attributes

Name	Type	Description
<a href="./#griptape.tools.FileManagerTool.file_manager_driver" title="file_manager_driver = field(default=Factory(lambda: LocalFileManagerDriver()), kw_only=True) class-attribute instance-attribute (griptape.tools.FileManagerTool.file_manager_driver)"><code class="language-python">	<a href="../drivers/file_manager#griptape.drivers.file_manager.BaseFileManagerDriver" title="BaseFileManagerDriver (griptape.drivers.file_manager.BaseFileManagerDriver)"><code class="language-python">BaseFileManagerDriver</co	File Manager Driver to use to list, load, and save files.

Source Code in griptape/tools/file_manager/tool.py

```
@define
class FileManagerTool(BaseTool):
    """FileManagerTool is a tool that can be used to list, load, and save files.

    Attributes:
        file_manager_driver: File Manager Driver to use to list, load, and save files.
    """

    file_manager_driver: BaseFileManagerDriver = field(default=Factory(lambda:␣
↪LocalFileManagerDriver()), kw_only=True)

    loaders: dict[str, loaders.BaseLoader] = field(
        default=Factory(
            lambda self: {
                "application/pdf": loaders.PdfLoader(file_manager_driver=self.file_
↪manager_driver),
                "text/csv": loaders.CsvLoader(file_manager_driver=self.file_manager_
↪driver),
                "text": loaders.TextLoader(file_manager_driver=self.file_manager_
```

(continues on next page)

(continued from previous page)

```
↪driver),
                "image": loaders.ImageLoader(file_manager_driver=self.file_manager_
↪driver),
                "application/octet-stream": BlobLoader(file_manager_driver=self.file_
↪manager_driver),
            },
            takes_self=True,
        ),
        kw_only=True,
    )

    @activity(
        config={
            "description": "Can be used to list files on disk",
            "schema": Schema(
                {Literal("path", description="Relative path in the POSIX format. For␣
↪example, foo/bar"): str},
            ),
        },
    )
    def list_files_from_disk(self, params: dict) -> TextArtifact | ErrorArtifact:
        path = params["values"]["path"]
        return self.file_manager_driver.list_files(path)

    @activity(
        config={
            "description": "Can be used to load files from disk",
            "schema": Schema(
                {
                    Literal(
                        "paths",
                        description="Relative paths to files to be loaded in the␣
↪POSIX format. For example, [foo/bar/file.txt]",
                    ): Schema([str]),
                },
            ),
        },
    )
    def load_files_from_disk(self, params: dict) -> ListArtifact | ErrorArtifact:
        paths = params["values"]["paths"]
        artifacts = []

        for path in paths:
            # Fetch the file to try and determine the appropriate loader
            abs_path = os.path.join(self.file_manager_driver.workdir, path)
            mime_type = get_mime_type(abs_path)
            loader = next((loader for key, loader in self.loaders.items() if mime_
↪type.startswith(key)))

            artifact = loader.load(path)
            if isinstance(artifact, ListArtifact):
                artifacts.extend(artifact.value)
```

(continues on next page)

(continued from previous page)

```
            else:
                artifacts.append(artifact)
        return ListArtifact(artifacts)

    @activity(
        config={
            "description": "Can be used to save memory artifacts to disk",
            "schema": Schema(
                {
                    Literal(
                        "dir_name",
                        description="Relative destination path name on disk in the␣
↪POSIX format. For example, foo/bar",
                    ): str,
                    Literal("file_name", description="Destination file name. For␣
↪example, baz.txt"): str,
                    "memory_name": str,
                    "artifact_namespace": str,
                },
            ),
        },
    )
    def save_memory_artifacts_to_disk(self, params: dict) -> ErrorArtifact |␣
↪InfoArtifact:
        dir_name = params["values"]["dir_name"]
        file_name = params["values"]["file_name"]
        memory_name = params["values"]["memory_name"]
        artifact_namespace = params["values"]["artifact_namespace"]

        memory = self.find_input_memory(params["values"]["memory_name"])
        if not memory:
            return ErrorArtifact(f"Failed to save memory artifacts to disk - memory␣
↪named {memory_name} not found")

        list_artifact = memory.load_artifacts(artifact_namespace)

        if len(list_artifact) == 0:
            return ErrorArtifact(
                f"Failed to save memory artifacts to disk - memory named {memory_name}
↪ does not contain any artifacts",
            )

        for artifact in list_artifact.value:
            formatted_file_name = f"{artifact.name}-{file_name}" if len(list_
↪artifact) > 1 else file_name
            try:
                value = artifact.value if isinstance(artifact.value, (str, bytes))␣
↪else artifact.to_text()
                self.file_manager_driver.save_file(os.path.join(dir_name, formatted_
↪file_name), value)
            except FileNotFoundError:
                return ErrorArtifact("Path not found")
```

(continues on next page)

(continued from previous page)

```
            except IsADirectoryError:
                return ErrorArtifact("Path is a directory")
            except NotADirectoryError:
                return ErrorArtifact("Not a directory")
            except Exception as e:
                return ErrorArtifact(f"Failed to load file: {e!s}")

        return InfoArtifact("Successfully saved memory artifacts to disk")

    @activity(
        config={
            "description": "Can be used to save content to a file",
            "schema": Schema(
                {
                    Literal(
                        "path",
                        description="Destination file path on disk in the POSIX␣
↪format. For example, foo/bar/baz.txt",
                    ): str,
                    "content": str,
                },
            ),
        },
    )
    def save_content_to_file(self, params: dict) -> ErrorArtifact | InfoArtifact:
        path = params["values"]["path"]
        content = params["values"]["content"]
        return self.file_manager_driver.save_file(path, content)
```

- `file_manager_driver = field(default=Factory(lambda: LocalFileManagerDriver()), kw_only=True)` class-attribute instance-attribute
- `loaders = field(default=Factory(lambda self: {'application/pdf': loaders.PdfLoader(file_manager_driver=self.file_manager_driver), 'text/csv': loaders.CsvLoader(file_manager_driver=self.file_manager_driver), 'text': loaders.TextLoader(file_manager_driver=self.file_manager_driver), 'image': loaders.ImageLoader(file_manager_driver=self.file_manager_driver), 'application/octet-stream': BlobLoader(file_manager_driver=self.file_manager_driver)}, takes_self=True), kw_only=True)` class-attribute instance-attribute

list_files_from_disk(params)

Source Code in griptape/tools/file_manager/tool.py

```
@activity(
    config={
        "description": "Can be used to list files on disk",
        "schema": Schema(
            {Literal("path", description="Relative path in the POSIX format. For␣
↪example, foo/bar"): str},
        ),
    },
)
```

(continues on next page)

(continued from previous page)

```
def list_files_from_disk(self, params: dict) -> TextArtifact | ErrorArtifact:
    path = params["values"]["path"]
    return self.file_manager_driver.list_files(path)
```

load_files_from_disk(params)

Source Code in griptape/tools/file_manager/tool.py

```
@activity(
    config={
        "description": "Can be used to load files from disk",
        "schema": Schema(
            {
                Literal(
                    "paths",
                    description="Relative paths to files to be loaded in the POSIX␣
↪format. For example, [foo/bar/file.txt]",
                ): Schema([str]),
            },
        ),
    },
)
def load_files_from_disk(self, params: dict) -> ListArtifact | ErrorArtifact:
    paths = params["values"]["paths"]
    artifacts = []

    for path in paths:
        # Fetch the file to try and determine the appropriate loader
        abs_path = os.path.join(self.file_manager_driver.workdir, path)
        mime_type = get_mime_type(abs_path)
        loader = next((loader for key, loader in self.loaders.items() if mime_type.
↪startswith(key)))

        artifact = loader.load(path)
        if isinstance(artifact, ListArtifact):
            artifacts.extend(artifact.value)
        else:
            artifacts.append(artifact)
    return ListArtifact(artifacts)
```

save_content_to_file(params)

Source Code in griptape/tools/file_manager/tool.py

```
@activity(
    config={
        "description": "Can be used to save content to a file",
        "schema": Schema(
            {
                Literal(
                    "path",
                    description="Destination file path on disk in the POSIX format.␣
```

(continues on next page)

(continued from previous page)

```
↪For example, foo/bar/baz.txt",
                ): str,
                "content": str,
            },
        ),
    },
)
def save_content_to_file(self, params: dict) -> ErrorArtifact | InfoArtifact:
    path = params["values"]["path"]
    content = params["values"]["content"]
    return self.file_manager_driver.save_file(path, content)
```

save_memory_artifacts_to_disk(params)

Source Code in griptape/tools/file_manager/tool.py

```
@activity(
    config={
        "description": "Can be used to save memory artifacts to disk",
        "schema": Schema(
            {
                Literal(
                    "dir_name",
                    description="Relative destination path name on disk in the POSIX␣
↪format. For example, foo/bar",
                ): str,
                Literal("file_name", description="Destination file name. For example,␣
↪baz.txt"): str,
                "memory_name": str,
                "artifact_namespace": str,
            },
        ),
    },
)
def save_memory_artifacts_to_disk(self, params: dict) -> ErrorArtifact | InfoArtifact:
    dir_name = params["values"]["dir_name"]
    file_name = params["values"]["file_name"]
    memory_name = params["values"]["memory_name"]
    artifact_namespace = params["values"]["artifact_namespace"]

    memory = self.find_input_memory(params["values"]["memory_name"])
    if not memory:
        return ErrorArtifact(f"Failed to save memory artifacts to disk - memory named
↪{memory_name} not found")

    list_artifact = memory.load_artifacts(artifact_namespace)

    if len(list_artifact) == 0:
        return ErrorArtifact(
            f"Failed to save memory artifacts to disk - memory named {memory_name}␣
↪does not contain any artifacts",
        )
```

(continues on next page)

(continued from previous page)

```
    for artifact in list_artifact.value:
        formatted_file_name = f"{artifact.name}-{file_name}" if len(list_artifact) >␣
↪1 else file_name
        try:
            value = artifact.value if isinstance(artifact.value, (str, bytes)) else␣
↪artifact.to_text()
            self.file_manager_driver.save_file(os.path.join(dir_name, formatted_file_
↪name), value)
        except FileNotFoundError:
            return ErrorArtifact("Path not found")
        except IsADirectoryError:
            return ErrorArtifact("Path is a directory")
        except NotADirectoryError:
            return ErrorArtifact("Not a directory")
        except Exception as e:
            return ErrorArtifact(f"Failed to load file: {e!s}")

    return InfoArtifact("Successfully saved memory artifacts to disk")
```

GriptapeCloudToolTool

Bases: *BaseGriptapeCloudTool* “)

Attributes

Name	Type	Description
<a href="./#griptape.tools.GriptapeCloudToolTool.tool_id" title="tool_id = field(kw_only=True) class-attribute instance-attribute (griptape.tools.GriptapeCloudToolTool.tool_id)"><code class="language-python">	*str*	The ID of the tool to run.

Source Code in griptape/tools/griptape_cloud_tool/tool.py

```
@define()
class GriptapeCloudToolTool(BaseGriptapeCloudTool):
    """Runs a Gen AI Builder hosted Tool.

    Attributes:
        tool_id: The ID of the tool to run.
    """
```

(continues on next page)

(continued from previous page)

```
    tool_id: str = field(kw_only=True)

    def __attrs_post_init__(self) -> None:
        self._init_activities()

    def _init_activities(self) -> None:
        schema = self._get_schema()
        tool_name, activity_schemas = self._parse_schema(schema)

        if self.name == self.__class__.__name__:
            self.name = tool_name

        for activity_name, (description, activity_schema) in activity_schemas.items():
            activity_handler = self._create_activity_handler(activity_name,␣
↪description, activity_schema)

            setattr(self, activity_name, MethodType(activity_handler, self))

    def _get_schema(self) -> dict:
        response = requests.get(urljoin(self.base_url, f"/api/tools/{self.tool_id}/
↪openapi"), headers=self.headers)

        response.raise_for_status()
        schema = response.json()

        if not isinstance(schema, dict):
            raise RuntimeError(f"Invalid schema for tool {self.tool_id}: {schema}")

        if "error" in schema and "tool_run_id" in schema:
            raise RuntimeError(f"Failed to retrieve schema for tool {self.tool_id}:
↪{schema[error]}")

        return schema

    def _parse_schema(self, schema: dict) -> tuple[str, dict[str, tuple[str,␣
↪Schema]]]:
        """Parses an openapi schema into a dictionary of activity names and their␣
↪respective descriptions + schemas."""
        activities = {}

        name = schema.get("info", {}).get("title")

        for path, path_info in schema.get("paths", {}).items():
            if not path.startswith("/activities"):
                continue
            for method, method_info in path_info.items():
                if "post" in method.lower():
                    activity_name = method_info["operationId"]
                    description = method_info.get("description", "")

                    activity_schema = self.__extract_schema_from_ref(
                        schema,
```

(continues on next page)

(continued from previous page)

```
                        method_info.get("requestBody", {})
                        .get("content", {})
                        .get("application/json", {})
                        .get("schema", {}),
                    )

                    activities[activity_name] = (description, activity_schema)

        return name, activities

    def __extract_schema_from_ref(self, schema: dict, schema_ref: dict) -> Schema:
        """Extracts a schema from a $ref if present, resolving it into native schema␣
↪properties."""
        if "$ref" in schema_ref:
            # Resolve the reference and retrieve the schema data
            ref_path = schema_ref["$ref"].split("/")[-1]
            schema_data = schema["components"]["schemas"].get(ref_path, {})
        else:
            # Use the provided schema directly if no $ref is found
            schema_data = schema_ref

        # Convert the schema_data dictionary into a Schema with its properties
        properties = {}
        for prop, prop_info in schema_data.get("properties", {}).items():
            prop_type = prop_info.get("type", "string")
            prop_description = prop_info.get("description", "")
            schema_prop = Literal(prop, description=prop_description)
            is_optional = prop not in schema_data.get("required", [])

            if is_optional:
                schema_prop = SchemaOptional(schema_prop)

            properties[schema_prop] = self._map_openapi_type_to_python(prop_type,␣
↪prop_info)

        return Schema(properties)

    def _map_openapi_type_to_python(
        self, openapi_type: str, schema_info: Optional[dict] = None
    ) -> type | list[type] | Or:
        """Maps OpenAPI types to native Python types."""
        type_mapping = {"string": str, "integer": int, "boolean": bool, "number":␣
↪float, "object": dict}

        if openapi_type == "array" and schema_info is not None and "items" in schema_
↪info:
            enum = schema_info["items"].get("enum")
            if enum:
                return enum
            items_type = schema_info["items"].get("type", "string")
            return [self._map_openapi_type_to_python(items_type)]  # pyright:␣
↪ignore[reportReturnType]
```

(continues on next page)

(continued from previous page)

```
        if schema_info is not None and schema_info.get("enum"):
            return Or(*schema_info["enum"])
        return type_mapping.get(openapi_type, str)

    def _create_activity_handler(self, activity_name: str, description: str, activity_
↪schema: Schema) -> Callable:
        """Creates an activity handler method for the tool."""

        @activity(config={"name": activity_name, "description": description, "schema
↪": activity_schema})
        def activity_handler(self: GriptapeCloudToolTool, values: dict) -> Any:
            return self._run_activity(activity_name, values)

        return activity_handler

    def _run_activity(self, activity_name: str, params: dict) -> BaseArtifact:
        """Runs an activity on the tool with the provided parameters."""
        url = urljoin(self.base_url, f"/api/tools/{self.tool_id}/activities/{activity_
↪name}")

        response = requests.post(url, json=params, headers=self.headers)

        response.raise_for_status()

        try:
            return BaseArtifact.from_dict(response.json())
        except ValueError:
            return TextArtifact(response.text)
```

- `tool_id = field(kw_only=True)` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/tools/griptape_cloud_tool/tool.py

```
def __attrs_post_init__(self) -> None:
    self._init_activities()
```

__extract_schema_from_ref(schema, schema_ref)

Source Code in griptape/tools/griptape_cloud_tool/tool.py

```
def __extract_schema_from_ref(self, schema: dict, schema_ref: dict) -> Schema:
    """Extracts a schema from a $ref if present, resolving it into native schema␣
↪properties."""
    if "$ref" in schema_ref:
        # Resolve the reference and retrieve the schema data
        ref_path = schema_ref["$ref"].split("/")[-1]
        schema_data = schema["components"]["schemas"].get(ref_path, {})
    else:
        # Use the provided schema directly if no $ref is found
        schema_data = schema_ref
```

(continues on next page)

(continued from previous page)

```
    # Convert the schema_data dictionary into a Schema with its properties
    properties = {}
    for prop, prop_info in schema_data.get("properties", {}).items():
        prop_type = prop_info.get("type", "string")
        prop_description = prop_info.get("description", "")
        schema_prop = Literal(prop, description=prop_description)
        is_optional = prop not in schema_data.get("required", [])

        if is_optional:
            schema_prop = SchemaOptional(schema_prop)

        properties[schema_prop] = self._map_openapi_type_to_python(prop_type, prop_
↪info)

    return Schema(properties)
```

_create_activity_handler(activity_name, description, activity_schema)

Source Code in griptape/tools/griptape_cloud_tool/tool.py

```
def _create_activity_handler(self, activity_name: str, description: str, activity_
↪schema: Schema) -> Callable:
    """Creates an activity handler method for the tool."""

    @activity(config={"name": activity_name, "description": description, "schema":␣
↪activity_schema})
    def activity_handler(self: GriptapeCloudToolTool, values: dict) -> Any:
        return self._run_activity(activity_name, values)

    return activity_handler
```

_get_schema()

Source Code in griptape/tools/griptape_cloud_tool/tool.py

```
def _get_schema(self) -> dict:
    response = requests.get(urljoin(self.base_url, f"/api/tools/{self.tool_id}/openapi
↪"), headers=self.headers)

    response.raise_for_status()
    schema = response.json()

    if not isinstance(schema, dict):
        raise RuntimeError(f"Invalid schema for tool {self.tool_id}: {schema}")

    if "error" in schema and "tool_run_id" in schema:
        raise RuntimeError(f"Failed to retrieve schema for tool {self.tool_id}:
↪{schema[error]}")

    return schema
```

_init_activities()

Source Code in griptape/tools/griptape_cloud_tool/tool.py

```
def _init_activities(self) -> None:
    schema = self._get_schema()
    tool_name, activity_schemas = self._parse_schema(schema)

    if self.name == self.__class__.__name__:
        self.name = tool_name

    for activity_name, (description, activity_schema) in activity_schemas.items():
        activity_handler = self._create_activity_handler(activity_name, description,␣
↪activity_schema)

        setattr(self, activity_name, MethodType(activity_handler, self))
```

_map_openapi_type_to_python(openapi_type, schema_info=None)

Source Code in griptape/tools/griptape_cloud_tool/tool.py

```
def _map_openapi_type_to_python(
    self, openapi_type: str, schema_info: Optional[dict] = None
) -> type | list[type] | Or:
    """Maps OpenAPI types to native Python types."""
    type_mapping = {"string": str, "integer": int, "boolean": bool, "number": float,
↪"object": dict}

    if openapi_type == "array" and schema_info is not None and "items" in schema_info:
        enum = schema_info["items"].get("enum")
        if enum:
            return enum
        items_type = schema_info["items"].get("type", "string")
        return [self._map_openapi_type_to_python(items_type)]  # pyright:␣
↪ignore[reportReturnType]
    if schema_info is not None and schema_info.get("enum"):
        return Or(*schema_info["enum"])
    return type_mapping.get(openapi_type, str)
```

_parse_schema(schema)

Source Code in griptape/tools/griptape_cloud_tool/tool.py

```
def _parse_schema(self, schema: dict) -> tuple[str, dict[str, tuple[str, Schema]]]:
    """Parses an openapi schema into a dictionary of activity names and their␣
↪respective descriptions + schemas."""
    activities = {}

    name = schema.get("info", {}).get("title")

    for path, path_info in schema.get("paths", {}).items():
        if not path.startswith("/activities"):
            continue
        for method, method_info in path_info.items():
```

(continues on next page)

(continued from previous page)

```
            if "post" in method.lower():
                activity_name = method_info["operationId"]
                description = method_info.get("description", "")

                activity_schema = self.__extract_schema_from_ref(
                    schema,
                    method_info.get("requestBody", {})
                    .get("content", {})
                    .get("application/json", {})
                    .get("schema", {}),
                )

                activities[activity_name] = (description, activity_schema)

    return name, activities
```

_run_activity(activity_name, params)

Source Code in griptape/tools/griptape_cloud_tool/tool.py

```
def _run_activity(self, activity_name: str, params: dict) -> BaseArtifact:
    """Runs an activity on the tool with the provided parameters."""
    url = urljoin(self.base_url, f"/api/tools/{self.tool_id}/activities/{activity_
↪name}")

    response = requests.post(url, json=params, headers=self.headers)

    response.raise_for_status()

    try:
        return BaseArtifact.from_dict(response.json())
    except ValueError:
        return TextArtifact(response.text)
```

ImageQueryTool

Bases: *BaseTool* “)

Source Code in griptape/tools/image_query/tool.py

```
@define
class ImageQueryTool(BaseTool):
    prompt_driver: BasePromptDriver = field(kw_only=True)
    image_loader: ImageLoader = field(default=Factory(lambda: ImageLoader()), kw_
↪only=True)

    @activity(
        config={
            "description": "This tool can be used to query the contents of images on␣
↪disk.",
            "schema": Schema(
                {
```

(continues on next page)

(continued from previous page)

```
                    Literal(
                        "query",
                        description="A detailed question to be answered using the␣
↪contents of the provided images.",
                    ): str,
                    Literal("image_paths", description="The paths to an image files␣
↪on disk."): Schema([str]),
                },
            ),
        },
    )
    def query_image_from_disk(self, params: dict) -> TextArtifact | ErrorArtifact:
        query = params["values"]["query"]
        image_paths = params["values"]["image_paths"]

        image_artifacts = []
        for image_path in image_paths:
            image_artifacts.append(self.image_loader.load(image_path))

        return cast(
            "TextArtifact",
            self.prompt_driver.run(
                PromptStack.from_artifact(
                    ListArtifact([TextArtifact(query), *image_artifacts]),
                )
            ).to_artifact(),
        )

    @activity(
        config={
            "description": "This tool can be used to query the contents of images in␣
↪memory.",
            "schema": Schema(
                {
                    Literal(
                        "query",
                        description="A detailed question to be answered using the␣
↪contents of the provided images.",
                    ): str,
                    Literal("image_artifacts", description="Image artifact memory␣
↪references."): [
                        {"image_artifact_namespace": str, "image_artifact_name": str},
                    ],
                    "memory_name": str,
                },
            ),
        },
    )
    def query_images_from_memory(self, params: dict[str, Any]) -> TextArtifact |␣
↪ErrorArtifact:
        query = params["values"]["query"]
        image_artifact_references = params["values"]["image_artifacts"]
```

(continues on next page)

(continued from previous page)

```
        memory = self.find_input_memory(params["values"]["memory_name"])

        if memory is None:
            return ErrorArtifact("memory not found")

        image_artifacts = []
        for image_artifact_reference in image_artifact_references:
            try:
                image_artifact = load_artifact_from_memory(
                    memory,
                    image_artifact_reference["image_artifact_namespace"],
                    image_artifact_reference["image_artifact_name"],
                    ImageArtifact,
                )

                image_artifacts.append(cast("ImageArtifact", image_artifact))
            except ValueError:
                # If were unable to parse the artifact as an ImageArtifact, attempt to
                # parse a BlobArtifact and load it as an ImageArtifact.
                blob_artifact = load_artifact_from_memory(
                    memory,
                    image_artifact_reference["image_artifact_namespace"],
                    image_artifact_reference["image_artifact_name"],
                    BlobArtifact,
                )

                image_artifacts.append(self.image_loader.load(blob_artifact.value))
            except Exception as e:
                return ErrorArtifact(str(e))

        return cast(
            "TextArtifact",
            self.prompt_driver.run(
                PromptStack.from_artifact(
                    ListArtifact([TextArtifact(query), *image_artifacts]),
                )
            ).to_artifact(),
        )
```

- `image_loader = field(default=Factory(lambda: ImageLoader()), kw_only=True)` class-attribute instance-attribute
- `prompt_driver = field(kw_only=True)` class-attribute instance-attribute

query_image_from_disk(params)

Source Code in griptape/tools/image_query/tool.py

```
@activity(
    config={
        "description": "This tool can be used to query the contents of images on disk.
↪",
        "schema": Schema(
```

(continues on next page)

(continued from previous page)

```
            {
                Literal(
                    "query",
                    description="A detailed question to be answered using the␣
↪contents of the provided images.",
                ): str,
                Literal("image_paths", description="The paths to an image files on␣
↪disk."): Schema([str]),
            },
        ),
    },
)
def query_image_from_disk(self, params: dict) -> TextArtifact | ErrorArtifact:
    query = params["values"]["query"]
    image_paths = params["values"]["image_paths"]

    image_artifacts = []
    for image_path in image_paths:
        image_artifacts.append(self.image_loader.load(image_path))

    return cast(
        "TextArtifact",
        self.prompt_driver.run(
            PromptStack.from_artifact(
                ListArtifact([TextArtifact(query), *image_artifacts]),
            )
        ).to_artifact(),
    )
```

query_images_from_memory(params)

Source Code in griptape/tools/image_query/tool.py

```
@activity(
    config={
        "description": "This tool can be used to query the contents of images in␣
↪memory.",
        "schema": Schema(
            {
                Literal(
                    "query",
                    description="A detailed question to be answered using the␣
↪contents of the provided images.",
                ): str,
                Literal("image_artifacts", description="Image artifact memory␣
↪references."): [
                    {"image_artifact_namespace": str, "image_artifact_name": str},
                ],
                "memory_name": str,
            },
        ),
    },
```

(continues on next page)

(continued from previous page)

```
)
def query_images_from_memory(self, params: dict[str, Any]) -> TextArtifact |␣
↪ErrorArtifact:
    query = params["values"]["query"]
    image_artifact_references = params["values"]["image_artifacts"]
    memory = self.find_input_memory(params["values"]["memory_name"])

    if memory is None:
        return ErrorArtifact("memory not found")

    image_artifacts = []
    for image_artifact_reference in image_artifact_references:
        try:
            image_artifact = load_artifact_from_memory(
                memory,
                image_artifact_reference["image_artifact_namespace"],
                image_artifact_reference["image_artifact_name"],
                ImageArtifact,
            )

            image_artifacts.append(cast("ImageArtifact", image_artifact))
        except ValueError:
            # If were unable to parse the artifact as an ImageArtifact, attempt to
            # parse a BlobArtifact and load it as an ImageArtifact.
            blob_artifact = load_artifact_from_memory(
                memory,
                image_artifact_reference["image_artifact_namespace"],
                image_artifact_reference["image_artifact_name"],
                BlobArtifact,
            )

            image_artifacts.append(self.image_loader.load(blob_artifact.value))
        except Exception as e:
            return ErrorArtifact(str(e))

    return cast(
        "TextArtifact",
        self.prompt_driver.run(
            PromptStack.from_artifact(
                ListArtifact([TextArtifact(query), *image_artifacts]),
            )
        ).to_artifact(),
    )
```

InpaintingImageGenerationTool

Bases: *BaseImageGenerationTool* “)

Attributes

Name	Type	Description
<a href="./#griptape.tools.InpaintingImageGenerationTool.image_generation_driver" title="image_generation_driver = field(kw_only=True) class-attribute instance-attribute (griptape.tools.InpaintingImageGenerationTool.image_generation_driver)"><code class="language-python">	<a href="../drivers/image_generation#griptape.drivers.image_generation.BaseImageGenerationDriver" title="BaseImageGenerationDriver (griptape.drivers.image_generation.BaseImageGenerationDriver)"><code class="language-python">BaseImageGenerationDrive	The image generation driver used to generate the image.
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.InpaintingImageGenerationTool.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk in output_dir.

continues on next page

Table 388 – continued from previous page

Name	Type	Description
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.InpaintingImageGenerationTool.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk as output_file.

Source Code in griptape/tools/inpainting_image_generation/tool.py

```
@define
class InpaintingImageGenerationTool(BaseImageGenerationTool):
    """A tool that can be used to generate prompted inpaintings of an image.

    Attributes:
        image_generation_driver: The image generation driver used to generate the␣
↪image.
        output_dir: If provided, the generated image will be written to disk in␣
↪output_dir.
        output_file: If provided, the generated image will be written to disk as␣
↪output_file.
    """

    image_generation_driver: BaseImageGenerationDriver = field(kw_only=True)
    image_loader: ImageLoader = field(default=ImageLoader(), kw_only=True)

    @activity(
        config={
            "description": "Modifies an image within a specified mask area using␣
↪image and mask files.",
            "schema": Schema(
                {
                    Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                    Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                    Literal(
                        "image_file",
                        description="The path to an image file to be used as a base␣
```

(continues on next page)

(continued from previous page)

```
↪to generate variations from.",
                    ): str,
                    Literal("mask_file", description="The path to mask image file."):␣
↪str,
                },
            ),
        },
    )
    def image_inpainting_from_file(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
        prompt = params["values"]["prompt"]
        negative_prompt = params["values"]["negative_prompt"]
        image_file = params["values"]["image_file"]
        mask_file = params["values"]["mask_file"]

        input_artifact = self.image_loader.load(image_file)
        mask_artifact = self.image_loader.load(mask_file)

        return self._generate_inpainting(
            prompt, negative_prompt, cast("ImageArtifact", input_artifact), cast(
↪"ImageArtifact", mask_artifact)
        )

    @activity(
        config={
            "description": "Modifies an image within a specified mask area using␣
↪image and mask artifacts in memory.",
            "schema": Schema(
                {
                    Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                    Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                    "memory_name": str,
                    "image_artifact_namespace": str,
                    "image_artifact_name": str,
                    "mask_artifact_namespace": str,
                    "mask_artifact_name": str,
                },
            ),
        },
    )
    def image_inpainting_from_memory(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
        prompt = params["values"]["prompt"]
        negative_prompt = params["values"]["negative_prompt"]
        image_artifact_namespace = params["values"]["image_artifact_namespace"]
        image_artifact_name = params["values"]["image_artifact_name"]
        mask_artifact_namespace = params["values"]["mask_artifact_namespace"]
        mask_artifact_name = params["values"]["mask_artifact_name"]
        memory = self.find_input_memory(params["values"]["memory_name"])
```

(continues on next page)

(continued from previous page)

```
        if memory is None:
            return ErrorArtifact("memory not found")

        try:
            image_artifact = load_artifact_from_memory(
                memory,
                image_artifact_namespace,
                image_artifact_name,
                ImageArtifact,
            )
            mask_artifact = load_artifact_from_memory(
                memory,
                mask_artifact_namespace,
                mask_artifact_name,
                ImageArtifact,
            )
        except ValueError as e:
            return ErrorArtifact(str(e))

        return self._generate_inpainting(
            prompt, negative_prompt, cast("ImageArtifact", image_artifact), cast(
↪"ImageArtifact", mask_artifact)
        )

    def _generate_inpainting(
        self, prompt: str, negative_prompt: str, image_artifact: ImageArtifact, mask_
↪artifact: ImageArtifact
    ) -> ImageArtifact:
        output_artifact = self.image_generation_driver.run_image_inpainting(
            prompts=[prompt], negative_prompts=[negative_prompt], image=image_
↪artifact, mask=mask_artifact
        )

        if self.output_dir or self.output_file:
            self._write_to_file(output_artifact)

        return output_artifact
```

- `image_generation_driver = field(kw_only=True)` class-attribute instance-attribute
- `image_loader = field(default=ImageLoader(), kw_only=True)` class-attribute instance-attribute

_generate_inpainting(prompt, negative_prompt, image_artifact, mask_artifact)

Source Code in griptape/tools/inpainting_image_generation/tool.py

```
def _generate_inpainting(
    self, prompt: str, negative_prompt: str, image_artifact: ImageArtifact, mask_
↪artifact: ImageArtifact
) -> ImageArtifact:
    output_artifact = self.image_generation_driver.run_image_inpainting(
        prompts=[prompt], negative_prompts=[negative_prompt], image=image_artifact,␣
↪mask=mask_artifact
```

(continues on next page)

(continued from previous page)

```
    )

    if self.output_dir or self.output_file:
        self._write_to_file(output_artifact)

    return output_artifact
```

image_inpainting_from_file(params)

Source Code in griptape/tools/inpainting_image_generation/tool.py

```
@activity(
    config={
        "description": "Modifies an image within a specified mask area using image␣
↪and mask files.",
        "schema": Schema(
            {
                Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                Literal(
                    "image_file",
                    description="The path to an image file to be used as a base to␣
↪generate variations from.",
                ): str,
                Literal("mask_file", description="The path to mask image file."): str,
            },
        ),
    },
)
def image_inpainting_from_file(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
    prompt = params["values"]["prompt"]
    negative_prompt = params["values"]["negative_prompt"]
    image_file = params["values"]["image_file"]
    mask_file = params["values"]["mask_file"]

    input_artifact = self.image_loader.load(image_file)
    mask_artifact = self.image_loader.load(mask_file)

    return self._generate_inpainting(
        prompt, negative_prompt, cast("ImageArtifact", input_artifact), cast(
↪"ImageArtifact", mask_artifact)
    )
```

image_inpainting_from_memory(params)

Source Code in griptape/tools/inpainting_image_generation/tool.py

```
@activity(
    config={
```

(continues on next page)

(continued from previous page)

```
        "description": "Modifies an image within a specified mask area using image␣
↪and mask artifacts in memory.",
        "schema": Schema(
            {
                Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                "memory_name": str,
                "image_artifact_namespace": str,
                "image_artifact_name": str,
                "mask_artifact_namespace": str,
                "mask_artifact_name": str,
            },
        ),
    },
)
def image_inpainting_from_memory(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
    prompt = params["values"]["prompt"]
    negative_prompt = params["values"]["negative_prompt"]
    image_artifact_namespace = params["values"]["image_artifact_namespace"]
    image_artifact_name = params["values"]["image_artifact_name"]
    mask_artifact_namespace = params["values"]["mask_artifact_namespace"]
    mask_artifact_name = params["values"]["mask_artifact_name"]
    memory = self.find_input_memory(params["values"]["memory_name"])

    if memory is None:
        return ErrorArtifact("memory not found")

    try:
        image_artifact = load_artifact_from_memory(
            memory,
            image_artifact_namespace,
            image_artifact_name,
            ImageArtifact,
        )
        mask_artifact = load_artifact_from_memory(
            memory,
            mask_artifact_namespace,
            mask_artifact_name,
            ImageArtifact,
        )
    except ValueError as e:
        return ErrorArtifact(str(e))

    return self._generate_inpainting(
        prompt, negative_prompt, cast("ImageArtifact", image_artifact), cast(
↪"ImageArtifact", mask_artifact)
    )
```

OutpaintingImageGenerationTool

Bases: *BaseImageGenerationTool* “)

Attributes

Name	Type	Description
<a href="./#griptape.tools.OutpaintingImageGenerationTool.image_generation_driver" title="image_generation_driver = field(kw_only=True) class-attribute instance-attribute (griptape.tools.OutpaintingImageGenerationTool.image_generation_driver)"><code class="language-python">	<a href="../drivers/image_generation#griptape.drivers.image_generation.BaseImageGenerationDriver" title="BaseImageGenerationDriver (griptape.drivers.image_generation.BaseImageGenerationDriver)"><code class="language-python">BaseImageGenerationDrive	The image generation driver used to generate the image.
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.OutpaintingImageGenerationTool.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk in output_dir.

continues on next page

Table 389 – continued from previous page

Name	Type	Description
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.OutpaintingImageGenerationTool.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk as output_file.

Source Code in griptape/tools/outpainting_image_generation/tool.py

```
@define
class OutpaintingImageGenerationTool(BaseImageGenerationTool):
    """A tool that can be used to generate prompted outpaintings of an image.

    Attributes:
        image_generation_driver: The image generation driver used to generate the␣
↪image.
        output_dir: If provided, the generated image will be written to disk in␣
↪output_dir.
        output_file: If provided, the generated image will be written to disk as␣
↪output_file.
    """

    image_generation_driver: BaseImageGenerationDriver = field(kw_only=True)
    image_loader: ImageLoader = field(default=ImageLoader(), kw_only=True)

    @activity(
        config={
            "description": "Modifies an image outside a specified mask area using␣
↪image and mask files.",
            "schema": Schema(
                {
                    Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                    Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                    Literal(
                        "image_file",
                        description="The path to an image file to be used as a base␣
```

(continues on next page)

(continued from previous page)

```
↪to generate variations from.",
                    ): str,
                    Literal("mask_file", description="The path to mask image file."):␣
↪str,
                },
            ),
        },
    )
    def image_outpainting_from_file(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
        prompt = params["values"]["prompt"]
        negative_prompt = params["values"]["negative_prompt"]
        image_file = params["values"]["image_file"]
        mask_file = params["values"]["mask_file"]

        input_artifact = self.image_loader.load(image_file)
        mask_artifact = self.image_loader.load(mask_file)

        return self._generate_outpainting(prompt, negative_prompt, input_artifact,␣
↪mask_artifact)

    @activity(
        config={
            "description": "Modifies an image outside a specified mask area using␣
↪image and mask artifacts in memory.",
            "schema": Schema(
                {
                    Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                    Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                    "memory_name": str,
                    "image_artifact_namespace": str,
                    "mask_artifact_namespace": str,
                },
            ),
        },
    )
    def image_outpainting_from_memory(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
        prompt = params["values"]["prompt"]
        negative_prompt = params["values"]["negative_prompt"]
        image_artifact_namespace = params["values"]["image_artifact_namespace"]
        image_artifact_name = params["values"]["image_artifact_name"]
        mask_artifact_namespace = params["values"]["mask_artifact_namespace"]
        mask_artifact_name = params["values"]["mask_artifact_name"]
        memory = self.find_input_memory(params["values"]["memory_name"])

        if memory is None:
            return ErrorArtifact("memory not found")

        try:
```

(continues on next page)

(continued from previous page)

```
            image_artifact = load_artifact_from_memory(
                memory,
                image_artifact_namespace,
                image_artifact_name,
                ImageArtifact,
            )
            mask_artifact = load_artifact_from_memory(
                memory,
                mask_artifact_namespace,
                mask_artifact_name,
                ImageArtifact,
            )
        except ValueError as e:
            return ErrorArtifact(str(e))

        return self._generate_outpainting(
            prompt, negative_prompt, cast("ImageArtifact", image_artifact), cast(
↪"ImageArtifact", mask_artifact)
        )

    def _generate_outpainting(
        self, prompt: str, negative_prompt: str, image_artifact: ImageArtifact, mask_
↪artifact: ImageArtifact
    ) -> ImageArtifact | ErrorArtifact:
        output_artifact = self.image_generation_driver.run_image_outpainting(
            prompts=[prompt], negative_prompts=[negative_prompt], image=image_
↪artifact, mask=mask_artifact
        )

        if self.output_dir or self.output_file:
            self._write_to_file(output_artifact)

        return output_artifact
```

- `image_generation_driver = field(kw_only=True)` class-attribute instance-attribute
- `image_loader = field(default=ImageLoader(), kw_only=True)` class-attribute instance-attribute

_generate_outpainting(prompt, negative_prompt, image_artifact, mask_artifact)

Source Code in griptape/tools/outpainting_image_generation/tool.py

```
def _generate_outpainting(
    self, prompt: str, negative_prompt: str, image_artifact: ImageArtifact, mask_
↪artifact: ImageArtifact
) -> ImageArtifact | ErrorArtifact:
    output_artifact = self.image_generation_driver.run_image_outpainting(
        prompts=[prompt], negative_prompts=[negative_prompt], image=image_artifact,␣
↪mask=mask_artifact
    )

    if self.output_dir or self.output_file:
        self._write_to_file(output_artifact)
```

(continues on next page)

(continued from previous page)

```
    return output_artifact
```

image_outpainting_from_file(params)

Source Code in griptape/tools/outpainting_image_generation/tool.py

```
@activity(
    config={
        "description": "Modifies an image outside a specified mask area using image␣
↪and mask files.",
        "schema": Schema(
            {
                Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                Literal(
                    "image_file",
                    description="The path to an image file to be used as a base to␣
↪generate variations from.",
                ): str,
                Literal("mask_file", description="The path to mask image file."): str,
            },
        ),
    },
)
def image_outpainting_from_file(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
    prompt = params["values"]["prompt"]
    negative_prompt = params["values"]["negative_prompt"]
    image_file = params["values"]["image_file"]
    mask_file = params["values"]["mask_file"]

    input_artifact = self.image_loader.load(image_file)
    mask_artifact = self.image_loader.load(mask_file)

    return self._generate_outpainting(prompt, negative_prompt, input_artifact, mask_
↪artifact)
```

image_outpainting_from_memory(params)

Source Code in griptape/tools/outpainting_image_generation/tool.py

```
@activity(
    config={
        "description": "Modifies an image outside a specified mask area using image␣
↪and mask artifacts in memory.",
        "schema": Schema(
            {
                Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
```

(continues on next page)

(continued from previous page)

```
                Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                "memory_name": str,
                "image_artifact_namespace": str,
                "mask_artifact_namespace": str,
            },
        ),
    },
)
def image_outpainting_from_memory(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
    prompt = params["values"]["prompt"]
    negative_prompt = params["values"]["negative_prompt"]
    image_artifact_namespace = params["values"]["image_artifact_namespace"]
    image_artifact_name = params["values"]["image_artifact_name"]
    mask_artifact_namespace = params["values"]["mask_artifact_namespace"]
    mask_artifact_name = params["values"]["mask_artifact_name"]
    memory = self.find_input_memory(params["values"]["memory_name"])

    if memory is None:
        return ErrorArtifact("memory not found")

    try:
        image_artifact = load_artifact_from_memory(
            memory,
            image_artifact_namespace,
            image_artifact_name,
            ImageArtifact,
        )
        mask_artifact = load_artifact_from_memory(
            memory,
            mask_artifact_namespace,
            mask_artifact_name,
            ImageArtifact,
        )
    except ValueError as e:
        return ErrorArtifact(str(e))

    return self._generate_outpainting(
        prompt, negative_prompt, cast("ImageArtifact", image_artifact), cast(
↪"ImageArtifact", mask_artifact)
    )
```

PromptImageGenerationTool

Bases: *BaseImageGenerationTool* “)

Attributes

Name	Type	Description
<a href="./#griptape.tools.PromptImageGenerationTool.image_generation_driver" title="image_generation_driver = field(kw_only=True) class-attribute instance-attribute (griptape.tools.PromptImageGenerationTool.image_generation_driver)"><code class="language-python">	<a href="../drivers/image_generation#griptape.drivers.image_generation.BaseImageGenerationDriver" title="BaseImageGenerationDriver (griptape.drivers.image_generation.BaseImageGenerationDriver)"><code class="language-python">BaseImageGenerationDrive	The image generation driver used to generate the image.
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.PromptImageGenerationTool.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk in output_dir.
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.PromptImageGenerationTool.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk as output_file.

Source Code in griptape/tools/prompt_image_generation/tool.py

```
@define
class PromptImageGenerationTool(BaseImageGenerationTool):
    """A tool that can be used to generate an image from a text prompt.

    Attributes:
        image_generation_driver: The image generation driver used to generate the␣
↪image.
        output_dir: If provided, the generated image will be written to disk in␣
↪output_dir.
        output_file: If provided, the generated image will be written to disk as␣
↪output_file.
    """

    image_generation_driver: BaseImageGenerationDriver = field(kw_only=True)

    @activity(
        config={
            "description": "Generates an image from text prompts.",
            "schema": Schema(
                {
                    Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                    Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                }
            ),
        },
    )
    def generate_image(self, params: dict[str, dict[str, str]]) -> ImageArtifact |␣
↪ErrorArtifact:
        prompt = params["values"]["prompt"]
        negative_prompt = params["values"]["negative_prompt"]

        output_artifact = self.image_generation_driver.run_text_to_image(
            prompts=[prompt], negative_prompts=[negative_prompt]
        )

        if self.output_dir or self.output_file:
            self._write_to_file(output_artifact)

        return output_artifact
```

- `image_generation_driver = field(kw_only=True)` class-attribute instance-attribute

generate_image(params)

Source Code in griptape/tools/prompt_image_generation/tool.py

```
@activity(
    config={
        "description": "Generates an image from text prompts.",
        "schema": Schema(
```

(continues on next page)

(continued from previous page)

```
            {
                Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
            }
        ),
    },
)
def generate_image(self, params: dict[str, dict[str, str]]) -> ImageArtifact |␣
↪ErrorArtifact:
    prompt = params["values"]["prompt"]
    negative_prompt = params["values"]["negative_prompt"]

    output_artifact = self.image_generation_driver.run_text_to_image(
        prompts=[prompt], negative_prompts=[negative_prompt]
    )

    if self.output_dir or self.output_file:
        self._write_to_file(output_artifact)

    return output_artifact
```

PromptSummaryTool

Bases: *BaseTool* “) , *RuleMixin* ”)

Attributes

Name	Type	Description
<a href="./#griptape.tools.PromptSummaryTool.prompt_summary_engine" title="prompt_summary_engine = field(kw_only=True, default=Factory(lambda: PromptSummaryEngine())) class-attribute instance-attribute (griptape.tools.PromptSummaryTool.prompt_summary_engine)"><code class="language-python">	<a href="../engines#griptape.engines.PromptSummaryEngine" title="PromptSummaryEngine (griptape.engines.PromptSummaryEngine)"><code class="language-python">PromptSummaryEngine</co	*PromptSummaryEngine*.

Source Code in griptape/tools/prompt_summary/tool.py

```
@define(kw_only=True)
class PromptSummaryTool(BaseTool, RuleMixin):
    """Tool for using a Prompt Summary Engine.

    Attributes:
        prompt_summary_engine: `PromptSummaryEngine`.
    """

    prompt_summary_engine: PromptSummaryEngine = field(kw_only=True,␣
↪default=Factory(lambda: PromptSummaryEngine()))

    @activity(
        config={
            "description": "Can be used to summarize text content.",
            "schema": Schema(
                {
                    Literal("summary"): Or(
                        str,
                        Schema(
                            {
                                "memory_name": str,
                                "artifact_namespace": str,
                            }
                        ),
                    ),
                }
            ),
        },
    )
    def summarize(self, params: dict) -> BaseArtifact:
        summary = params["values"]["summary"]

        if isinstance(summary, str):
            artifacts = ListArtifact([TextArtifact(summary)])
        else:
            memory = self.find_input_memory(summary["memory_name"])
            artifact_namespace = summary["artifact_namespace"]

            if memory is not None:
                artifacts = memory.load_artifacts(artifact_namespace)
            else:
                return ErrorArtifact("memory not found")

        return self.prompt_summary_engine.summarize_artifacts(artifacts,␣
↪rulesets=self.rulesets)
```

- `prompt_summary_engine = field(kw_only=True, default=Factory(lambda: PromptSummaryEngine()))` class-attribute instance-attribute

summarize(params)

Source Code in griptape/tools/prompt_summary/tool.py

```
@activity(
    config={
        "description": "Can be used to summarize text content.",
        "schema": Schema(
            {
                Literal("summary"): Or(
                    str,
                    Schema(
                        {
                            "memory_name": str,
                            "artifact_namespace": str,
                        }
                    ),
                ),
            }
        ),
    },
)
def summarize(self, params: dict) -> BaseArtifact:
    summary = params["values"]["summary"]

    if isinstance(summary, str):
        artifacts = ListArtifact([TextArtifact(summary)])
    else:
        memory = self.find_input_memory(summary["memory_name"])
        artifact_namespace = summary["artifact_namespace"]

        if memory is not None:
            artifacts = memory.load_artifacts(artifact_namespace)
        else:
            return ErrorArtifact("memory not found")

    return self.prompt_summary_engine.summarize_artifacts(artifacts, rulesets=self.
↪rulesets)
```

QueryTool

Bases: *BaseTool* “) , *RuleMixin* ”)

Source Code in griptape/tools/query/tool.py

```
@define(kw_only=True)
class QueryTool(BaseTool, RuleMixin):
    """Tool for performing a query against data."""

    prompt_driver: BasePromptDriver = field(default=Factory(lambda: Defaults.drivers_
↪config.prompt_driver))

    _rag_engine: RagEngine = field(
        default=Factory(
```

(continues on next page)

(continued from previous page)

```
            lambda self: RagEngine(
                response_stage=ResponseRagStage(
                    response_modules=[
                        PromptResponseRagModule(prompt_driver=self.prompt_driver,␣
↪rulesets=self.rulesets)
                    ],
                ),
            ),
            takes_self=True,
        ),
        alias="_rag_engine",
    )

    @activity(
        config={
            "description": "Can be used to search through textual content.",
            "schema": Schema(
                {
                    Literal("query", description="A natural language search query"):␣
↪str,
                    Literal("content"): Or(
                        str,
                        Schema(
                            {
                                "memory_name": str,
                                "artifact_namespace": str,
                            }
                        ),
                    ),
                }
            ),
        },
    )
    def query(self, params: dict) -> ListArtifact | ErrorArtifact:
        query = params["values"]["query"]
        content = params["values"]["content"]

        if isinstance(content, str):
            text_artifacts = [TextArtifact(content)]
        else:
            memory = self.find_input_memory(content["memory_name"])
            artifact_namespace = content["artifact_namespace"]

            if memory is not None:
                artifacts = memory.load_artifacts(artifact_namespace)
            else:
                return ErrorArtifact("memory not found")

            text_artifacts = [artifact for artifact in artifacts if␣
↪isinstance(artifact, TextArtifact)]

        outputs = self._rag_engine.process(RagContext(query=query, text_chunks=text_
```

(continues on next page)

(continued from previous page)

```
↪artifacts)).outputs

        if len(outputs) > 0:
            return ListArtifact(outputs)
        return ErrorArtifact("query output is empty")
```

- `_rag_engine = field(default=Factory(lambda self: RagEngine(response_stage=ResponseRagStage(respo prompt_driver, rulesets=self.rulesets)])), takes_self=True), alias='_rag_engine')` class-attribute instance-attribute
- `prompt_driver = field(default=Factory(lambda: Defaults.drivers_config. prompt_driver))` class-attribute instance-attribute

query(params)

Source Code in griptape/tools/query/tool.py

```
@activity(
    config={
        "description": "Can be used to search through textual content.",
        "schema": Schema(
            {
                Literal("query", description="A natural language search query"): str,
                Literal("content"): Or(
                    str,
                    Schema(
                        {
                            "memory_name": str,
                            "artifact_namespace": str,
                        }
                    ),
                ),
            }
        ),
    },
)
def query(self, params: dict) -> ListArtifact | ErrorArtifact:
    query = params["values"]["query"]
    content = params["values"]["content"]

    if isinstance(content, str):
        text_artifacts = [TextArtifact(content)]
    else:
        memory = self.find_input_memory(content["memory_name"])
        artifact_namespace = content["artifact_namespace"]

        if memory is not None:
            artifacts = memory.load_artifacts(artifact_namespace)
        else:
            return ErrorArtifact("memory not found")

        text_artifacts = [artifact for artifact in artifacts if isinstance(artifact,␣
↪TextArtifact)]
```

(continues on next page)

(continued from previous page)

```
    outputs = self._rag_engine.process(RagContext(query=query, text_chunks=text_
↪artifacts)).outputs

    if len(outputs) > 0:
        return ListArtifact(outputs)
    return ErrorArtifact("query output is empty")
```

RagTool

Bases: *BaseTool* “)

Attributes

Name	Type	Description
<code class="language-python">	*str*	LLM-friendly RAG engine description.
<code class="language-python">	<code class="language-python">RagEngine</code>	*RagEngine*.

Source Code in griptape/tools/rag/tool.py

```
@define(kw_only=True)
class RagTool(BaseTool):
    """Tool for querying a RAG engine.

    Attributes:
        description: LLM-friendly RAG engine description.
        rag_engine: `RagEngine`.
    """

    description: str = field()
    rag_engine: RagEngine = field()

    @activity(
```

(continues on next page)

(continued from previous page)

```
        config={
            "description": "{{ _self.description }}",
            "schema": Schema({Literal("query", description="A natural language search
↪query"): str}),
        },
    )
    def search(self, params: dict) -> ListArtifact | ErrorArtifact:
        query = params["values"]["query"]

        try:
            artifacts = self.rag_engine.process_query(query).outputs

            outputs = []
            for artifact in artifacts:
                if isinstance(artifact, ListArtifact):
                    outputs.extend(artifact.value)
                else:
                    outputs.append(artifact)

            if len(outputs) > 0:
                return ListArtifact(outputs)
            return ErrorArtifact("query output is empty")

        except Exception as e:
            return ErrorArtifact(f"error querying: {e}")
```

- `description = field()` class-attribute instance-attribute
- `rag_engine = field()` class-attribute instance-attribute

search(params)

Source Code in griptape/tools/rag/tool.py

```
@activity(
    config={
        "description": "{{ _self.description }}",
        "schema": Schema({Literal("query", description="A natural language search
↪query"): str}),
    },
)
def search(self, params: dict) -> ListArtifact | ErrorArtifact:
    query = params["values"]["query"]

    try:
        artifacts = self.rag_engine.process_query(query).outputs

        outputs = []
        for artifact in artifacts:
            if isinstance(artifact, ListArtifact):
                outputs.extend(artifact.value)
            else:
                outputs.append(artifact)
```

(continues on next page)

(continued from previous page)

```
        if len(outputs) > 0:
            return ListArtifact(outputs)
        return ErrorArtifact("query output is empty")

    except Exception as e:
        return ErrorArtifact(f"error querying: {e}")
```

RestApiTool

Bases: *BaseTool* “)

Attributes

Name	Type	Description
<a href="./#griptape.tools.RestApiTool.base_url" title="base_url = field(kw_only=True) class-attribute instance-attribute (griptape.tools.RestApiTool.base_url)"><code class="language-python">	*str*	The base url that will be used for the request.
<a href="./#griptape.tools.RestApiTool.path" title="path = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.RestApiTool.path)"><code class="language-python">	*Optional[str]*	The resource path that will be appended to base_url.
<a href="./#griptape.tools.RestApiTool.description" title="description = field(kw_only=True) class-attribute instance-attribute (griptape.tools.RestApiTool.description)"><code class="language-python">	*str*	A description of what the REST API does.

continues on next page

Table 393 – continued from previous page

Name	Type	Description
<a href="./#griptape.tools.RestApiTool.request_body_schema" title="request_body_schema = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.RestApiTool.request_body_schema)"><c class="language-python">	*Optional[str]*	A JSON schema string describing the request body. Recommended for PUT, POST, and PATCH requests.
<a href="./#griptape.tools.RestApiTool.request_query_params_sch title="request_query_params_sch = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.RestApiTool.request_query_params_sch class="language-python">	*Optional[str]*	A JSON schema string describing the available query parameters.
<a href="./#griptape.tools.RestApiTool.request_path_params_sche title="request_path_params_sche = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.RestApiTool.request_path_params_sche class="language-python">	*Optional[str]*	A JSON schema string describing the available path parameters. The schema must describe an array of string values.

continues on next page

Table 393 – continued from previous page

Name	Type	Description
<a href="./#griptape.tools.RestApiTool.response_body_schema" title="response_body_schema = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.RestApiTool.response_body_schema)"><code class="language-python">	*Optional[str]*	A JSON schema string describing the response body.
<a href="./#griptape.tools.RestApiTool.request_headers" title="request_headers = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.RestApiTool.request_headers)"><code class="language-python">	*Optional[dict[str, str]]*	Headers to include in the requests.

Source Code in griptape/tools/rest_api/tool.py

```
@define
class RestApiTool(BaseTool):
    """A tool for making REST API requests.

    Attributes:
        base_url: The base url that will be used for the request.
        path: The resource path that will be appended to base_url.
        description: A description of what the REST API does.
        request_body_schema: A JSON schema string describing the request body.␣
↪Recommended for PUT, POST, and PATCH requests.
        request_query_params_schema: A JSON schema string describing the available␣
↪query parameters.
        request_path_params_schema: A JSON schema string describing the available␣
↪path parameters. The schema must describe an array of string values.
        response_body_schema: A JSON schema string describing the response body.
        request_headers: Headers to include in the requests.
    """

    base_url: str = field(kw_only=True)
    path: Optional[str] = field(default=None, kw_only=True)
    description: str = field(kw_only=True)
```

(continues on next page)

(continued from previous page)

```
    request_path_params_schema: Optional[str] = field(default=None, kw_only=True)
    request_query_params_schema: Optional[str] = field(default=None, kw_only=True)
    request_body_schema: Optional[str] = field(default=None, kw_only=True)
    response_body_schema: Optional[str] = field(default=None, kw_only=True)
    request_headers: Optional[dict[str, str]] = field(default=None, kw_only=True)

    @property
    def full_url(self) -> str:
        return self._build_url(self.base_url, path=self.path)

    @activity(
        config={
            "description": dedent(
                """
                This tool can be used to make a put request to the rest api url: {{ _
↪self.full_url }}
                This rest api has the following description: {{ _self.description }}
                {% if _self.request_body_schema %}The request body must follow this␣
↪JSON schema: {{ _self.request_body_schema }}{% endif %}
                {% if _self.response_body_schema %}The response body must follow this␣
↪JSON schema: {{ _self.response_body_schema }}{% endif %}
                """,
            ),
            "schema": Schema({Literal("body", description="The request body."): dict}
↪),
        },
    )
    def put(self, params: dict) -> BaseArtifact:
        from requests import exceptions, put

        values = params["values"]
        base_url = self.base_url
        path = self.path
        body = values["body"]
        url = self._build_url(base_url, path=path)

        try:
            response = put(url, json=body, timeout=30, headers=self.request_headers)

            return TextArtifact(response.text)
        except exceptions.RequestException as err:
            return ErrorArtifact(str(err))

    @activity(
        config={
            "description": dedent(
                """
                This tool can be used to make a patch request to the rest api url: {{␣
↪_self.full_url }}
                This rest api has the following description: {{ _self.description }}
                {% if _self.request_path_parameters %}The request path parameters␣
↪must follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
```

(continues on next page)

(continued from previous page)

```
                {% if _self.request_body_schema %}The request body must follow this␣
↪JSON schema: {{ _self.request_body_schema }}{% endif %}
                {% if _self.response_body_schema %}The response body must follow this␣
↪JSON schema: {{ _self.response_body_schema }}{% endif %}
                """,
            ),
            "schema": Schema(
                {
                    Literal("path_params", description="The request path parameters.
↪"): Schema([str]),
                    Literal("body", description="The request body."): dict,
                },
            ),
        },
    )
    def patch(self, params: dict) -> BaseArtifact:
        from requests import exceptions, patch

        values = params["values"]
        base_url = self.base_url
        path = self.path
        body = values["body"]
        path_params = values["path_params"]
        url = self._build_url(base_url, path=path, path_params=path_params)

        try:
            response = patch(url, json=body, timeout=30, headers=self.request_headers)
            return TextArtifact(response.text)
        except exceptions.RequestException as err:
            return ErrorArtifact(str(err))

    @activity(
        config={
            "description": dedent(
                """
                This tool can be used to make a post request to the rest api url: {{ _
↪self.full_url }}
                This rest api has the following description: {{ _self.description }}
                {% if _self.request_body_schema %}The request body must follow this␣
↪JSON schema: {{ _self.request_body_schema }}{% endif %}
                {% if _self.response_body_schema %}The response body must follow this␣
↪JSON schema: {{ _self.response_body_schema }}{% endif %}
                """,
            ),
            "schema": Schema({Literal("body", description="The request body."): dict}
↪),
        },
    )
    def post(self, params: dict) -> BaseArtifact:
        from requests import exceptions, post

        values = params["values"]
```

(continues on next page)

(continued from previous page)

```
        base_url = self.base_url
        path = self.path
        url = self._build_url(base_url, path=path)
        body = values["body"]

        try:
            response = post(url, json=body, timeout=30, headers=self.request_headers)
            return TextArtifact(response.text)
        except exceptions.RequestException as err:
            return ErrorArtifact(str(err))

    @activity(
        config={
            "description": dedent(
                """
                This tool can be used to make a get request to the rest api url: {{ _
↪self.full_url }}
                This rest api has the following description: {{ _self.description }}
                {% if _self.request_path_parameters %}The request path parameters␣
↪must follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
                {% if _self.request_query_parameters %}The request query parameters␣
↪must follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
                {% if _self.response_body_schema %}The response body must follow this␣
↪JSON schema: {{ _self.response_body_schema }}{% endif %}
                """,
            ),
            "schema": schema.Optional(
                Schema(
                    {
                        schema.Optional(Literal("query_params", description="The␣
↪request query parameters.")): dict,
                        schema.Optional(Literal("path_params", description="The␣
↪request path parameters.")): Schema(
                            [str]
                        ),
                    },
                ),
            ),
        },
    )
    def get(self, params: dict) -> BaseArtifact:
        from requests import exceptions, get

        values = params["values"]
        base_url = self.base_url
        path = self.path

        query_params = {}
        path_params = []
        if values:
            query_params = values.get("query_params", {})
            path_params = values.get("path_params", [])
```

(continues on next page)

(continued from previous page)

```
        url = self._build_url(base_url, path=path, path_params=path_params)

        try:
            response = get(url, params=query_params, timeout=30, headers=self.request_
↪headers)
            return TextArtifact(response.text)
        except exceptions.RequestException as err:
            return ErrorArtifact(str(err))

    @activity(
        config={
            "description": dedent(
                """
                This tool can be used to make a delete request to the rest api url: {
↪{ _self.full_url }}
                This rest api has the following description: {{ _self.description }}
                {% if _self.request_path_parameters %}The request path parameters␣
↪must follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
                {% if _self.request_query_parameters %}The request query parameters␣
↪must follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
                """,
            ),
            "schema": Schema(
                {
                    schema.Optional(Literal("query_params", description="The request␣
↪query parameters.")): dict,
                    schema.Optional(Literal("path_params", description="The request␣
↪path parameters.")): Schema([str]),
                },
            ),
        },
    )
    def delete(self, params: dict) -> BaseArtifact:
        from requests import delete, exceptions

        values = params["values"]
        base_url = self.base_url
        path = self.path

        query_params = values.get("query_params", {})
        path_params = values.get("path_params", [])
        url = self._build_url(base_url, path=path, path_params=path_params)

        try:
            response = delete(url, params=query_params, timeout=30, headers=self.
↪request_headers)
            return TextArtifact(response.text)
        except exceptions.RequestException as err:
            return ErrorArtifact(str(err))

    def _build_url(self, base_url: str, path: Optional[str] = None, path_params:␣
↪Optional[list] = None) -> str:
```

(continues on next page)

(continued from previous page)

```
        url = ""

        if path:
            url += path.strip("/")

        if path_params:
            url += f"/{str.join(/, map(str, path_params))}"

        return urljoin(base_url.strip("/"), url)
```

- `base_url = field(kw_only=True)` class-attribute instance-attribute
- `description = field(kw_only=True)` class-attribute instance-attribute
- `full_url` property
- `path = field(default=None, kw_only=True)` class-attribute instance-attribute
- `request_body_schema = field(default=None, kw_only=True)` class-attribute instance-attribute
- `request_headers = field(default=None, kw_only=True)` class-attribute instance-attribute
- `request_path_params_schema = field(default=None, kw_only=True)` class-attribute instance-attribute
- `request_query_params_schema = field(default=None, kw_only=True)` class-attribute instance-attribute
- `response_body_schema = field(default=None, kw_only=True)` class-attribute instance-attribute

_build_url(base_url, path=None, path_params=None)

Source Code in griptape/tools/rest_api/tool.py

```
def _build_url(self, base_url: str, path: Optional[str] = None, path_params:␣
↪Optional[list] = None) -> str:
    url = ""

    if path:
        url += path.strip("/")

    if path_params:
        url += f"/{str.join(/, map(str, path_params))}"

    return urljoin(base_url.strip("/"), url)
```

delete(params)

Source Code in griptape/tools/rest_api/tool.py

```
@activity(
    config={
        "description": dedent(
            """
            This tool can be used to make a delete request to the rest api url: {{ _
↪self.full_url }}
```

(continues on next page)

(continued from previous page)

```
            This rest api has the following description: {{ _self.description }}
            {% if _self.request_path_parameters %}The request path parameters must
↪follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
            {% if _self.request_query_parameters %}The request query parameters must
↪follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
            """,
        ),
        "schema": Schema(
            {
                schema.Optional(Literal("query_params", description="The request
↪query parameters.")): dict,
                schema.Optional(Literal("path_params", description="The request path
↪parameters.")): Schema([str]),
            },
        ),
    },
)
def delete(self, params: dict) -> BaseArtifact:
    from requests import delete, exceptions

    values = params["values"]
    base_url = self.base_url
    path = self.path

    query_params = values.get("query_params", {})
    path_params = values.get("path_params", [])
    url = self._build_url(base_url, path=path, path_params=path_params)

    try:
        response = delete(url, params=query_params, timeout=30, headers=self.request_
↪headers)
        return TextArtifact(response.text)
    except exceptions.RequestException as err:
        return ErrorArtifact(str(err))
```

get(params)

Source Code in griptape/tools/rest_api/tool.py

```
@activity(
    config={
        "description": dedent(
            """
            This tool can be used to make a get request to the rest api url: {{ _self.
↪full_url }}
            This rest api has the following description: {{ _self.description }}
            {% if _self.request_path_parameters %}The request path parameters must
↪follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
            {% if _self.request_query_parameters %}The request query parameters must
↪follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
            {% if _self.response_body_schema %}The response body must follow this
↪JSON schema: {{ _self.response_body_schema }}{% endif %}
```

(continues on next page)

(continued from previous page)

```
            """,
        ),
        "schema": schema.Optional(
            Schema(
                {
                    schema.Optional(Literal("query_params", description="The request␣
↪query parameters.")): dict,
                    schema.Optional(Literal("path_params", description="The request␣
↪path parameters.")): Schema(
                        [str]
                    ),
                },
            ),
        ),
    },
)
def get(self, params: dict) -> BaseArtifact:
    from requests import exceptions, get

    values = params["values"]
    base_url = self.base_url
    path = self.path

    query_params = {}
    path_params = []
    if values:
        query_params = values.get("query_params", {})
        path_params = values.get("path_params", [])
    url = self._build_url(base_url, path=path, path_params=path_params)

    try:
        response = get(url, params=query_params, timeout=30, headers=self.request_
↪headers)
        return TextArtifact(response.text)
    except exceptions.RequestException as err:
        return ErrorArtifact(str(err))
```

patch(params)

Source Code in griptape/tools/rest_api/tool.py

```
@activity(
    config={
        "description": dedent(
            """
            This tool can be used to make a patch request to the rest api url: {{ _
↪self.full_url }}
            This rest api has the following description: {{ _self.description }}
            {% if _self.request_path_parameters %}The request path parameters must␣
↪follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
            {% if _self.request_body_schema %}The request body must follow this JSON␣
↪schema: {{ _self.request_body_schema }}{% endif %}
```

(continues on next page)

(continued from previous page)

```
            {% if _self.response_body_schema %}The response body must follow this␣
↪JSON schema: {{ _self.response_body_schema }}{% endif %}
            """,
        ),
        "schema": Schema(
            {
                Literal("path_params", description="The request path parameters."):␣
↪Schema([str]),
                Literal("body", description="The request body."): dict,
            },
        ),
    },
)
def patch(self, params: dict) -> BaseArtifact:
    from requests import exceptions, patch

    values = params["values"]
    base_url = self.base_url
    path = self.path
    body = values["body"]
    path_params = values["path_params"]
    url = self._build_url(base_url, path=path, path_params=path_params)

    try:
        response = patch(url, json=body, timeout=30, headers=self.request_headers)
        return TextArtifact(response.text)
    except exceptions.RequestException as err:
        return ErrorArtifact(str(err))
```

post(params)

Source Code in griptape/tools/rest_api/tool.py

```
@activity(
    config={
        "description": dedent(
            """
            This tool can be used to make a post request to the rest api url: {{ _
↪self.full_url }}
            This rest api has the following description: {{ _self.description }}
            {% if _self.request_body_schema %}The request body must follow this JSON␣
↪schema: {{ _self.request_body_schema }}{% endif %}
            {% if _self.response_body_schema %}The response body must follow this␣
↪JSON schema: {{ _self.response_body_schema }}{% endif %}
            """,
        ),
        "schema": Schema({Literal("body", description="The request body."): dict}),
    },
)
def post(self, params: dict) -> BaseArtifact:
    from requests import exceptions, post
```

(continues on next page)

(continued from previous page)

```
    values = params["values"]
    base_url = self.base_url
    path = self.path
    url = self._build_url(base_url, path=path)
    body = values["body"]

    try:
        response = post(url, json=body, timeout=30, headers=self.request_headers)
        return TextArtifact(response.text)
    except exceptions.RequestException as err:
        return ErrorArtifact(str(err))
```

put(params)

Source Code in griptape/tools/rest_api/tool.py

```
@activity(
    config={
        "description": dedent(
            """
            This tool can be used to make a put request to the rest api url: {{ _self.
↪full_url }}
            This rest api has the following description: {{ _self.description }}
            {% if _self.request_body_schema %}The request body must follow this JSON␣
↪schema: {{ _self.request_body_schema }}{% endif %}
            {% if _self.response_body_schema %}The response body must follow this␣
↪JSON schema: {{ _self.response_body_schema }}{% endif %}
            """,
        ),
        "schema": Schema({Literal("body", description="The request body."): dict}),
    },
)
def put(self, params: dict) -> BaseArtifact:
    from requests import exceptions, put

    values = params["values"]
    base_url = self.base_url
    path = self.path
    body = values["body"]
    url = self._build_url(base_url, path=path)

    try:
        response = put(url, json=body, timeout=30, headers=self.request_headers)

        return TextArtifact(response.text)
    except exceptions.RequestException as err:
        return ErrorArtifact(str(err))
```

SqlTool

Bases: *BaseTool* “)

Source Code in griptape/tools/sql/tool.py

```
@define
class SqlTool(BaseTool):
    sql_loader: SqlLoader = field(kw_only=True)
    schema_name: Optional[str] = field(default=None, kw_only=True)
    table_name: str = field(kw_only=True)
    table_description: Optional[str] = field(default=None, kw_only=True)
    engine_name: Optional[str] = field(default=None, kw_only=True)

    @property
    def full_table_name(self) -> str:
        return f"{self.schema_name}.{self.table_name}" if self.schema_name else self.
↪table_name

    @property
    def table_schema(self) -> Optional[str]:
        return self.sql_loader.sql_driver.get_table_schema(self.full_table_name,␣
↪schema=self.schema_name)

    @activity(
        config={
            "description": "Can be used to execute{% if _self.engine_name %} {{ _self.
↪engine_name }}{% endif %} SQL SELECT queries "
            "in table {{ _self.full_table_name }}. "
            "Make sure the `SELECT` statement contains enough columns to get an␣
↪answer without knowing "
            "the original question. "
            "Be creative when you use `WHERE` statements: you can use wildcards,␣
↪`LOWER()`, and other functions "
            "to get better results. "
            "You can use JOINs if more tables are available in other tools.\n"
            "{{ _self.table_name }} schema: {{ _self.table_schema }}\n"
            "{% if _self.table_description %}{{ _self.table_name }} description: {{ _
↪self.table_description }}{% endif %}",
            "schema": Schema({"sql_query": str}),
        },
    )
    def execute_query(self, params: dict) -> ListArtifact | InfoArtifact |␣
↪ErrorArtifact:
        try:
            query = params["values"]["sql_query"]
            rows = self.sql_loader.load(query)
        except Exception as e:
            return ErrorArtifact(f"error executing query: {e}")

        if len(rows) > 0:
            return rows
        return InfoArtifact("No results found")
```

- `engine_name = field(default=None, kw_only=True)` class-attribute instance-attribute

- `full_table_name` property
- `schema_name = field(default=None, kw_only=True)` class-attribute instance-attribute
- `sql_loader = field(kw_only=True)` class-attribute instance-attribute
- `table_description = field(default=None, kw_only=True)` class-attribute instance-attribute
- `table_name = field(kw_only=True)` class-attribute instance-attribute
- `table_schema` property

execute_query(params)

Source Code in griptape/tools/sql/tool.py

```
@activity(
    config={
        "description": "Can be used to execute{% if _self.engine_name %} {{ _self.
↪engine_name }}{% endif %} SQL SELECT queries "
        "in table {{ _self.full_table_name }}. "
        "Make sure the `SELECT` statement contains enough columns to get an answer␣
↪without knowing "
        "the original question. "
        "Be creative when you use `WHERE` statements: you can use wildcards,␣
↪`LOWER()`, and other functions "
        "to get better results. "
        "You can use JOINs if more tables are available in other tools.\n"
        "{{ _self.table_name }} schema: {{ _self.table_schema }}\n"
        "{% if _self.table_description %}{{ _self.table_name }} description: {{ _self.
↪table_description }}{% endif %}",
        "schema": Schema({"sql_query": str}),
    },
)
def execute_query(self, params: dict) -> ListArtifact | InfoArtifact | ErrorArtifact:
    try:
        query = params["values"]["sql_query"]
        rows = self.sql_loader.load(query)
    except Exception as e:
        return ErrorArtifact(f"error executing query: {e}")

    if len(rows) > 0:
        return rows
    return InfoArtifact("No results found")
```

StructureRunTool

Bases: *BaseTool* “)

Attributes

Name	Type	Description
<a href="./#griptape.tools.StructureRunTool.description" title="description = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.tools.StructureRunTool.description)"><code class="language-python">	*str*	A description of what the Structure does.
<a href="./#griptape.tools.StructureRunTool.structure_run_driver" title="structure_run_driver = field(kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.tools.StructureRunTool.structure_run_driver)"><code class="language-python">	<a href="../drivers/structure_run#griptape.drivers.structure_run.BaseStructureRunDriver" title="BaseStructureRunDriver (griptape.drivers.structure_run.BaseStructureRunDriver)"><code class="language-python">BaseStructureRunDriver</c	Driver to run the Structure.

Source Code in griptape/tools/structure_run/tool.py

```
@define
class StructureRunTool(BaseTool):
    """Tool for running a Structure.

    Attributes:
        description: A description of what the Structure does.
        structure_run_driver: Driver to run the Structure.
    """

    description: str = field(kw_only=True, metadata={"serializable": True})
    structure_run_driver: BaseStructureRunDriver = field(kw_only=True, metadata={
↪"serializable": True})

    @activity(
        config={
            "description": "Can be used to run a Gen AI Builder Structure with the␣
↪following description: {{ _self.description }}",
            "schema": Schema(
                {
                    Literal("args", description="A list of string arguments to submit␣
```

(continues on next page)

(continued from previous page)

```
↪to the Structure Run"): Schema(
                        [str]
                    )
                },
            ),
        },
    )
    def run_structure(self, params: dict) -> BaseArtifact:
        args: list[str] = params["values"]["args"]

        return self.structure_run_driver.run(*[TextArtifact(arg) for arg in args])
```

- `description = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structure_run_driver = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

run_structure(params)

Source Code in griptape/tools/structure_run/tool.py

```
@activity(
    config={
        "description": "Can be used to run a Gen AI Builder Structure with the␣
↪following description: {{ _self.description }}",
        "schema": Schema(
            {
                Literal("args", description="A list of string arguments to submit to␣
↪the Structure Run"): Schema(
                    [str]
                )
            },
        ),
    },
)
def run_structure(self, params: dict) -> BaseArtifact:
    args: list[str] = params["values"]["args"]

    return self.structure_run_driver.run(*[TextArtifact(arg) for arg in args])
```

StructuredOutputTool

Bases: *BaseTool* “)

Source Code in griptape/tools/structured_output/tool.py

```
@define
class StructuredOutputTool(BaseTool):
    output_schema: Union[Schema, type[BaseModel]] = field(kw_only=True)

    @activity(
        config={
```

(continues on next page)

(continued from previous page)

```
            "description": "Used to provide the final response which ends this␣
↪conversation.",
            "schema": lambda _self: _self.output_schema,
        }
    )
    def provide_output(self, params: dict) -> BaseArtifact:
        return JsonArtifact(params["values"])
```

- `output_schema = field(kw_only=True)` class-attribute instance-attribute

provide_output(params)

Source Code in griptape/tools/structured_output/tool.py

```
@activity(
    config={
        "description": "Used to provide the final response which ends this␣
↪conversation.",
        "schema": lambda _self: _self.output_schema,
    }
)
def provide_output(self, params: dict) -> BaseArtifact:
    return JsonArtifact(params["values"])
```

TextToSpeechTool

Bases: *ArtifactFileOutputMixin* “) , *BaseTool* ”)

Attributes

Name	Type	Description
<a href="./#griptape.tools.TextToSpeechTool.text_to_speech_driv ti- tle="text_to_speech_driv = field(kw_only=True) class-attribute instance-attribute (griptape.tools.TextToSpeechTool.text_to_speech_driv class="language-python">	<a href="../drivers/text_to_speech#griptape.drivers.text_to_speech.BaseTextToSpeechDriver" title="BaseTextToSpeechDriver (griptape.drivers.text_to_speech.BaseTextToSpeechDriver)"><code class="language-python">BaseTextToSpeechDriver</c	The text to audio generation driver used to generate the speech audio.

continues on next page

Table 395 – continued from previous page

Name	Type	Description
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.TextToSpeechTool.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated audio will be written to disk in output_dir.
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.TextToSpeechTool.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated audio will be written to disk as output_file.

Source Code in griptape/tools/text_to_speech/tool.py

```
@define
class TextToSpeechTool(ArtifactFileOutputMixin, BaseTool):
    """A tool that can be used to generate speech from input text.

    Attributes:
        text_to_speech_driver: The text to audio generation driver used to generate␣
↪the speech audio.
        output_dir: If provided, the generated audio will be written to disk in␣
↪output_dir.
        output_file: If provided, the generated audio will be written to disk as␣
↪output_file.
    """
```

(continues on next page)

(continued from previous page)

```
    text_to_speech_driver: BaseTextToSpeechDriver = field(kw_only=True)

    @activity(
        config={
            "description": "Can be used to generate speech from the provided input␣
↪text.",
            "schema": Schema({Literal("text", description="The literal text to be␣
↪converted to speech."): str}),
        },
    )
    def text_to_speech(self, params: dict[str, Any]) -> AudioArtifact | ErrorArtifact:
        text = params["values"]["text"]

        output_artifact = self.text_to_speech_driver.run_text_to_audio(prompts=[text])

        if self.output_dir or self.output_file:
            self._write_to_file(output_artifact)

        return output_artifact
```

- `text_to_speech_driver = field(kw_only=True)` class-attribute instance-attribute

text_to_speech(params)

Source Code in griptape/tools/text_to_speech/tool.py

```
@activity(
    config={
        "description": "Can be used to generate speech from the provided input text.",
        "schema": Schema({Literal("text", description="The literal text to be␣
↪converted to speech."): str}),
    },
)
def text_to_speech(self, params: dict[str, Any]) -> AudioArtifact | ErrorArtifact:
    text = params["values"]["text"]

    output_artifact = self.text_to_speech_driver.run_text_to_audio(prompts=[text])

    if self.output_dir or self.output_file:
        self._write_to_file(output_artifact)

    return output_artifact
```

VariationImageGenerationTool

Bases: *BaseImageGenerationTool* “)

Attributes

Name	Type	Description
<a href="./#griptape.tools.VariationImageGenerationTool.image_generation_driver" title="image_generation_driver = field(kw_only=True) class-attribute instance-attribute (griptape.tools.VariationImageGenerationTool.image_generation_driver)"><code class="language-python">	<a href="../drivers/image_generation#griptape.drivers.image_generation.BaseImageGenerationDriver" title="BaseImageGenerationDriver (griptape.drivers.image_generation.BaseImageGenerationDriver)"><code class="language-python">BaseImageGenerationDrive	The image generation driver used to generate the image.
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.VariationImageGenerationTool.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk in output_dir.
<a href="../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.VariationImageGenerationTool.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk as output_file.

Source Code in griptape/tools/variation_image_generation/tool.py

```
@define
class VariationImageGenerationTool(BaseImageGenerationTool):
    """A tool that can be used to generate prompted variations of an image.

    Attributes:
        image_generation_driver: The image generation driver used to generate the␣
↪image.
        output_dir: If provided, the generated image will be written to disk in␣
↪output_dir.
        output_file: If provided, the generated image will be written to disk as␣
↪output_file.
    """

    image_generation_driver: BaseImageGenerationDriver = field(kw_only=True)
    image_loader: ImageLoader = field(default=ImageLoader(), kw_only=True)

    @activity(
        config={
            "description": "Generates a variation of a given input image file.",
            "schema": Schema(
                {
                    Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                    Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                    Literal(
                        "image_file",
                        description="The path to an image file to be used as a base␣
↪to generate variations from.",
                    ): str,
                },
            ),
        },
    )
    def image_variation_from_file(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
        prompt = params["values"]["prompt"]
        negative_prompt = params["values"]["negative_prompt"]
        image_file = params["values"]["image_file"]

        image_artifact = self.image_loader.load(image_file)

        return self._generate_variation(prompt, negative_prompt, image_artifact)

    @activity(
        config={
            "description": "Generates a variation of a given input image artifact in␣
↪memory.",
            "schema": Schema(
                {
                    Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                    Literal("negative_prompt", description=BaseImageGenerationTool.
```

(continues on next page)

(continued from previous page)

```
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                    "memory_name": str,
                    "artifact_namespace": str,
                    "artifact_name": str,
                },
            ),
        },
    )
    def image_variation_from_memory(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
        prompt = params["values"]["prompt"]
        negative_prompt = params["values"]["negative_prompt"]
        artifact_namespace = params["values"]["artifact_namespace"]
        artifact_name = params["values"]["artifact_name"]
        memory = self.find_input_memory(params["values"]["memory_name"])

        if memory is None:
            return ErrorArtifact("memory not found")

        try:
            image_artifact = load_artifact_from_memory(memory, artifact_namespace,␣
↪artifact_name, ImageArtifact)
        except ValueError as e:
            return ErrorArtifact(str(e))

        return self._generate_variation(prompt, negative_prompt, cast("ImageArtifact",
↪ image_artifact))

    def _generate_variation(
        self, prompt: str, negative_prompt: str, image_artifact: ImageArtifact
    ) -> ImageArtifact | ErrorArtifact:
        output_artifact = self.image_generation_driver.run_image_variation(
            prompts=[prompt], negative_prompts=[negative_prompt], image=image_artifact
        )

        if self.output_dir or self.output_file:
            self._write_to_file(output_artifact)

        return output_artifact
```

- `image_generation_driver = field(kw_only=True)` class-attribute instance-attribute
- `image_loader = field(default=ImageLoader(), kw_only=True)` class-attribute instance-attribute

_generate_variation(prompt, negative_prompt, image_artifact)

Source Code in griptape/tools/variation_image_generation/tool.py

```
def _generate_variation(
    self, prompt: str, negative_prompt: str, image_artifact: ImageArtifact
) -> ImageArtifact | ErrorArtifact:
    output_artifact = self.image_generation_driver.run_image_variation(
        prompts=[prompt], negative_prompts=[negative_prompt], image=image_artifact
```

(continues on next page)

(continued from previous page)

```
    )

    if self.output_dir or self.output_file:
        self._write_to_file(output_artifact)

    return output_artifact
```

image_variation_from_file(params)

Source Code in griptape/tools/variation_image_generation/tool.py

```
@activity(
    config={
        "description": "Generates a variation of a given input image file.",
        "schema": Schema(
            {
                Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                Literal(
                    "image_file",
                    description="The path to an image file to be used as a base to␣
↪generate variations from.",
                ): str,
            },
        ),
    },
)
def image_variation_from_file(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
    prompt = params["values"]["prompt"]
    negative_prompt = params["values"]["negative_prompt"]
    image_file = params["values"]["image_file"]

    image_artifact = self.image_loader.load(image_file)

    return self._generate_variation(prompt, negative_prompt, image_artifact)
```

image_variation_from_memory(params)

Source Code in griptape/tools/variation_image_generation/tool.py

```
@activity(
    config={
        "description": "Generates a variation of a given input image artifact in␣
↪memory.",
        "schema": Schema(
            {
                Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                Literal("negative_prompt", description=BaseImageGenerationTool.
```

(continues on next page)

(continued from previous page)

```
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                "memory_name": str,
                "artifact_namespace": str,
                "artifact_name": str,
            },
        ),
    },
)
def image_variation_from_memory(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
    prompt = params["values"]["prompt"]
    negative_prompt = params["values"]["negative_prompt"]
    artifact_namespace = params["values"]["artifact_namespace"]
    artifact_name = params["values"]["artifact_name"]
    memory = self.find_input_memory(params["values"]["memory_name"])

    if memory is None:
        return ErrorArtifact("memory not found")

    try:
        image_artifact = load_artifact_from_memory(memory, artifact_namespace,␣
↪artifact_name, ImageArtifact)
    except ValueError as e:
        return ErrorArtifact(str(e))

    return self._generate_variation(prompt, negative_prompt, cast("ImageArtifact",␣
↪image_artifact))
```

VectorStoreTool

Bases: *BaseTool* “)

Attributes

Name	Type	Description
<code class="language-python">	*str*	LLM-friendly vector DB description.

continues on next page

Table 397 – continued from previous page

Name	Type	Description
<code class="language-python">	<code class="language-python">BaseVectorStoreDriver</coc	*BaseVectorStoreDriver*.
<c class="language-python">	*dict[str, Any]*	Optional dictionary of vector store driver query parameters.
<code class="language-python">	<code class="language-python">Callable[[list[Entry]], BaseArtifact]</code>	Optional lambda for processing vector store driver query output `Entry`s.

Source Code in griptape/tools/vector_store/tool.py

```
@define(kw_only=True)
class VectorStoreTool(BaseTool):
    """A tool for querying a vector database.

    Attributes:
        description: LLM-friendly vector DB description.
        vector_store_driver: `BaseVectorStoreDriver`.
        query_params: Optional dictionary of vector store driver query parameters.
```

(continues on next page)

(continued from previous page)

```
        process_query_output: Optional lambda for processing vector store driver␣
↪query output `Entry`s.
    """

    DEFAULT_TOP_N = 5

    description: str = field()
    vector_store_driver: BaseVectorStoreDriver = field()
    query_params: dict[str, Any] = field(factory=dict)
    process_query_output: Callable[[list[BaseVectorStoreDriver.Entry]], BaseArtifact]␣
↪= field(
        default=Factory(lambda: lambda es: ListArtifact([e.to_artifact() for e in␣
↪es])),
    )

    @activity(
        config={
            "description": "Can be used to search a database with the following␣
↪description: {{ _self.description }}",
            "schema": Schema(
                {
                    Literal(
                        "query",
                        description="A natural language search query to run against␣
↪the vector database",
                    ): str,
                },
            ),
        },
    )
    def search(self, params: dict) -> BaseArtifact:
        query = params["values"]["query"]

        try:
            return self.process_query_output(self.vector_store_driver.query(query,␣
↪**self.query_params))
        except Exception as e:
            return ErrorArtifact(f"error querying vector store: {e}")
```

- `DEFAULT_TOP_N = 5` class-attribute instance-attribute
- `description = field()` class-attribute instance-attribute
- `process_query_output = field(default=Factory(lambda: lambda es: ListArtifact([e.to_artifact() for e in es])))` class-attribute instance-attribute
- `query_params = field(factory=dict)` class-attribute instance-attribute
- `vector_store_driver = field()` class-attribute instance-attribute

search(params)

Source Code in griptape/tools/vector_store/tool.py

```
@activity(
    config={
        "description": "Can be used to search a database with the following␣
↪description: {{ _self.description }}",
        "schema": Schema(
            {
                Literal(
                    "query",
                    description="A natural language search query to run against the␣
↪vector database",
                ): str,
            },
        ),
    },
)
def search(self, params: dict) -> BaseArtifact:
    query = params["values"]["query"]

    try:
        return self.process_query_output(self.vector_store_driver.query(query, **self.
↪query_params))
    except Exception as e:
        return ErrorArtifact(f"error querying vector store: {e}")
```

WebScraperTool

Bases: *BaseTool* “)

Source Code in griptape/tools/web_scraper/tool.py

```
@define
class WebScraperTool(BaseTool):
    web_loader: WebLoader = field(default=Factory(lambda: WebLoader()), kw_only=True)
    text_chunker: TextChunker = field(default=Factory(lambda: TextChunker(max_
↪tokens=400)), kw_only=True)

    @activity(
        config={
            "description": "Can be used to browse a web page and load its content",
            "schema": Schema({Literal("url", description="Valid HTTP URL"): str}),
        },
    )
    def get_content(self, params: dict) -> ListArtifact | ErrorArtifact:
        url = params["values"]["url"]

        try:
            result = self.web_loader.load(url)
            chunks = self.text_chunker.chunk(result)

            return ListArtifact(chunks)
```

(continues on next page)

(continued from previous page)

```
        except Exception as e:
            return ErrorArtifact("Error getting page content: " + str(e))
```

- `text_chunker = field(default=Factory(lambda: TextChunker(max_tokens=400)), kw_only=True)` class-attribute instance-attribute
- `web_loader = field(default=Factory(lambda: WebLoader()), kw_only=True)` class-attribute instance-attribute

get_content(params)

Source Code in griptape/tools/web_scraper/tool.py

```
@activity(
    config={
        "description": "Can be used to browse a web page and load its content",
        "schema": Schema({Literal("url", description="Valid HTTP URL"): str}),
    },
)
def get_content(self, params: dict) -> ListArtifact | ErrorArtifact:
    url = params["values"]["url"]

    try:
        result = self.web_loader.load(url)
        chunks = self.text_chunker.chunk(result)

        return ListArtifact(chunks)
    except Exception as e:
        return ErrorArtifact("Error getting page content: " + str(e))
```

WebSearchTool

Bases: *BaseTool* “)

Source Code in griptape/tools/web_search/tool.py

```
@define
class WebSearchTool(BaseTool):
    web_search_driver: BaseWebSearchDriver = field(kw_only=True)

    @activity(
        config={
            "description": "Can be used for searching the web via the {{ _self.web_
↪search_driver.__class__.__name__}}.",
            "schema": Schema(
                {
                    Literal(
                        "query",
                        description="Search engine request that returns a list of␣
↪pages with titles, descriptions, and URLs",
                    ): str,
                },
            ),
```

(continues on next page)

(continued from previous page)

```
        },
    )
    def search(self, values: dict) -> ListArtifact | ErrorArtifact:
        # `BaseWebSearchDriver.query` already has a parameter named `query`, so we␣
↪need to pop it from the values
        # to avoid passing it twice.
        query = values.pop("query")

        try:
            return self.web_search_driver.search(query, **values)
        except Exception as e:
            return ErrorArtifact(f"Error searching {query} with {self.web_search_
↪driver.__class__.__name__}: {e}")
```

- `web_search_driver = field(kw_only=True)` class-attribute instance-attribute

search(values)

Source Code in griptape/tools/web_search/tool.py

```
@activity(
    config={
        "description": "Can be used for searching the web via the {{ _self.web_search_
↪driver.__class__.__name__}}.",
        "schema": Schema(
            {
                Literal(
                    "query",
                    description="Search engine request that returns a list of pages␣
↪with titles, descriptions, and URLs",
                ): str,
            },
        ),
    },
)
def search(self, values: dict) -> ListArtifact | ErrorArtifact:
    # `BaseWebSearchDriver.query` already has a parameter named `query`, so we need␣
↪to pop it from the values
    # to avoid passing it twice.
    query = values.pop("query")

    try:
        return self.web_search_driver.search(query, **values)
    except Exception as e:
        return ErrorArtifact(f"Error searching {query} with {self.web_search_driver.__
↪class__.__name__}: {e}")
```

audio_transcription

tool

Bases: *BaseTool* “)

Source Code in griptape/tools/audio_transcription/tool.py

```
@define
class AudioTranscriptionTool(BaseTool):
    """A tool that can be used to generate transcriptions from input audio."""

    audio_transcription_driver: BaseAudioTranscriptionDriver = field(kw_only=True)
    audio_loader: AudioLoader = field(default=Factory(lambda: AudioLoader()), kw_
↪only=True)

    @activity(
        config={
            "description": "This tool can be used to generate transcriptions of audio␣
↪files on disk.",
            "schema": Schema({Literal("path", description="The paths to an audio file␣
↪on disk."): str}),
        },
    )
    def transcribe_audio_from_disk(self, params: dict) -> TextArtifact |␣
↪ErrorArtifact:
        audio_path = params["values"]["path"]

        audio_artifact = self.audio_loader.load(audio_path)

        return self.audio_transcription_driver.run(audio_artifact)

    @activity(
        config={
            "description": "This tool can be used to generate the transcription of an␣
↪audio artifact in memory.",
            "schema": Schema({"schema": Schema({"memory_name": str, "artifact_
↪namespace": str, "artifact_name": str})}),
        },
    )
    def transcribe_audio_from_memory(self, params: dict[str, Any]) -> TextArtifact |␣
↪ErrorArtifact:
        memory = self.find_input_memory(params["values"]["memory_name"])
        artifact_namespace = params["values"]["artifact_namespace"]
        artifact_name = params["values"]["artifact_name"]

        if memory is None:
            return ErrorArtifact("memory not found")

        audio_artifact = cast(
            "AudioArtifact",
            load_artifact_from_memory(memory, artifact_namespace, artifact_name,␣
↪AudioArtifact),
        )

        return self.audio_transcription_driver.run(audio_artifact)
```

- `audio_loader = field(default=Factory(lambda: AudioLoader()), kw_only=True)` class-attribute instance-attribute
- `audio_transcription_driver = field(kw_only=True)` class-attribute instance-attribute

transcribe_audio_from_disk(params)

Source Code in griptape/tools/audio_transcription/tool.py

```
@activity(
    config={
        "description": "This tool can be used to generate transcriptions of audio␣
↪files on disk.",
        "schema": Schema({Literal("path", description="The paths to an audio file on␣
↪disk."): str}),
    },
)
def transcribe_audio_from_disk(self, params: dict) -> TextArtifact | ErrorArtifact:
    audio_path = params["values"]["path"]

    audio_artifact = self.audio_loader.load(audio_path)

    return self.audio_transcription_driver.run(audio_artifact)
```

transcribe_audio_from_memory(params)

Source Code in griptape/tools/audio_transcription/tool.py

```
@activity(
    config={
        "description": "This tool can be used to generate the transcription of an␣
↪audio artifact in memory.",
        "schema": Schema({"schema": Schema({"memory_name": str, "artifact_namespace":␣
↪str, "artifact_name": str})}),
    },
)
def transcribe_audio_from_memory(self, params: dict[str, Any]) -> TextArtifact |␣
↪ErrorArtifact:
    memory = self.find_input_memory(params["values"]["memory_name"])
    artifact_namespace = params["values"]["artifact_namespace"]
    artifact_name = params["values"]["artifact_name"]

    if memory is None:
        return ErrorArtifact("memory not found")

    audio_artifact = cast(
        "AudioArtifact",
        load_artifact_from_memory(memory, artifact_namespace, artifact_name,␣
↪AudioArtifact),
    )

    return self.audio_transcription_driver.run(audio_artifact)
```

base_griptape_cloud_tool

Bases: *BaseTool* “) , ABC

Attributes

Name	Type	Description
<a href="base_griptape_cloud_tool.mdx#griptape.tools.base_griptape_cloud_tool.BaseGriptapeCloudTool.base_url" title="base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_ 'https://cloud.griptape.ai')), kw_only=True) class-attribute instance-attribute (griptape.tools.base_griptape_cloud_tool.BaseGriptapeCloudTool.base_url)"><code class="language-python">	*str*	Base URL for the Gen AI Builder Knowledge Base API.
<a href="base_griptape_cloud_tool.mdx#griptape.tools.base_griptape_cloud_tool.BaseGriptapeCloudTool.api_key" title="api_key = field(default=Factory(lambda: os.getenv('GT_CLOUD_ kw_only=True) class-attribute instance-attribute (griptape.tools.base_griptape_cloud_tool.BaseGriptapeCloudTool.api_key)"><code class="language-python">	*Optional[str]*	API key for Gen AI Builder.

continues on next page

Table 398 – continued from previous page

Name	Type	Description
<a href="base_griptape_cloud_tool.mdx#griptape.tools.base_griptape_cloud_tool.BaseGriptapeCloudTool.headers" title="headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.api_key}'}, takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.tools.base_griptape_cloud_tool.BaseGriptapeCloudTool.headers)"><code class="language-python">	*dict*	Headers for the Gen AI Builder Knowledge Base API.

Source Code in griptape/tools/base_griptape_cloud_tool.py

```
@define
class BaseGriptapeCloudTool(BaseTool, ABC):
    """Base class for Gen AI Builder clients.

    Attributes:
        base_url: Base URL for the Gen AI Builder Knowledge Base API.
        api_key: API key for Gen AI Builder.
        headers: Headers for the Gen AI Builder Knowledge Base API.
    """

    base_url: str = field(
        default=Factory(lambda: os.getenv("GT_CLOUD_BASE_URL", "https://cloud.
↪griptape.ai")),
        kw_only=True,
    )
    api_key: Optional[str] = field(default=Factory(lambda: os.getenv("GT_CLOUD_API_KEY
↪")), kw_only=True)
    headers: dict = field(
        default=Factory(lambda self: {"Authorization": f"Bearer {self.api_key}"},␣
↪takes_self=True),
        kw_only=True,
    )
```

- `api_key = field(default=Factory(lambda: os.getenv('GT_CLOUD_API_KEY')), kw_only=True)` class-attribute instance-attribute
- `base_url = field(default=Factory(lambda: os.getenv('GT_CLOUD_BASE_URL', 'https://cloud.griptape.ai')), kw_only=True)` class-attribute instance-attribute
- `headers = field(default=Factory(lambda self: {'Authorization': f'Bearer {self.`

`api_key}'}, takes_self=True), kw_only=True)` class-attribute instance-attribute

base_image_generation_tool

Bases: *ArtifactFileOutputMixin* “) , *BaseTool* ”)

Source Code in griptape/tools/base_image_generation_tool.py

```
@define
class BaseImageGenerationTool(ArtifactFileOutputMixin, BaseTool):
    """A base class for tools that generate images from text prompts."""

    PROMPT_DESCRIPTION = "Features and qualities to include in the generated image,␣
↪descriptive and succinct."
    NEGATIVE_PROMPT_DESCRIPTION = (
        "Features and qualities to avoid in the generated image. Affirmatively␣
↪describe "
        "what to avoid, for example: to avoid the color red, include red "
        "rather than no red."
    )
```

- `NEGATIVE_PROMPT_DESCRIPTION = "Features and qualities to avoid in the generated image. Affirmatively describe what to avoid, for example: to avoid the color red, include 'red' rather than 'no red'."` class-attribute instance-attribute
- `PROMPT_DESCRIPTION = 'Features and qualities to include in the generated image, descriptive and succinct.'` class-attribute instance-attribute

base_tool

- `logger = logging.getLogger(Defaults.logging_config.logger_name)` module-attribute

Bases: *ActivityMixin* “) , *SerializableMixin* ”) , *RunnableMixin['BaseTool'* “) , ABC

Attributes

Name	Type	Description
<code class="language-python">	*str*	Tool name.

continues on next page

Table 399 – continued from previous page

Name	Type	Description
<code class="language-python">	<code class="language-python">Optional[list[TaskMemory]]	TaskMemory available in tool activities. Gets automatically set if None.
<code class="language-python">	<code class="language-python">Optional[dict[str, list[TaskMemory]]]</code>	TaskMemory that activities write to be default. Gets automatically set if None.
<a href="base_tool.mdx#griptape.tools.base_tool.BaseTool.install_dependencies_on_init" title="install_dependencies_on_init = field(default=True, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.tools.base_tool.BaseTool.install_dependencies_on_init)"><code class="language-python">	*bool*	Determines whether dependencies from the tool requirements.txt file are installed in init.

continues on next page

Table 399 – continued from previous page

Name	Type	Description
<a href="base_tool.mdx#griptape.tools.base_tool.BaseTool.dependencies_install_directory" title="dependencies_install_directory = field(default=None, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.tools.base_tool.BaseTool.dependencies_install_directory)"><code class="language-python">	*Optional[str]*	Custom dependency install directory.
<a href="base_tool.mdx#griptape.tools.base_tool.BaseTool.verbose" title="verbose = field(default=False, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.tools.base_tool.BaseTool.verbose)"><code class="language-python">	*bool*	Determines whether tool operations (such as dependency installation) should be verbose.
<a href="base_tool.mdx#griptape.tools.base_tool.BaseTool.off_prompt" title="off_prompt = field(default=False, kw_only=True, metadata={'serializable': True}) class-attribute instance-attribute (griptape.tools.base_tool.BaseTool.off_prompt)"><code class="language-python">	*bool*	Determines whether tool activity output goes to the output memory.

Source Code in griptape/tools/base_tool.py

```
@define
class BaseTool(ActivityMixin, SerializableMixin, RunnableMixin["BaseTool"], ABC):
    """Abstract class for all tools to inherit from for.

    Attributes:
```

(continues on next page)

(continued from previous page)

```
        name: Tool name.
        input_memory: TaskMemory available in tool activities. Gets automatically set␣
↪if None.
        output_memory: TaskMemory that activities write to be default. Gets␣
↪automatically set if None.
        install_dependencies_on_init: Determines whether dependencies from the tool␣
↪requirements.txt file are installed in init.
        dependencies_install_directory: Custom dependency install directory.
        verbose: Determines whether tool operations (such as dependency installation)␣
↪should be verbose.
        off_prompt: Determines whether tool activity output goes to the output memory.
    """

    REQUIREMENTS_FILE = "requirements.txt"

    name: str = field(
        default=Factory(lambda self: self.__class__.__name__, takes_self=True),
        kw_only=True,
        metadata={"serializable": True},
    )
    input_memory: Optional[list[TaskMemory]] = field(default=None, kw_only=True,␣
↪metadata={"serializable": True})
    output_memory: Optional[dict[str, list[TaskMemory]]] = field(
        default=None, kw_only=True, metadata={"serializable": True}
    )
    install_dependencies_on_init: bool = field(default=True, kw_only=True, metadata={
↪"serializable": True})
    dependencies_install_directory: Optional[str] = field(default=None, kw_only=True,␣
↪metadata={"serializable": True})
    verbose: bool = field(default=False, kw_only=True, metadata={"serializable": True}
↪)
    off_prompt: bool = field(default=False, kw_only=True, metadata={"serializable":␣
↪True})

    def __attrs_post_init__(self) -> None:
        if (
            self.install_dependencies_on_init
            and self.has_requirements
            and not self.are_requirements_met(self.requirements_path)
        ):
            self.install_dependencies(os.environ.copy())

    @output_memory.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_output_memory(self, _: Attribute, output_memory: dict[str,␣
↪Optional[list[TaskMemory]]]) -> None:
        if output_memory:
            for activity_name, memory_list in output_memory.items():
                if not self.find_activity(activity_name):
                    raise ValueError(f"activity {activity_name} doesnt exist")
                if memory_list is None:
                    raise ValueError(f"memory list for activity {activity_name} cant␣
```

(continues on next page)

(continued from previous page)

```
↪be None")

                output_memory_names = [memory.name for memory in memory_list]

                if len(output_memory_names) > len(set(output_memory_names)):
                    raise ValueError(f"memory names have to be unique in activity
↪{activity_name} output")

    @property
    def requirements_path(self) -> str:
        return os.path.join(self.abs_dir_path, self.REQUIREMENTS_FILE)

    @property
    def abs_file_path(self) -> str:
        return os.path.abspath(inspect.getfile(self.__class__))

    @property
    def abs_dir_path(self) -> str:
        return os.path.dirname(self.abs_file_path)

    @property
    def has_requirements(self) -> bool:
        return os.path.exists(self.requirements_path)

    # This method has to remain a method and cant be decorated with @property because
    # of the max depth recursion issue in `self.activities`.
    def schema(self) -> dict:
        full_schema = Schema(Or(*self.activity_schemas()), description=f"{self.name}␣
↪action schema.")

        return full_schema.json_schema(f"{self.name} ToolAction Schema")

    def activity_schemas(self) -> list[Schema]:
        schemas = []

        for activity in self.activities():
            schema_dict: dict[Literal | schema.Optional, Any] = {
                Literal("name"): self.name,
                Literal("path", description=self.activity_description(activity)):␣
↪self.activity_name(activity),
            }

            activity_schema = self.activity_schema(activity)
            # If no schema is defined, we just make `input` optional instead of␣
↪omitting it.
            # This works better with lower-end models that may accidentally pass in␣
↪an empty dict.
            if activity_schema is None:
                schema_dict[schema.Optional("input")] = {}
            else:
                schema_dict[Literal("input")] = activity_schema.schema
```

(continues on next page)

(continued from previous page)

```
            schemas.append(Schema(schema_dict))

        return schemas

    def run(self, activity: Callable, subtask: ActionsSubtask, action: ToolAction) ->␣
↪BaseArtifact:
        try:
            output = self.before_run(activity, subtask, action)

            output = self.try_run(activity, subtask, action, output)

            output = self.after_run(activity, subtask, action, output)
        except Exception as e:
            logging.debug(traceback.format_exc())
            output = ErrorArtifact(str(e), exception=e)

        return output

    def before_run(self, activity: Callable, subtask: ActionsSubtask, action:␣
↪ToolAction) -> Optional[dict]:
        super().before_run()

        return action.input

    @observable(tags=["Tool.run()"])
    def try_run(
        self,
        activity: Callable,
        subtask: ActionsSubtask,
        action: ToolAction,
        value: Optional[dict],
    ) -> BaseArtifact:
        activity_result = activity(deepcopy(value))

        if isinstance(activity_result, BaseArtifact):
            result = activity_result
        else:
            logging.warning("Activity result is not an artifact; converting result to␣
↪InfoArtifact")

            if activity_result is None:
                result = InfoArtifact("Tool returned an empty value")
            else:
                result = InfoArtifact(activity_result)

        return result

    def after_run(
        self,
        activity: Callable,
        subtask: ActionsSubtask,
        action: ToolAction,
```

(continues on next page)

(continued from previous page)

```
        value: BaseArtifact,
    ) -> BaseArtifact:
        super().after_run()

        if self.output_memory:
            output_memories = self.output_memory[getattr(activity, "name")] or []
            for memory in output_memories:
                value = memory.process_output(activity, subtask, value)

            return value
        return value

    def validate(self) -> bool:
        if not os.path.exists(self.requirements_path):
            raise Exception(f"{self.REQUIREMENTS_FILE} not found")
        return True

    def tool_dir(self) -> str:
        class_file = inspect.getfile(self.__class__)

        return os.path.dirname(os.path.abspath(class_file))

    def install_dependencies(self, env: Optional[dict[str, str]] = None) -> None:
        env = env or {}

        command = [sys.executable, "-m", "pip", "install", "-r", "requirements.txt"]

        if self.dependencies_install_directory is None:
            command.extend(["-U"])
        else:
            command.extend(["-t", self.dependencies_install_directory])

        subprocess.run(
            command,
            env=env,
            cwd=self.tool_dir(),
            stdout=None if self.verbose else subprocess.DEVNULL,
            stderr=None if self.verbose else subprocess.DEVNULL,
            check=False,
        )

    def find_input_memory(self, memory_name: str) -> Optional[TaskMemory]:
        if self.input_memory:
            return next((m for m in self.input_memory if m.name == memory_name), None)
        return None

    def to_native_tool_name(self, activity: Callable) -> str:
        """Converts a Tools name and an Activity into to a native tool name.

        The native tool name is a combination of the Tools name and the Activitys␣
↪name.
        The Tools name may only contain letters and numbers, and the Activitys name␣
```

(continues on next page)

(continued from previous page)

```
↪may only contain letters, numbers, and underscores.

        Args:
            activity: Activity to convert

        Returns:
            str: Native tool name.
        """
        tool_name = self.name
        if re.match(r"^[a-zA-Z0-9]+$", tool_name) is None:
            raise ValueError("Tool name can only contain letters and numbers.")

        activity_name = self.activity_name(activity)
        if re.match(r"^[a-zA-Z0-9_]+$", activity_name) is None:
            raise ValueError("Activity name can only contain letters, numbers, and␣
↪underscores.")

        return f"{tool_name}_{activity_name}"

    def are_requirements_met(self, requirements_path: str) -> bool:
        requirements = Path(requirements_path).read_text().splitlines()

        try:
            for requirement in requirements:
                importlib.metadata.version(requirement)
            return True
        except importlib.metadata.PackageNotFoundError:
            return False
```

- `REQUIREMENTS_FILE = 'requirements.txt'` class-attribute instance-attribute
- `abs_dir_path` property
- `abs_file_path` property
- `dependencies_install_directory = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `has_requirements` property
- `input_memory = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `install_dependencies_on_init = field(default=True, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `name = field(default=Factory(lambda self: self.__class__.__name__, takes_self=True), kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `off_prompt = field(default=False, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `output_memory = field(default=None, kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `requirements_path` property

- `verbose = field(default=False, kw_only=True, metadata={'serializable': True})`
  class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/tools/base_tool.py

```
def __attrs_post_init__(self) -> None:
    if (
        self.install_dependencies_on_init
        and self.has_requirements
        and not self.are_requirements_met(self.requirements_path)
    ):
        self.install_dependencies(os.environ.copy())
```

activity_schemas()

Source Code in griptape/tools/base_tool.py

```
def activity_schemas(self) -> list[Schema]:
    schemas = []

    for activity in self.activities():
        schema_dict: dict[Literal | schema.Optional, Any] = {
            Literal("name"): self.name,
            Literal("path", description=self.activity_description(activity)): self.
↪activity_name(activity),
        }

        activity_schema = self.activity_schema(activity)
        # If no schema is defined, we just make `input` optional instead of omitting␣
↪it.
        # This works better with lower-end models that may accidentally pass in an␣
↪empty dict.
        if activity_schema is None:
            schema_dict[schema.Optional("input")] = {}
        else:
            schema_dict[Literal("input")] = activity_schema.schema

        schemas.append(Schema(schema_dict))

    return schemas
```

after_run(activity, subtask, action, value)

Source Code in griptape/tools/base_tool.py

```
def after_run(
    self,
    activity: Callable,
    subtask: ActionsSubtask,
    action: ToolAction,
    value: BaseArtifact,
```

(continues on next page)

(continued from previous page)

```
) -> BaseArtifact:
    super().after_run()

    if self.output_memory:
        output_memories = self.output_memory[getattr(activity, "name")] or []
        for memory in output_memories:
            value = memory.process_output(activity, subtask, value)

        return value
    return value
```

are_requirements_met(requirements_path)

Source Code in griptape/tools/base_tool.py

```
def are_requirements_met(self, requirements_path: str) -> bool:
    requirements = Path(requirements_path).read_text().splitlines()

    try:
        for requirement in requirements:
            importlib.metadata.version(requirement)
        return True
    except importlib.metadata.PackageNotFoundError:
        return False
```

before_run(activity, subtask, action)

Source Code in griptape/tools/base_tool.py

```
def before_run(self, activity: Callable, subtask: ActionsSubtask, action: ToolAction)␣
↪-> Optional[dict]:
    super().before_run()

    return action.input
```

find_input_memory(memory_name)

Source Code in griptape/tools/base_tool.py

```
def find_input_memory(self, memory_name: str) -> Optional[TaskMemory]:
    if self.input_memory:
        return next((m for m in self.input_memory if m.name == memory_name), None)
    return None
```

install_dependencies(env=None)

Source Code in griptape/tools/base_tool.py

```
def install_dependencies(self, env: Optional[dict[str, str]] = None) -> None:
    env = env or {}
```

(continues on next page)

(continued from previous page)

```
    command = [sys.executable, "-m", "pip", "install", "-r", "requirements.txt"]

    if self.dependencies_install_directory is None:
        command.extend(["-U"])
    else:
        command.extend(["-t", self.dependencies_install_directory])

    subprocess.run(
        command,
        env=env,
        cwd=self.tool_dir(),
        stdout=None if self.verbose else subprocess.DEVNULL,
        stderr=None if self.verbose else subprocess.DEVNULL,
        check=False,
    )
```

run(activity, subtask, action)

Source Code in griptape/tools/base_tool.py

```
def run(self, activity: Callable, subtask: ActionsSubtask, action: ToolAction) ->␣
↪BaseArtifact:
    try:
        output = self.before_run(activity, subtask, action)

        output = self.try_run(activity, subtask, action, output)

        output = self.after_run(activity, subtask, action, output)
    except Exception as e:
        logging.debug(traceback.format_exc())
        output = ErrorArtifact(str(e), exception=e)

    return output
```

schema()

Source Code in griptape/tools/base_tool.py

```
def schema(self) -> dict:
    full_schema = Schema(Or(*self.activity_schemas()), description=f"{self.name}␣
↪action schema.")

    return full_schema.json_schema(f"{self.name} ToolAction Schema")
```

to_native_tool_name(activity)

Converts a Tool's name and an Activity into to a native tool name.

The native tool name is a combination of the Tool's name and the Activity's name. The Tool's name may only contain letters and numbers, and the Activity's name may only contain letters, numbers, and underscores.

Parameters

Name	Type	Description	Default
activity	*Callable*	Activity to convert 	*required*

Returns

Name	Type	Description
str	*str*	Native tool name.

Source Code in griptape/tools/base_tool.py

```
def to_native_tool_name(self, activity: Callable) -> str:
    """Converts a Tools name and an Activity into to a native tool name.

    The native tool name is a combination of the Tools name and the Activitys name.
    The Tools name may only contain letters and numbers, and the Activitys name may␣
↪only contain letters, numbers, and underscores.

    Args:
        activity: Activity to convert

    Returns:
        str: Native tool name.
    """
    tool_name = self.name
    if re.match(r"^[a-zA-Z0-9]+$", tool_name) is None:
        raise ValueError("Tool name can only contain letters and numbers.")

    activity_name = self.activity_name(activity)
    if re.match(r"^[a-zA-Z0-9_]+$", activity_name) is None:
        raise ValueError("Activity name can only contain letters, numbers, and␣
↪underscores.")

    return f"{tool_name}_{activity_name}"
```

tool_dir()

Source Code in griptape/tools/base_tool.py

```
def tool_dir(self) -> str:
    class_file = inspect.getfile(self.__class__)

    return os.path.dirname(os.path.abspath(class_file))
```

try_run(activity, subtask, action, value)

Source Code in griptape/tools/base_tool.py

```
@observable(tags=["Tool.run()"])
def try_run(
    self,
    activity: Callable,
    subtask: ActionsSubtask,
    action: ToolAction,
    value: Optional[dict],
) -> BaseArtifact:
    activity_result = activity(deepcopy(value))

    if isinstance(activity_result, BaseArtifact):
        result = activity_result
    else:
        logging.warning("Activity result is not an artifact; converting result to␣
↪InfoArtifact")

        if activity_result is None:
            result = InfoArtifact("Tool returned an empty value")
        else:
            result = InfoArtifact(activity_result)

    return result
```

validate()

Source Code in griptape/tools/base_tool.py

```
def validate(self) -> bool:
    if not os.path.exists(self.requirements_path):
        raise Exception(f"{self.REQUIREMENTS_FILE} not found")
    return True
```

validate*output_memory*(, output_memory)

Source Code in griptape/tools/base_tool.py

```
@output_memory.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_output_memory(self, _: Attribute, output_memory: dict[str,␣
↪Optional[list[TaskMemory]]]) -> None:
    if output_memory:
        for activity_name, memory_list in output_memory.items():
            if not self.find_activity(activity_name):
                raise ValueError(f"activity {activity_name} doesnt exist")
            if memory_list is None:
                raise ValueError(f"memory list for activity {activity_name} cant be␣
↪None")

            output_memory_names = [memory.name for memory in memory_list]
```

(continues on next page)

(continued from previous page)

```
            if len(output_memory_names) > len(set(output_memory_names)):
                raise ValueError(f"memory names have to be unique in activity
↪{activity_name} output")
```

calculator

tool

Bases: *BaseTool* “)

Source Code in griptape/tools/calculator/tool.py

```
class CalculatorTool(BaseTool):
    @activity(
        config={
            "description": "Can be used for computing simple numerical or algebraic␣
↪calculations in Python",
            "schema": Schema(
                {
                    Literal(
                        "expression",
                        description="Arithmetic expression parsable in pure Python.␣
↪Single line only. "
                        "Dont use variables. Dont use any imports or external␣
↪libraries",
                    ): str,
                },
            ),
        },
    )
    def calculate(self, params: dict) -> BaseArtifact:
        import numexpr  # pyright: ignore[reportMissingImports]

        try:
            expression = params["values"]["expression"]

            return TextArtifact(numexpr.evaluate(expression))
        except Exception as e:
            return ErrorArtifact(f"error calculating: {e}")
```

calculate(params)

Source Code in griptape/tools/calculator/tool.py

```
@activity(
    config={
        "description": "Can be used for computing simple numerical or algebraic␣
↪calculations in Python",
        "schema": Schema(
            {
                Literal(
```

(continues on next page)

(continued from previous page)

```
                    "expression",
                    description="Arithmetic expression parsable in pure Python.␣
↪Single line only. "
                    "Dont use variables. Dont use any imports or external libraries",
                ): str,
            },
        ),
    },
)
def calculate(self, params: dict) -> BaseArtifact:
    import numexpr  # pyright: ignore[reportMissingImports]

    try:
        expression = params["values"]["expression"]

        return TextArtifact(numexpr.evaluate(expression))
    except Exception as e:
        return ErrorArtifact(f"error calculating: {e}")
```

computer

tool

Bases: *BaseTool* “)

Source Code in griptape/tools/computer/tool.py

```
@define
class ComputerTool(BaseTool):
    local_workdir: Optional[str] = field(default=None, kw_only=True)
    container_workdir: str = field(default="/griptape", kw_only=True)
    env_vars: dict = field(factory=dict, kw_only=True)
    dockerfile_path: str = field(
        default=Factory(lambda self: f"{os.path.join(self.tool_dir(), resources/
↪Dockerfile)}", takes_self=True),
        kw_only=True,
    )
    requirements_txt_path: str = field(
        default=Factory(lambda self: f"{os.path.join(self.tool_dir(), resources/
↪requirements.txt)}", takes_self=True),
        kw_only=True,
    )
    docker_client: DockerClient = field(
        default=Factory(lambda self: self.default_docker_client(), takes_self=True),
        kw_only=True,
    )

    _tempdir: Optional[tempfile.TemporaryDirectory] = field(default=None, kw_
↪only=True)

    def __attrs_post_init__(self) -> None:
        super().__attrs_post_init__()
```

(continues on next page)

(continued from previous page)

```
        if self.local_workdir:
            Path(self.local_workdir).mkdir(parents=True, exist_ok=True)
        else:
            self._tempdir = tempfile.TemporaryDirectory()
            self.local_workdir = self._tempdir.name

    @docker_client.validator  # pyright: ignore[reportAttributeAccessIssue]
    def validate_docker_client(self, _: Attribute, docker_client: DockerClient) ->␣
↪None:
        if not docker_client:
            raise ValueError("Docker client cant be initialized: make sure the Docker␣
↪daemon is running")

    def install_dependencies(self, env: Optional[dict[str, str]] = None) -> None:
        super().install_dependencies(env)

        self.remove_existing_container(self.container_name(self))
        self.build_image(self)

    @activity(
        config={
            "description": "Can be used to execute Python code to solve any␣
↪programmatic tasks and access and analyze"
            " files in the file system. If you need to use code output use `print`␣
↪statements. "
            "You have access to the following external Python libraries: "
            "{{ _self.dependencies() }}",
            "schema": Schema(
                {
                    Literal("code", description="Python code to execute"): str,
                    Literal(
                        "filename",
                        description="name of the file to put the Python code in␣
↪before executing it",
                    ): str,
                },
            ),
        },
    )
    def execute_code(self, params: dict) -> BaseArtifact:
        code = params["values"]["code"]
        filename = params["values"]["filename"]

        return self.execute_code_in_container(filename, code)

    @activity(
        config={
            "description": "Can be used to execute shell commands in Linux",
            "schema": Schema({Literal("command", description="shell command to execute
↪"): str}),
        },
```

(continues on next page)

(continued from previous page)

```
    )
    def execute_command(self, params: dict) -> BaseArtifact:
        command = params["values"]["command"]

        return self.execute_command_in_container(command)

    def execute_command_in_container(self, command: str) -> BaseArtifact:
        from docker.models.containers import Container

        try:
            binds = {self.local_workdir: {"bind": self.container_workdir, "mode": "rw
↪"}} if self.local_workdir else None

            container = self.docker_client.containers.run(  # pyright:␣
↪ignore[reportCallIssue]
                self.image_name(self),
                environment=self.env_vars,
                command=command,
                name=self.container_name(self),
                volumes=binds,  # pyright: ignore[reportArgumentType] According to␣
↪the [docs](https://docker-py.readthedocs.io/en/stable/containers.html), the type of␣
↪`volumes` is dict[str, dict[str, str]].
                stdout=True,
                stderr=True,
                detach=True,
            )

            if isinstance(container, Container):
                container.wait()

                stderr = container.logs(stdout=False, stderr=True).decode().strip()
                stdout = container.logs(stdout=True, stderr=False).decode().strip()

                container.stop()
                container.remove()

                if stderr:
                    return ErrorArtifact(stderr)
                return TextArtifact(stdout)
            return ErrorArtifact("error running container")
        except Exception as e:
            return ErrorArtifact(f"error executing command: {e}")

    def execute_code_in_container(self, filename: str, code: str) -> BaseArtifact:
        container_file_path = os.path.join(self.container_workdir, filename)

        if self.local_workdir:
            tempdir = None
            local_workdir = self.local_workdir
        else:
            tempdir = tempfile.TemporaryDirectory()
            local_workdir = tempdir.name
```

(continues on next page)

(continued from previous page)

```
        local_file_path = os.path.join(local_workdir, filename)

        try:
            Path(local_file_path).write_text(code)

            return self.execute_command_in_container(f"python {container_file_path}")
        except Exception as e:
            return ErrorArtifact(f"error executing code: {e}")
        finally:
            if tempdir:
                tempdir.cleanup()

    def default_docker_client(self) -> Optional[DockerClient]:
        import docker

        try:
            return docker.from_env()
        except Exception as e:
            logging.exception(e)

            return None

    def image_name(self, tool: BaseTool) -> str:
        import stringcase  # pyright: ignore[reportMissingImports]

        return f"{stringcase.snakecase(tool.name)}_image"

    def container_name(self, tool: BaseTool) -> str:
        import stringcase  # pyright: ignore[reportMissingImports]

        return f"{stringcase.snakecase(tool.name)}_container"

    def remove_existing_container(self, name: str) -> None:
        from docker.errors import NotFound
        from docker.models.containers import Container

        try:
            existing_container = self.docker_client.containers.get(name)
            if isinstance(existing_container, Container):
                existing_container.remove(force=True)

                logging.info("Removed existing container %s", name)
        except NotFound:
            pass

    def build_image(self, tool: BaseTool) -> None:
        with tempfile.TemporaryDirectory() as temp_dir:
            shutil.copy(self.dockerfile_path, temp_dir)
            shutil.copy(self.requirements_txt_path, temp_dir)

            image = self.docker_client.images.build(path=temp_dir, tag=self.image_
```

(continues on next page)

(continued from previous page)

```
↪name(tool), rm=True, forcerm=True)

            if isinstance(image, tuple):
                logging.info("Built image: %s", image[0].short_id)

    def dependencies(self) -> list[str]:
        with open(self.requirements_txt_path) as file:
            return [line.strip() for line in file]

    def __del__(self) -> None:
        if self._tempdir:
            self._tempdir.cleanup()
```

- `_tempdir = field(default=None, kw_only=True)` class-attribute instance-attribute
- `container_workdir = field(default='/griptape', kw_only=True)` class-attribute instance-attribute
- `docker_client = field(default=Factory(lambda self: self.default_docker_client(), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `dockerfile_path = field(default=Factory(lambda self: f'{os.path.join(self.tool_dir(), 'resources/Dockerfile')}', takes_self=True), kw_only=True)` class-attribute instance-attribute
- `env_vars = field(factory=dict, kw_only=True)` class-attribute instance-attribute
- `local_workdir = field(default=None, kw_only=True)` class-attribute instance-attribute
- `requirements_txt_path = field(default=Factory(lambda self: f'{os.path.join(self.tool_dir(), 'resources/requirements.txt')}', takes_self=True), kw_only=True)` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/tools/computer/tool.py

```
def __attrs_post_init__(self) -> None:
    super().__attrs_post_init__()

    if self.local_workdir:
        Path(self.local_workdir).mkdir(parents=True, exist_ok=True)
    else:
        self._tempdir = tempfile.TemporaryDirectory()
        self.local_workdir = self._tempdir.name
```

del()

Source Code in griptape/tools/computer/tool.py

```
def __del__(self) -> None:
    if self._tempdir:
        self._tempdir.cleanup()
```

build_image(tool)

Source Code in griptape/tools/computer/tool.py

```
def build_image(self, tool: BaseTool) -> None:
    with tempfile.TemporaryDirectory() as temp_dir:
        shutil.copy(self.dockerfile_path, temp_dir)
        shutil.copy(self.requirements_txt_path, temp_dir)

        image = self.docker_client.images.build(path=temp_dir, tag=self.image_
↪name(tool), rm=True, forcerm=True)

        if isinstance(image, tuple):
            logging.info("Built image: %s", image[0].short_id)
```

container_name(tool)

Source Code in griptape/tools/computer/tool.py

```
def container_name(self, tool: BaseTool) -> str:
    import stringcase  # pyright: ignore[reportMissingImports]

    return f"{stringcase.snakecase(tool.name)}_container"
```

default_docker_client()

Source Code in griptape/tools/computer/tool.py

```
def default_docker_client(self) -> Optional[DockerClient]:
    import docker

    try:
        return docker.from_env()
    except Exception as e:
        logging.exception(e)

        return None
```

dependencies()

Source Code in griptape/tools/computer/tool.py

```
def dependencies(self) -> list[str]:
    with open(self.requirements_txt_path) as file:
        return [line.strip() for line in file]
```

execute_code(params)

Source Code in griptape/tools/computer/tool.py

```
@activity(
    config={
```

(continues on next page)

(continued from previous page)

```
        "description": "Can be used to execute Python code to solve any programmatic␣
↪tasks and access and analyze"
        " files in the file system. If you need to use code output use `print`␣
↪statements. "
        "You have access to the following external Python libraries: "
        "{{ _self.dependencies() }}",
        "schema": Schema(
            {
                Literal("code", description="Python code to execute"): str,
                Literal(
                    "filename",
                    description="name of the file to put the Python code in before␣
↪executing it",
                ): str,
            },
        ),
    },
)
def execute_code(self, params: dict) -> BaseArtifact:
    code = params["values"]["code"]
    filename = params["values"]["filename"]

    return self.execute_code_in_container(filename, code)
```

execute_code_in_container(filename, code)

Source Code in griptape/tools/computer/tool.py

```
def execute_code_in_container(self, filename: str, code: str) -> BaseArtifact:
    container_file_path = os.path.join(self.container_workdir, filename)

    if self.local_workdir:
        tempdir = None
        local_workdir = self.local_workdir
    else:
        tempdir = tempfile.TemporaryDirectory()
        local_workdir = tempdir.name

    local_file_path = os.path.join(local_workdir, filename)

    try:
        Path(local_file_path).write_text(code)

        return self.execute_command_in_container(f"python {container_file_path}")
    except Exception as e:
        return ErrorArtifact(f"error executing code: {e}")
    finally:
        if tempdir:
            tempdir.cleanup()
```

execute_command(params)

Source Code in griptape/tools/computer/tool.py

```
@activity(
    config={
        "description": "Can be used to execute shell commands in Linux",
        "schema": Schema({Literal("command", description="shell command to execute"):␣
↪str}),
    },
)
def execute_command(self, params: dict) -> BaseArtifact:
    command = params["values"]["command"]

    return self.execute_command_in_container(command)
```

execute_command_in_container(command)

Source Code in griptape/tools/computer/tool.py

```
def execute_command_in_container(self, command: str) -> BaseArtifact:
    from docker.models.containers import Container

    try:
        binds = {self.local_workdir: {"bind": self.container_workdir, "mode": "rw"}}␣
↪if self.local_workdir else None

        container = self.docker_client.containers.run(  # pyright:␣
↪ignore[reportCallIssue]
            self.image_name(self),
            environment=self.env_vars,
            command=command,
            name=self.container_name(self),
            volumes=binds,  # pyright: ignore[reportArgumentType] According to the␣
↪[docs](https://docker-py.readthedocs.io/en/stable/containers.html), the type of␣
↪`volumes` is dict[str, dict[str, str]].
            stdout=True,
            stderr=True,
            detach=True,
        )

        if isinstance(container, Container):
            container.wait()

            stderr = container.logs(stdout=False, stderr=True).decode().strip()
            stdout = container.logs(stdout=True, stderr=False).decode().strip()

            container.stop()
            container.remove()

            if stderr:
                return ErrorArtifact(stderr)
            return TextArtifact(stdout)
```

(continues on next page)

(continued from previous page)

```
            return ErrorArtifact("error running container")
        except Exception as e:
            return ErrorArtifact(f"error executing command: {e}")
```

image_name(tool)

Source Code in griptape/tools/computer/tool.py

```
def image_name(self, tool: BaseTool) -> str:
    import stringcase  # pyright: ignore[reportMissingImports]

    return f"{stringcase.snakecase(tool.name)}_image"
```

install_dependencies(env=None)

Source Code in griptape/tools/computer/tool.py

```
def install_dependencies(self, env: Optional[dict[str, str]] = None) -> None:
    super().install_dependencies(env)

    self.remove_existing_container(self.container_name(self))
    self.build_image(self)
```

remove_existing_container(name)

Source Code in griptape/tools/computer/tool.py

```
def remove_existing_container(self, name: str) -> None:
    from docker.errors import NotFound
    from docker.models.containers import Container

    try:
        existing_container = self.docker_client.containers.get(name)
        if isinstance(existing_container, Container):
            existing_container.remove(force=True)

            logging.info("Removed existing container %s", name)
    except NotFound:
        pass
```

validate*docker_client*(, docker_client)

Source Code in griptape/tools/computer/tool.py

```
@docker_client.validator  # pyright: ignore[reportAttributeAccessIssue]
def validate_docker_client(self, _: Attribute, docker_client: DockerClient) -> None:
    if not docker_client:
        raise ValueError("Docker client cant be initialized: make sure the Docker␣
↪daemon is running")
```

date_time

tool

Bases: *BaseTool* “)

Source Code in griptape/tools/date_time/tool.py

```
@define
class DateTimeTool(BaseTool):
    denylist: list[str] = field(default=Factory(lambda: ["get_relative_datetime"]),␣
↪kw_only=True)

    @activity(config={"description": "Can be used to return current date and time."})
    def get_current_datetime(self) -> BaseArtifact:
        return TextArtifact(str(datetime.now()))

    @activity(
        config={
            "description": "Can be used to return a relative date and time.",
            "schema": schema.Schema(
                {
                    schema.Literal(
                        "relative_date_string",
                        description=Relative date in English compatible with the␣
↪dateparser library. For example, "now EST", "20 minutes ago",
                        "in 2 days", "3 months, 1 week and 1 day ago", or "yesterday␣
↪at 2pm",
                    ): str,
                },
            ),
        },
    )
    def get_relative_datetime(self, params: dict) -> BaseArtifact:
        from dateparser import parse

        try:
            date_string = params["values"]["relative_date_string"]
            relative_datetime = parse(date_string)

            if relative_datetime:
                return TextArtifact(str(relative_datetime))
            return ErrorArtifact("invalid date string")
        except Exception as e:
            return ErrorArtifact(f"error getting current datetime: {e}")

    @activity(
        config={
            "description": "Can be used to add a timedelta to a datetime.",
            "schema": schema.Schema(
                {
                    schema.Literal(
                        "timedelta_kwargs",
                        description=A dictionary of keyword arguments to pass to the␣
```

(continues on next page)

(continued from previous page)

```
↪timedelta function. For example, {"days": -1, "hours": 2},
                    ): dict,
                    schema.Optional(
                        schema.Literal(
                            "iso_datetime",
                            description=Datetime represented as a string in ISO 8601␣
↪format. For example, "2021-01-01T00:00:00". Defaults to the current datetime if not␣
↪provided.,
                        )
                    ): str,
                },
            ),
        },
    )
    def add_timedelta(self, timedelta_kwargs: dict, iso_datetime: Optional[str] =␣
↪None) -> BaseArtifact:
        if iso_datetime is None:
            iso_datetime = datetime.now().isoformat()
        return TextArtifact((datetime.fromisoformat(iso_datetime) +␣
↪timedelta(**timedelta_kwargs)).isoformat())

    @activity(
        config={
            "description": "Can be used to calculate the difference between two␣
↪datetimes. The difference is calculated as end_datetime - start_datetime.",
            "schema": schema.Schema(
                {
                    schema.Literal(
                        "start_datetime",
                        description=Datetime represented as a string in ISO 8601␣
↪format. For example, "2021-01-01T00:00:00",
                    ): str,
                    schema.Literal(
                        "end_datetime",
                        description=Datetime represented as a string in ISO 8601␣
↪format. For example, "2021-01-02T00:00:00",
                    ): str,
                }
            ),
        }
    )
    def get_datetime_diff(self, start_datetime: str, end_datetime: str) ->␣
↪BaseArtifact:
        return TextArtifact(str(datetime.fromisoformat(end_datetime) - datetime.
↪fromisoformat(start_datetime)))
```

- `denylist = field(default=Factory(lambda: ['get_relative_datetime']), kw_only=True)`
  class-attribute instance-attribute

add_timedelta(timedelta_kwargs, iso_datetime=None)

Source Code in griptape/tools/date_time/tool.py

```
@activity(
    config={
        "description": "Can be used to add a timedelta to a datetime.",
        "schema": schema.Schema(
            {
                schema.Literal(
                    "timedelta_kwargs",
                    description=A dictionary of keyword arguments to pass to the␣
↪timedelta function. For example, {"days": -1, "hours": 2},
                ): dict,
                schema.Optional(
                    schema.Literal(
                        "iso_datetime",
                        description=Datetime represented as a string in ISO 8601␣
↪format. For example, "2021-01-01T00:00:00". Defaults to the current datetime if not␣
↪provided.,
                    )
                ): str,
            },
        ),
    },
)
def add_timedelta(self, timedelta_kwargs: dict, iso_datetime: Optional[str] = None) ->
↪ BaseArtifact:
    if iso_datetime is None:
        iso_datetime = datetime.now().isoformat()
    return TextArtifact((datetime.fromisoformat(iso_datetime) + timedelta(**timedelta_
↪kwargs)).isoformat())
```

get_current_datetime()

Source Code in griptape/tools/date_time/tool.py

```
@activity(config={"description": "Can be used to return current date and time."})
def get_current_datetime(self) -> BaseArtifact:
    return TextArtifact(str(datetime.now()))
```

get_datetime_diff(start_datetime, end_datetime)

Source Code in griptape/tools/date_time/tool.py

```
@activity(
    config={
        "description": "Can be used to calculate the difference between two datetimes.
↪ The difference is calculated as end_datetime - start_datetime.",
        "schema": schema.Schema(
            {
                schema.Literal(
                    "start_datetime",
```

(continues on next page)

(continued from previous page)

```
                    description=Datetime represented as a string in ISO 8601 format.␣
↪For example, "2021-01-01T00:00:00",
                ): str,
                schema.Literal(
                    "end_datetime",
                    description=Datetime represented as a string in ISO 8601 format.␣
↪For example, "2021-01-02T00:00:00",
                ): str,
            }
        ),
    }
)
def get_datetime_diff(self, start_datetime: str, end_datetime: str) -> BaseArtifact:
    return TextArtifact(str(datetime.fromisoformat(end_datetime) - datetime.
↪fromisoformat(start_datetime)))
```

get_relative_datetime(params)

Source Code in griptape/tools/date_time/tool.py

```
@activity(
    config={
        "description": "Can be used to return a relative date and time.",
        "schema": schema.Schema(
            {
                schema.Literal(
                    "relative_date_string",
                    description=Relative date in English compatible with the␣
↪dateparser library. For example, "now EST", "20 minutes ago",
                    "in 2 days", "3 months, 1 week and 1 day ago", or "yesterday at␣
↪2pm",
                ): str,
            },
        ),
    },
)
def get_relative_datetime(self, params: dict) -> BaseArtifact:
    from dateparser import parse

    try:
        date_string = params["values"]["relative_date_string"]
        relative_datetime = parse(date_string)

        if relative_datetime:
            return TextArtifact(str(relative_datetime))
        return ErrorArtifact("invalid date string")
    except Exception as e:
        return ErrorArtifact(f"error getting current datetime: {e}")
```

email

tool

Bases: *BaseTool* “)

Attributes

Name	Type	Description
<a href="tool.mdx#grip-tape.tools.email.tool.EmaTool.username" title="username = field(default=None, kw_only=True) class-attribute instance-attribute (grip-tape.tools.email.tool.EmaTool.username)"><code class="language-python">	*Optional[str]*	Username/email address used to send email via the SMTP protocol and retrieve email via the IMAP protocol. Example: bender@futurama.com.
<a href="tool.mdx#grip-tape.tools.email.tool.EmaTool.password" title="password = field(default=None, kw_only=True) class-attribute instance-attribute (grip-tape.tools.email.tool.EmaTool.password)"><code class="language-python">	*Optional[str]*	Password used to send email via the SMTP protocol and retrieve email via the IMAP protocol. If using gmail, this would be an App Password.
<a href="tool.mdx#grip-tape.tools.email.tool.EmaTool.email_max_retrieve_count" title="email_max_retrieve_count = field(default=None, kw_only=True) class-attribute instance-attribute (grip-tape.tools.email.tool.EmaTool.email_max_retrieve_count)"><code class="language-python">	*Optional[int]*	Used to limit the number of messages retrieved during any given activities.

continues on next page

Table 402 – continued from previous page

Name	Type	Description
<a href="tool.mdx#grip-tape.tools.email.tool.EmaTool.smtp_host" title="smtp_host = field(default=None, kw_only=True) class-attribute instance-attribute (grip-tape.tools.email.tool.EmaTool.smtp_host)"><code class="language-python">	*Optional[str]*	Hostname or url of the SMTP server. Example: smtp.gmail.com. Required when using the *send* activity.
<a href="tool.mdx#grip-tape.tools.email.tool.EmaTool.smtp_port" title="smtp_port = field(default=None, kw_only=True) class-attribute instance-attribute (grip-tape.tools.email.tool.EmaTool.smtp_port)"><code class="language-python">	*Optional[int]*	Port of the SMTP server. Example: 465. Required when using the *send* activity.
<a href="tool.mdx#grip-tape.tools.email.tool.EmaTool.smtp_use_ssl" title="smtp_use_ssl = field(default=True, kw_only=True) class-attribute instance-attribute (grip-tape.tools.email.tool.EmaTool.smtp_use_ssl)"><co class="language-python">	*bool*	Whether to use SSL when sending email via the SMTP protocol.

continues on next page

Table 402 – continued from previous page

Name	Type	Description
<a href="tool.mdx#griptape.tools.email.tool.EmailTool.smtp_user" title="smtp_user = field(default=Factory(lambda self: self.username, takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.tools.email.tool.EmailTool.smtp_user)"><code class="language-python">	*Optional[str]*	Username/email address used to send email via the SMTP protocol. Overrides username for SMTP only. Required when using the *send* activity.
<a href="tool.mdx#griptape.tools.email.tool.EmailTool.smtp_password" title="smtp_password = field(default=Factory(lambda self: self.password, takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.tools.email.tool.EmailTool.smtp_password)"><code class="language-python">	*Optional[str]*	Password to send email via the SMTP protocol. Overrides password for SMTP only. Required when using the *send* activity.
<a href="tool.mdx#griptape.tools.email.tool.EmailTool.imap_url" title="imap_url = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.email.tool.EmailTool.imap_url)"><code class="language-python">	*Optional[str]*	Hostname or url of the IMAP server. Example: imap.gmail.com. Required when using the *retrieve* activity.

continues on next page

Table 402 – continued from previous page

Name	Type	Description
<code class="language-python">	*Optional[str]*	Username/email address used to retrieve email via the IMAP protocol. Overrides username for IMAP only. Required when using the *retrieve* activity.
<code class="language-python">	*Optional[str]*	Password to retrieve email via the IMAP protocol. Overrides password for IMAP only. Required when using the *retrieve* activity.
<code class="language-python">	*Optional[dict[str, Optional[str]]]*	Descriptions of mailboxes available for retrieving email via the IMAP protocol. Required when using the *retrieve* activity. Example: {'INBOX': 'default mailbox for incoming email', 'SENT': 'default mailbox for sent email'}

continues on next page

Table 402 – continued from previous page

Name	Type	Description
<co class="language-python">	<code class="language-python">EmailLoader</code>	Instance of *EmailLoader*.

Source Code in griptape/tools/email/tool.py

```
@define
class EmailTool(BaseTool):
    """Tool for working with email.

    Attributes:
        username: Username/email address used to send email via the SMTP protocol and␣
↪retrieve email via the IMAP protocol.
            Example: bender@futurama.com.
        password: Password used to send email via the SMTP protocol and retrieve␣
↪email via the IMAP protocol. If using gmail,
            this would be an App Password.
        email_max_retrieve_count: Used to limit the number of messages retrieved␣
↪during any given activities.
        smtp_host: Hostname or url of the SMTP server. Example: smtp.gmail.com.␣
↪Required when using the `send` activity.
        smtp_port: Port of the SMTP server. Example: 465. Required when using the␣
↪`send` activity.
        smtp_use_ssl: Whether to use SSL when sending email via the SMTP protocol.
        smtp_user: Username/email address used to send email via the SMTP protocol.␣
↪Overrides username for SMTP only. Required when using the `send` activity.
        smtp_password: Password to send email via the SMTP protocol. Overrides␣
↪password for SMTP only. Required when using the `send` activity.
        imap_url: Hostname or url of the IMAP server. Example: imap.gmail.com.␣
↪Required when using the `retrieve` activity.
        imap_user: Username/email address used to retrieve email via the IMAP␣
↪protocol. Overrides username for IMAP only. Required when using the `retrieve`␣
```

(continues on next page)

(continued from previous page)

```
↪activity.
        imap_password: Password to retrieve email via the IMAP protocol.  Overrides␣
↪password for IMAP only. Required when using the `retrieve` activity.
        mailboxes: Descriptions of mailboxes available for retrieving email via the␣
↪IMAP protocol. Required when using the `retrieve` activity.
            Example: {INBOX: default mailbox for incoming email, SENT: default␣
↪mailbox for sent email}
        email_loader: Instance of `EmailLoader`.
    """

    username: Optional[str] = field(default=None, kw_only=True)
    password: Optional[str] = field(default=None, kw_only=True)
    email_max_retrieve_count: Optional[int] = field(default=None, kw_only=True)
    smtp_host: Optional[str] = field(default=None, kw_only=True)
    smtp_port: Optional[int] = field(default=None, kw_only=True)
    smtp_use_ssl: bool = field(default=True, kw_only=True)
    smtp_user: Optional[str] = field(default=Factory(lambda self: self.username,␣
↪takes_self=True), kw_only=True)
    smtp_password: Optional[str] = field(default=Factory(lambda self: self.password,␣
↪takes_self=True), kw_only=True)
    imap_url: Optional[str] = field(default=None, kw_only=True)
    imap_user: Optional[str] = field(default=Factory(lambda self: self.username,␣
↪takes_self=True), kw_only=True)
    imap_password: Optional[str] = field(default=Factory(lambda self: self.password,␣
↪takes_self=True), kw_only=True)
    mailboxes: Optional[dict[str, Optional[str]]] = field(default=None, kw_only=True)
    email_loader: EmailLoader = field(
        default=Factory(
            lambda self: EmailLoader(imap_url=self.imap_url, username=self.imap_user,␣
↪password=self.imap_password),
            takes_self=True,
        ),
        kw_only=True,
    )

    @activity(
        config={
            "description": "Can be used to retrieve emails."
            "{% if _self.mailboxes %} Available mailboxes: {{ _self.mailboxes }}{%␣
↪endif %}",
            "schema": Schema(
                {
                    Literal("label", description="Label to retrieve emails from such␣
↪as INBOX or SENT"): str,
                    schema.Optional(
                        Literal("key", description="Optional key for filtering such␣
↪as FROM or SUBJECT"),
                    ): str,
                    schema.Optional(
                        Literal("search_criteria", description="Optional search␣
↪criteria to filter emails by key"),
                    ): str,
```

(continues on next page)

(continued from previous page)

```
                    schema.Optional(Literal("max_count", description="Optional max_
↪email count")): int,
                },
            ),
        },
    )
    def retrieve(self, params: dict) -> ListArtifact | ErrorArtifact:
        if self.mailboxes is None:
            return ErrorArtifact("mailboxes is required")

        values = params["values"]
        max_count = int(values["max_count"]) if values.get("max_count") is not None_
↪else self.email_max_retrieve_count

        return self.email_loader.load(
            EmailLoader.EmailQuery(
                label=values["label"],
                key=values.get("key"),
                search_criteria=values.get("search_criteria"),
                max_count=max_count,
            ),
        )

    @activity(
        config={
            "description": "Can be used to send emails",
            "schema": Schema(
                {
                    Literal("to", description="Recipients email address"): str,
                    Literal("subject", description="Email subject"): str,
                    Literal("body", description="Email body"): str,
                },
            ),
        },
    )
    def send(self, params: dict) -> InfoArtifact | ErrorArtifact:
        values = params["values"]
        if self.smtp_user is None:
            return ErrorArtifact("smtp_user is required")
        if self.smtp_password is None:
            return ErrorArtifact("smtp_password is required")
        if self.smtp_host is None:
            return ErrorArtifact("smtp_host is required")
        if self.smtp_port is None:
            return ErrorArtifact("smtp_port is required")

        msg = MIMEText(values["body"])
        msg["Subject"] = values["subject"]
        msg["From"] = self.smtp_user
        msg["To"] = values["to"]

        try:
```

(continues on next page)

(continued from previous page)

```
            with self._create_smtp_client(self.smtp_host, self.smtp_port) as client:
                client.login(self.smtp_user, self.smtp_password)
                client.sendmail(msg["From"], [msg["To"]], msg.as_string())
                return InfoArtifact("email was successfully sent")
        except Exception as e:
            logging.exception(e)
            return ErrorArtifact(f"error sending email: {e}")

    def _create_smtp_client(self, smtp_host: str, smtp_port: int) -> smtplib.SMTP |␣
↪smtplib.SMTP_SSL:
        if self.smtp_use_ssl:
            return smtplib.SMTP_SSL(smtp_host, smtp_port)
        return smtplib.SMTP(smtp_host, smtp_port)
```

- `email_loader = field(default=Factory(lambda self: EmailLoader(imap_url=self.imap_url, username=self.imap_user, password=self.imap_password), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `email_max_retrieve_count = field(default=None, kw_only=True)` class-attribute instance-attribute
- `imap_password = field(default=Factory(lambda self: self.password, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `imap_url = field(default=None, kw_only=True)` class-attribute instance-attribute
- `imap_user = field(default=Factory(lambda self: self.username, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `mailboxes = field(default=None, kw_only=True)` class-attribute instance-attribute
- `password = field(default=None, kw_only=True)` class-attribute instance-attribute
- `smtp_host = field(default=None, kw_only=True)` class-attribute instance-attribute
- `smtp_password = field(default=Factory(lambda self: self.password, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `smtp_port = field(default=None, kw_only=True)` class-attribute instance-attribute
- `smtp_use_ssl = field(default=True, kw_only=True)` class-attribute instance-attribute
- `smtp_user = field(default=Factory(lambda self: self.username, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `username = field(default=None, kw_only=True)` class-attribute instance-attribute

_create_smtp_client(smtp_host, smtp_port)

Source Code in griptape/tools/email/tool.py

```
def _create_smtp_client(self, smtp_host: str, smtp_port: int) -> smtplib.SMTP |␣
↪smtplib.SMTP_SSL:
    if self.smtp_use_ssl:
        return smtplib.SMTP_SSL(smtp_host, smtp_port)
    return smtplib.SMTP(smtp_host, smtp_port)
```

retrieve(params)

Source Code in griptape/tools/email/tool.py

```
@activity(
    config={
        "description": "Can be used to retrieve emails."
        "{% if _self.mailboxes %} Available mailboxes: {{ _self.mailboxes }}{% endif
↪%}",
        "schema": Schema(
            {
                Literal("label", description="Label to retrieve emails from such as␣
↪INBOX or SENT"): str,
                schema.Optional(
                    Literal("key", description="Optional key for filtering such as␣
↪FROM or SUBJECT"),
                ): str,
                schema.Optional(
                    Literal("search_criteria", description="Optional search criteria␣
↪to filter emails by key"),
                ): str,
                schema.Optional(Literal("max_count", description="Optional max email␣
↪count")): int,
            },
        ),
    },
)
def retrieve(self, params: dict) -> ListArtifact | ErrorArtifact:
    if self.mailboxes is None:
        return ErrorArtifact("mailboxes is required")

    values = params["values"]
    max_count = int(values["max_count"]) if values.get("max_count") is not None else␣
↪self.email_max_retrieve_count

    return self.email_loader.load(
        EmailLoader.EmailQuery(
            label=values["label"],
            key=values.get("key"),
            search_criteria=values.get("search_criteria"),
            max_count=max_count,
        ),
    )
```

send(params)

Source Code in griptape/tools/email/tool.py

```
@activity(
    config={
        "description": "Can be used to send emails",
        "schema": Schema(
            {
```

(continues on next page)

(continued from previous page)

```
                Literal("to", description="Recipients email address"): str,
                Literal("subject", description="Email subject"): str,
                Literal("body", description="Email body"): str,
            },
        ),
    },
)
def send(self, params: dict) -> InfoArtifact | ErrorArtifact:
    values = params["values"]
    if self.smtp_user is None:
        return ErrorArtifact("smtp_user is required")
    if self.smtp_password is None:
        return ErrorArtifact("smtp_password is required")
    if self.smtp_host is None:
        return ErrorArtifact("smtp_host is required")
    if self.smtp_port is None:
        return ErrorArtifact("smtp_port is required")

    msg = MIMEText(values["body"])
    msg["Subject"] = values["subject"]
    msg["From"] = self.smtp_user
    msg["To"] = values["to"]

    try:
        with self._create_smtp_client(self.smtp_host, self.smtp_port) as client:
            client.login(self.smtp_user, self.smtp_password)
            client.sendmail(msg["From"], [msg["To"]], msg.as_string())
            return InfoArtifact("email was successfully sent")
    except Exception as e:
        logging.exception(e)
        return ErrorArtifact(f"error sending email: {e}")
```

extraction

tool

Bases: *BaseTool* “) , *RuleMixin* ”)

Attributes

Name	Type	Description
<a href="tool.mdx#griptape.tools.extraction.tool.ExtractionTool.extraction_engine" title="extraction_engine = field() class-attribute instance-attribute (griptape.tools.extraction.tool.ExtractionTool.extraction_engine)"><code class="language-python">	<a href="../../engines#griptape.engines.BaseExtractionEngine" title="BaseExtractionEngine (griptape.engines.BaseExtractionEngine)"><code class="language-python">BaseExtractionEngine</cod	*ExtractionEngine*.

Source Code in griptape/tools/extraction/tool.py

```
@define(kw_only=True)
class ExtractionTool(BaseTool, RuleMixin):
    """Tool for using an Extraction Engine.

    Attributes:
        extraction_engine: `ExtractionEngine`.
    """

    extraction_engine: BaseExtractionEngine = field()

    @activity(
        config={
            "description": "Can be used extract structured text from data.",
            "schema": Schema(
                {
                    Literal("data"): Or(
                        str,
                        Schema(
                            {
                                "memory_name": str,
                                "artifact_namespace": str,
                            }
                        ),
                    ),
                }
            ),
        },
    )
    def extract(self, params: dict) -> ListArtifact | InfoArtifact | ErrorArtifact:
        data = params["values"]["data"]

        if isinstance(data, str):
            artifacts = ListArtifact([TextArtifact(data)])
        else:
```

(continues on next page)

(continued from previous page)

```
            memory = self.find_input_memory(data["memory_name"])
            artifact_namespace = data["artifact_namespace"]

            if memory is not None:
                artifacts = memory.load_artifacts(artifact_namespace)
            else:
                return ErrorArtifact("memory not found")

        return self.extraction_engine.extract_artifacts(artifacts)
```

- `extraction_engine = field()` class-attribute instance-attribute

extract(params)

Source Code in griptape/tools/extraction/tool.py

```
@activity(
    config={
        "description": "Can be used extract structured text from data.",
        "schema": Schema(
            {
                Literal("data"): Or(
                    str,
                    Schema(
                        {
                            "memory_name": str,
                            "artifact_namespace": str,
                        }
                    ),
                ),
            }
        ),
    },
)
def extract(self, params: dict) -> ListArtifact | InfoArtifact | ErrorArtifact:
    data = params["values"]["data"]

    if isinstance(data, str):
        artifacts = ListArtifact([TextArtifact(data)])
    else:
        memory = self.find_input_memory(data["memory_name"])
        artifact_namespace = data["artifact_namespace"]

        if memory is not None:
            artifacts = memory.load_artifacts(artifact_namespace)
        else:
            return ErrorArtifact("memory not found")

    return self.extraction_engine.extract_artifacts(artifacts)
```

file_manager

tool

Bases: *BaseTool* “)

Attributes

Name	Type	Description
<a href="tool.mdx#griptape.tools.file_manager.tool.FileManagerTool.file_manager_driver" title="file_manager_driver = field(default=Factory(lambda: LocalFileManagerDriver()), kw_only=True) class-attribute instance-attribute (griptape.tools.file_manager.tool.FileManagerTool.file_manager_driver)"><code class="language-python">	<a href="../../drivers/file_manager#griptape.drivers.file_manager.BaseFileManagerDriver" title="BaseFileManagerDriver (griptape.drivers.file_manager.BaseFileManagerDriver)"><code class="language-python">BaseFileManagerDriver</co	File Manager Driver to use to list, load, and save files.

Source Code in griptape/tools/file_manager/tool.py

```
@define
class FileManagerTool(BaseTool):
    """FileManagerTool is a tool that can be used to list, load, and save files.

    Attributes:
        file_manager_driver: File Manager Driver to use to list, load, and save files.
    """

    file_manager_driver: BaseFileManagerDriver = field(default=Factory(lambda:␣
↪LocalFileManagerDriver()), kw_only=True)

    loaders: dict[str, loaders.BaseLoader] = field(
        default=Factory(
            lambda self: {
                "application/pdf": loaders.PdfLoader(file_manager_driver=self.file_
↪manager_driver),
                "text/csv": loaders.CsvLoader(file_manager_driver=self.file_manager_
↪driver),
                "text": loaders.TextLoader(file_manager_driver=self.file_manager_
↪driver),
                "image": loaders.ImageLoader(file_manager_driver=self.file_manager_
```

(continues on next page)

(continued from previous page)

```
↪driver),
                "application/octet-stream": BlobLoader(file_manager_driver=self.file_
↪manager_driver),
            },
            takes_self=True,
        ),
        kw_only=True,
    )

    @activity(
        config={
            "description": "Can be used to list files on disk",
            "schema": Schema(
                {Literal("path", description="Relative path in the POSIX format. For␣
↪example, foo/bar"): str},
            ),
        },
    )
    def list_files_from_disk(self, params: dict) -> TextArtifact | ErrorArtifact:
        path = params["values"]["path"]
        return self.file_manager_driver.list_files(path)

    @activity(
        config={
            "description": "Can be used to load files from disk",
            "schema": Schema(
                {
                    Literal(
                        "paths",
                        description="Relative paths to files to be loaded in the␣
↪POSIX format. For example, [foo/bar/file.txt]",
                    ): Schema([str]),
                },
            ),
        },
    )
    def load_files_from_disk(self, params: dict) -> ListArtifact | ErrorArtifact:
        paths = params["values"]["paths"]
        artifacts = []

        for path in paths:
            # Fetch the file to try and determine the appropriate loader
            abs_path = os.path.join(self.file_manager_driver.workdir, path)
            mime_type = get_mime_type(abs_path)
            loader = next((loader for key, loader in self.loaders.items() if mime_
↪type.startswith(key)))

            artifact = loader.load(path)
            if isinstance(artifact, ListArtifact):
                artifacts.extend(artifact.value)
            else:
                artifacts.append(artifact)
```

(continues on next page)

(continued from previous page)

```
        return ListArtifact(artifacts)

    @activity(
        config={
            "description": "Can be used to save memory artifacts to disk",
            "schema": Schema(
                {
                    Literal(
                        "dir_name",
                        description="Relative destination path name on disk in the␣
↪POSIX format. For example, foo/bar",
                    ): str,
                    Literal("file_name", description="Destination file name. For␣
↪example, baz.txt"): str,
                    "memory_name": str,
                    "artifact_namespace": str,
                },
            ),
        },
    )
    def save_memory_artifacts_to_disk(self, params: dict) -> ErrorArtifact |␣
↪InfoArtifact:
        dir_name = params["values"]["dir_name"]
        file_name = params["values"]["file_name"]
        memory_name = params["values"]["memory_name"]
        artifact_namespace = params["values"]["artifact_namespace"]

        memory = self.find_input_memory(params["values"]["memory_name"])
        if not memory:
            return ErrorArtifact(f"Failed to save memory artifacts to disk - memory␣
↪named {memory_name} not found")

        list_artifact = memory.load_artifacts(artifact_namespace)

        if len(list_artifact) == 0:
            return ErrorArtifact(
                f"Failed to save memory artifacts to disk - memory named {memory_name}
↪ does not contain any artifacts",
            )

        for artifact in list_artifact.value:
            formatted_file_name = f"{artifact.name}-{file_name}" if len(list_
↪artifact) > 1 else file_name
            try:
                value = artifact.value if isinstance(artifact.value, (str, bytes))␣
↪else artifact.to_text()
                self.file_manager_driver.save_file(os.path.join(dir_name, formatted_
↪file_name), value)
            except FileNotFoundError:
                return ErrorArtifact("Path not found")
            except IsADirectoryError:
                return ErrorArtifact("Path is a directory")
```

(continues on next page)

(continued from previous page)

```
            except NotADirectoryError:
                return ErrorArtifact("Not a directory")
            except Exception as e:
                return ErrorArtifact(f"Failed to load file: {e!s}")

        return InfoArtifact("Successfully saved memory artifacts to disk")

    @activity(
        config={
            "description": "Can be used to save content to a file",
            "schema": Schema(
                {
                    Literal(
                        "path",
                        description="Destination file path on disk in the POSIX␣
↪format. For example, foo/bar/baz.txt",
                    ): str,
                    "content": str,
                },
            ),
        },
    )
    def save_content_to_file(self, params: dict) -> ErrorArtifact | InfoArtifact:
        path = params["values"]["path"]
        content = params["values"]["content"]
        return self.file_manager_driver.save_file(path, content)
```

- `file_manager_driver = field(default=Factory(lambda: LocalFileManagerDriver()), kw_only=True)` class-attribute instance-attribute
- `loaders = field(default=Factory(lambda self: {'application/pdf': loaders.PdfLoader(file_manager_driver=self.file_manager_driver), 'text/csv': loaders.CsvLoader(file_manager_driver=self.file_manager_driver), 'text': loaders.TextLoader(file_manager_driver=self.file_manager_driver), 'image': loaders.ImageLoader(file_manager_driver=self.file_manager_driver), 'application/octet-stream': BlobLoader(file_manager_driver=self.file_manager_driver)}, takes_self=True), kw_only=True)` class-attribute instance-attribute

list_files_from_disk(params)

Source Code in griptape/tools/file_manager/tool.py

```
@activity(
    config={
        "description": "Can be used to list files on disk",
        "schema": Schema(
            {Literal("path", description="Relative path in the POSIX format. For␣
↪example, foo/bar"): str},
        ),
    },
)
def list_files_from_disk(self, params: dict) -> TextArtifact | ErrorArtifact:
    path = params["values"]["path"]
```

(continues on next page)

(continued from previous page)

```
    return self.file_manager_driver.list_files(path)
```

load_files_from_disk(params)

Source Code in griptape/tools/file_manager/tool.py

```
@activity(
    config={
        "description": "Can be used to load files from disk",
        "schema": Schema(
            {
                Literal(
                    "paths",
                    description="Relative paths to files to be loaded in the POSIX␣
↪format. For example, [foo/bar/file.txt]",
                ): Schema([str]),
            },
        ),
    },
)
def load_files_from_disk(self, params: dict) -> ListArtifact | ErrorArtifact:
    paths = params["values"]["paths"]
    artifacts = []

    for path in paths:
        # Fetch the file to try and determine the appropriate loader
        abs_path = os.path.join(self.file_manager_driver.workdir, path)
        mime_type = get_mime_type(abs_path)
        loader = next((loader for key, loader in self.loaders.items() if mime_type.
↪startswith(key)))

        artifact = loader.load(path)
        if isinstance(artifact, ListArtifact):
            artifacts.extend(artifact.value)
        else:
            artifacts.append(artifact)
    return ListArtifact(artifacts)
```

save_content_to_file(params)

Source Code in griptape/tools/file_manager/tool.py

```
@activity(
    config={
        "description": "Can be used to save content to a file",
        "schema": Schema(
            {
                Literal(
                    "path",
                    description="Destination file path on disk in the POSIX format.␣
↪For example, foo/bar/baz.txt",
```

(continues on next page)

(continued from previous page)

```
                ): str,
                "content": str,
            },
        ),
    },
)
def save_content_to_file(self, params: dict) -> ErrorArtifact | InfoArtifact:
    path = params["values"]["path"]
    content = params["values"]["content"]
    return self.file_manager_driver.save_file(path, content)
```

save_memory_artifacts_to_disk(params)

Source Code in griptape/tools/file_manager/tool.py

```
@activity(
    config={
        "description": "Can be used to save memory artifacts to disk",
        "schema": Schema(
            {
                Literal(
                    "dir_name",
                    description="Relative destination path name on disk in the POSIX␣
↪format. For example, foo/bar",
                ): str,
                Literal("file_name", description="Destination file name. For example,␣
↪baz.txt"): str,
                "memory_name": str,
                "artifact_namespace": str,
            },
        ),
    },
)
def save_memory_artifacts_to_disk(self, params: dict) -> ErrorArtifact | InfoArtifact:
    dir_name = params["values"]["dir_name"]
    file_name = params["values"]["file_name"]
    memory_name = params["values"]["memory_name"]
    artifact_namespace = params["values"]["artifact_namespace"]

    memory = self.find_input_memory(params["values"]["memory_name"])
    if not memory:
        return ErrorArtifact(f"Failed to save memory artifacts to disk - memory named
↪{memory_name} not found")

    list_artifact = memory.load_artifacts(artifact_namespace)

    if len(list_artifact) == 0:
        return ErrorArtifact(
            f"Failed to save memory artifacts to disk - memory named {memory_name}␣
↪does not contain any artifacts",
        )
```

(continues on next page)

(continued from previous page)

```
    for artifact in list_artifact.value:
        formatted_file_name = f"{artifact.name}-{file_name}" if len(list_artifact) >␣
↪1 else file_name
        try:
            value = artifact.value if isinstance(artifact.value, (str, bytes)) else␣
↪artifact.to_text()
            self.file_manager_driver.save_file(os.path.join(dir_name, formatted_file_
↪name), value)
        except FileNotFoundError:
            return ErrorArtifact("Path not found")
        except IsADirectoryError:
            return ErrorArtifact("Path is a directory")
        except NotADirectoryError:
            return ErrorArtifact("Not a directory")
        except Exception as e:
            return ErrorArtifact(f"Failed to load file: {e!s}")

    return InfoArtifact("Successfully saved memory artifacts to disk")
```

griptape_cloud_tool

tool

Bases: *BaseGriptapeCloudTool* “)

Attributes

Name	Type	Description
<a href="tool.mdx#griptape.tools.griptape_cloud_tool.tool.GriptapeCloudToolTool.tool_id" title="tool_id = field(kw_only=True) class-attribute instance-attribute (griptape.tools.griptape_cloud_tool.tool.GriptapeCloudToolTool.tool_id)"><code class="language-python">	*str*	The ID of the tool to run.

Source Code in griptape/tools/griptape_cloud_tool/tool.py

```
@define()
class GriptapeCloudToolTool(BaseGriptapeCloudTool):
    """Runs a Gen AI Builder hosted Tool.
```

(continues on next page)

(continued from previous page)

```
    Attributes:
        tool_id: The ID of the tool to run.
    """

    tool_id: str = field(kw_only=True)

    def __attrs_post_init__(self) -> None:
        self._init_activities()

    def _init_activities(self) -> None:
        schema = self._get_schema()
        tool_name, activity_schemas = self._parse_schema(schema)

        if self.name == self.__class__.__name__:
            self.name = tool_name

        for activity_name, (description, activity_schema) in activity_schemas.items():
            activity_handler = self._create_activity_handler(activity_name,␣
↪description, activity_schema)

            setattr(self, activity_name, MethodType(activity_handler, self))

    def _get_schema(self) -> dict:
        response = requests.get(urljoin(self.base_url, f"/api/tools/{self.tool_id}/
↪openapi"), headers=self.headers)

        response.raise_for_status()
        schema = response.json()

        if not isinstance(schema, dict):
            raise RuntimeError(f"Invalid schema for tool {self.tool_id}: {schema}")

        if "error" in schema and "tool_run_id" in schema:
            raise RuntimeError(f"Failed to retrieve schema for tool {self.tool_id}:
↪{schema[error]}")

        return schema

    def _parse_schema(self, schema: dict) -> tuple[str, dict[str, tuple[str,␣
↪Schema]]]:
        """Parses an openapi schema into a dictionary of activity names and their␣
↪respective descriptions + schemas."""
        activities = {}

        name = schema.get("info", {}).get("title")

        for path, path_info in schema.get("paths", {}).items():
            if not path.startswith("/activities"):
                continue
            for method, method_info in path_info.items():
                if "post" in method.lower():
                    activity_name = method_info["operationId"]
```

(continues on next page)

(continued from previous page)

```
                    description = method_info.get("description", "")

                    activity_schema = self.__extract_schema_from_ref(
                        schema,
                        method_info.get("requestBody", {})
                        .get("content", {})
                        .get("application/json", {})
                        .get("schema", {}),
                    )

                    activities[activity_name] = (description, activity_schema)

        return name, activities

    def __extract_schema_from_ref(self, schema: dict, schema_ref: dict) -> Schema:
        """Extracts a schema from a $ref if present, resolving it into native schema␣
↪properties."""
        if "$ref" in schema_ref:
            # Resolve the reference and retrieve the schema data
            ref_path = schema_ref["$ref"].split("/")[-1]
            schema_data = schema["components"]["schemas"].get(ref_path, {})
        else:
            # Use the provided schema directly if no $ref is found
            schema_data = schema_ref

        # Convert the schema_data dictionary into a Schema with its properties
        properties = {}
        for prop, prop_info in schema_data.get("properties", {}).items():
            prop_type = prop_info.get("type", "string")
            prop_description = prop_info.get("description", "")
            schema_prop = Literal(prop, description=prop_description)
            is_optional = prop not in schema_data.get("required", [])

            if is_optional:
                schema_prop = SchemaOptional(schema_prop)

            properties[schema_prop] = self._map_openapi_type_to_python(prop_type,␣
↪prop_info)

        return Schema(properties)

    def _map_openapi_type_to_python(
        self, openapi_type: str, schema_info: Optional[dict] = None
    ) -> type | list[type] | Or:
        """Maps OpenAPI types to native Python types."""
        type_mapping = {"string": str, "integer": int, "boolean": bool, "number":␣
↪float, "object": dict}

        if openapi_type == "array" and schema_info is not None and "items" in schema_
↪info:
            enum = schema_info["items"].get("enum")
            if enum:
```

(continues on next page)

(continued from previous page)

```
                return enum
            items_type = schema_info["items"].get("type", "string")
            return [self._map_openapi_type_to_python(items_type)]  # pyright:␣
↪ignore[reportReturnType]
        if schema_info is not None and schema_info.get("enum"):
            return Or(*schema_info["enum"])
        return type_mapping.get(openapi_type, str)

    def _create_activity_handler(self, activity_name: str, description: str, activity_
↪schema: Schema) -> Callable:
        """Creates an activity handler method for the tool."""

        @activity(config={"name": activity_name, "description": description, "schema
↪": activity_schema})
        def activity_handler(self: GriptapeCloudToolTool, values: dict) -> Any:
            return self._run_activity(activity_name, values)

        return activity_handler

    def _run_activity(self, activity_name: str, params: dict) -> BaseArtifact:
        """Runs an activity on the tool with the provided parameters."""
        url = urljoin(self.base_url, f"/api/tools/{self.tool_id}/activities/{activity_
↪name}")

        response = requests.post(url, json=params, headers=self.headers)

        response.raise_for_status()

        try:
            return BaseArtifact.from_dict(response.json())
        except ValueError:
            return TextArtifact(response.text)
```

- `tool_id = field(kw_only=True)` class-attribute instance-attribute

attrs_post_init()

Source Code in griptape/tools/griptape_cloud_tool/tool.py

```
def __attrs_post_init__(self) -> None:
    self._init_activities()
```

__extract_schema_from_ref(schema, schema_ref)

Source Code in griptape/tools/griptape_cloud_tool/tool.py

```
def __extract_schema_from_ref(self, schema: dict, schema_ref: dict) -> Schema:
    """Extracts a schema from a $ref if present, resolving it into native schema␣
↪properties."""
    if "$ref" in schema_ref:
        # Resolve the reference and retrieve the schema data
        ref_path = schema_ref["$ref"].split("/")[-1]
```

(continues on next page)

(continued from previous page)

```
        schema_data = schema["components"]["schemas"].get(ref_path, {})
    else:
        # Use the provided schema directly if no $ref is found
        schema_data = schema_ref

    # Convert the schema_data dictionary into a Schema with its properties
    properties = {}
    for prop, prop_info in schema_data.get("properties", {}).items():
        prop_type = prop_info.get("type", "string")
        prop_description = prop_info.get("description", "")
        schema_prop = Literal(prop, description=prop_description)
        is_optional = prop not in schema_data.get("required", [])

        if is_optional:
            schema_prop = SchemaOptional(schema_prop)

        properties[schema_prop] = self._map_openapi_type_to_python(prop_type, prop_
↪info)

    return Schema(properties)
```

_create_activity_handler(activity_name, description, activity_schema)

Source Code in griptape/tools/griptape_cloud_tool/tool.py

```
def _create_activity_handler(self, activity_name: str, description: str, activity_
↪schema: Schema) -> Callable:
    """Creates an activity handler method for the tool."""

    @activity(config={"name": activity_name, "description": description, "schema":␣
↪activity_schema})
    def activity_handler(self: GriptapeCloudToolTool, values: dict) -> Any:
        return self._run_activity(activity_name, values)

    return activity_handler
```

_get_schema()

Source Code in griptape/tools/griptape_cloud_tool/tool.py

```
def _get_schema(self) -> dict:
    response = requests.get(urljoin(self.base_url, f"/api/tools/{self.tool_id}/openapi
↪"), headers=self.headers)

    response.raise_for_status()
    schema = response.json()

    if not isinstance(schema, dict):
        raise RuntimeError(f"Invalid schema for tool {self.tool_id}: {schema}")

    if "error" in schema and "tool_run_id" in schema:
```

(continues on next page)

(continued from previous page)

```
        raise RuntimeError(f"Failed to retrieve schema for tool {self.tool_id}:
↪{schema[error]}")

    return schema
```

_init_activities()

Source Code in griptape/tools/griptape_cloud_tool/tool.py

```
def _init_activities(self) -> None:
    schema = self._get_schema()
    tool_name, activity_schemas = self._parse_schema(schema)

    if self.name == self.__class__.__name__:
        self.name = tool_name

    for activity_name, (description, activity_schema) in activity_schemas.items():
        activity_handler = self._create_activity_handler(activity_name, description,␣
↪activity_schema)

        setattr(self, activity_name, MethodType(activity_handler, self))
```

_map_openapi_type_to_python(openapi_type, schema_info=None)

Source Code in griptape/tools/griptape_cloud_tool/tool.py

```
def _map_openapi_type_to_python(
    self, openapi_type: str, schema_info: Optional[dict] = None
) -> type | list[type] | Or:
    """Maps OpenAPI types to native Python types."""
    type_mapping = {"string": str, "integer": int, "boolean": bool, "number": float,
↪"object": dict}

    if openapi_type == "array" and schema_info is not None and "items" in schema_info:
        enum = schema_info["items"].get("enum")
        if enum:
            return enum
        items_type = schema_info["items"].get("type", "string")
        return [self._map_openapi_type_to_python(items_type)]  # pyright:␣
↪ignore[reportReturnType]
    if schema_info is not None and schema_info.get("enum"):
        return Or(*schema_info["enum"])
    return type_mapping.get(openapi_type, str)
```

_parse_schema(schema)

Source Code in griptape/tools/griptape_cloud_tool/tool.py

```
def _parse_schema(self, schema: dict) -> tuple[str, dict[str, tuple[str, Schema]]]:
    """Parses an openapi schema into a dictionary of activity names and their␣
↪respective descriptions + schemas."""
```

(continues on next page)

(continued from previous page)

```
    activities = {}

    name = schema.get("info", {}).get("title")

    for path, path_info in schema.get("paths", {}).items():
        if not path.startswith("/activities"):
            continue
        for method, method_info in path_info.items():
            if "post" in method.lower():
                activity_name = method_info["operationId"]
                description = method_info.get("description", "")

                activity_schema = self.__extract_schema_from_ref(
                    schema,
                    method_info.get("requestBody", {})
                    .get("content", {})
                    .get("application/json", {})
                    .get("schema", {}),
                )

                activities[activity_name] = (description, activity_schema)

    return name, activities
```

_run_activity(activity_name, params)

Source Code in griptape/tools/griptape_cloud_tool/tool.py

```
def _run_activity(self, activity_name: str, params: dict) -> BaseArtifact:
    """Runs an activity on the tool with the provided parameters."""
    url = urljoin(self.base_url, f"/api/tools/{self.tool_id}/activities/{activity_
↪name}")

    response = requests.post(url, json=params, headers=self.headers)

    response.raise_for_status()

    try:
        return BaseArtifact.from_dict(response.json())
    except ValueError:
        return TextArtifact(response.text)
```

image_query

tool

Bases: *BaseTool* “)

Source Code in griptape/tools/image_query/tool.py

```
@define
class ImageQueryTool(BaseTool):
```

(continues on next page)

(continued from previous page)

```
    prompt_driver: BasePromptDriver = field(kw_only=True)
    image_loader: ImageLoader = field(default=Factory(lambda: ImageLoader()), kw_
↪only=True)

    @activity(
        config={
            "description": "This tool can be used to query the contents of images on␣
↪disk.",
            "schema": Schema(
                {
                    Literal(
                        "query",
                        description="A detailed question to be answered using the␣
↪contents of the provided images.",
                    ): str,
                    Literal("image_paths", description="The paths to an image files␣
↪on disk."): Schema([str]),
                },
            ),
        },
    )
    def query_image_from_disk(self, params: dict) -> TextArtifact | ErrorArtifact:
        query = params["values"]["query"]
        image_paths = params["values"]["image_paths"]

        image_artifacts = []
        for image_path in image_paths:
            image_artifacts.append(self.image_loader.load(image_path))

        return cast(
            "TextArtifact",
            self.prompt_driver.run(
                PromptStack.from_artifact(
                    ListArtifact([TextArtifact(query), *image_artifacts]),
                )
            ).to_artifact(),
        )

    @activity(
        config={
            "description": "This tool can be used to query the contents of images in␣
↪memory.",
            "schema": Schema(
                {
                    Literal(
                        "query",
                        description="A detailed question to be answered using the␣
↪contents of the provided images.",
                    ): str,
                    Literal("image_artifacts", description="Image artifact memory␣
↪references."): [
                        {"image_artifact_namespace": str, "image_artifact_name": str},
```

(continues on next page)

(continued from previous page)

```
                    ],
                    "memory_name": str,
                },
            ),
        },
    )
    def query_images_from_memory(self, params: dict[str, Any]) -> TextArtifact |␣
↪ErrorArtifact:
        query = params["values"]["query"]
        image_artifact_references = params["values"]["image_artifacts"]
        memory = self.find_input_memory(params["values"]["memory_name"])

        if memory is None:
            return ErrorArtifact("memory not found")

        image_artifacts = []
        for image_artifact_reference in image_artifact_references:
            try:
                image_artifact = load_artifact_from_memory(
                    memory,
                    image_artifact_reference["image_artifact_namespace"],
                    image_artifact_reference["image_artifact_name"],
                    ImageArtifact,
                )

                image_artifacts.append(cast("ImageArtifact", image_artifact))
            except ValueError:
                # If were unable to parse the artifact as an ImageArtifact, attempt to
                # parse a BlobArtifact and load it as an ImageArtifact.
                blob_artifact = load_artifact_from_memory(
                    memory,
                    image_artifact_reference["image_artifact_namespace"],
                    image_artifact_reference["image_artifact_name"],
                    BlobArtifact,
                )

                image_artifacts.append(self.image_loader.load(blob_artifact.value))
            except Exception as e:
                return ErrorArtifact(str(e))

        return cast(
            "TextArtifact",
            self.prompt_driver.run(
                PromptStack.from_artifact(
                    ListArtifact([TextArtifact(query), *image_artifacts]),
                )
            ).to_artifact(),
        )
```

- `image_loader = field(default=Factory(lambda: ImageLoader()), kw_only=True)` class-attribute instance-attribute
- `prompt_driver = field(kw_only=True)` class-attribute instance-attribute

query_image_from_disk(params)

Source Code in griptape/tools/image_query/tool.py

```
@activity(
    config={
        "description": "This tool can be used to query the contents of images on disk.
↪",
        "schema": Schema(
            {
                Literal(
                    "query",
                    description="A detailed question to be answered using the␣
↪contents of the provided images.",
                ): str,
                Literal("image_paths", description="The paths to an image files on␣
↪disk."): Schema([str]),
            },
        ),
    },
)
def query_image_from_disk(self, params: dict) -> TextArtifact | ErrorArtifact:
    query = params["values"]["query"]
    image_paths = params["values"]["image_paths"]

    image_artifacts = []
    for image_path in image_paths:
        image_artifacts.append(self.image_loader.load(image_path))

    return cast(
        "TextArtifact",
        self.prompt_driver.run(
            PromptStack.from_artifact(
                ListArtifact([TextArtifact(query), *image_artifacts]),
            )
        ).to_artifact(),
    )
```

query_images_from_memory(params)

Source Code in griptape/tools/image_query/tool.py

```
@activity(
    config={
        "description": "This tool can be used to query the contents of images in␣
↪memory.",
        "schema": Schema(
            {
                Literal(
                    "query",
                    description="A detailed question to be answered using the␣
↪contents of the provided images.",
                ): str,
```

(continues on next page)

(continued from previous page)

```
                Literal("image_artifacts", description="Image artifact memory␣
↪references."): [
                    {"image_artifact_namespace": str, "image_artifact_name": str},
                ],
                "memory_name": str,
            },
        ),
    },
)
def query_images_from_memory(self, params: dict[str, Any]) -> TextArtifact |␣
↪ErrorArtifact:
    query = params["values"]["query"]
    image_artifact_references = params["values"]["image_artifacts"]
    memory = self.find_input_memory(params["values"]["memory_name"])

    if memory is None:
        return ErrorArtifact("memory not found")

    image_artifacts = []
    for image_artifact_reference in image_artifact_references:
        try:
            image_artifact = load_artifact_from_memory(
                memory,
                image_artifact_reference["image_artifact_namespace"],
                image_artifact_reference["image_artifact_name"],
                ImageArtifact,
            )

            image_artifacts.append(cast("ImageArtifact", image_artifact))
        except ValueError:
            # If were unable to parse the artifact as an ImageArtifact, attempt to
            # parse a BlobArtifact and load it as an ImageArtifact.
            blob_artifact = load_artifact_from_memory(
                memory,
                image_artifact_reference["image_artifact_namespace"],
                image_artifact_reference["image_artifact_name"],
                BlobArtifact,
            )

            image_artifacts.append(self.image_loader.load(blob_artifact.value))
        except Exception as e:
            return ErrorArtifact(str(e))

    return cast(
        "TextArtifact",
        self.prompt_driver.run(
            PromptStack.from_artifact(
                ListArtifact([TextArtifact(query), *image_artifacts]),
            )
        ).to_artifact(),
    )
```

inpainting_image_generation

tool

Bases: *BaseImageGenerationTool* “)

Attributes

Name	Type	Description
<a href="tool.mdx#griptape.tools.inpainting_image_generation.tool.InpaintingImageGenerationTool.image_generation_driver" title="image_generation_driver = field(kw_only=True) class-attribute instance-attribute (griptape.tools.inpainting_image_generation.tool.InpaintingImageGenerationTool.image_generation_driver)"><code class="language-python">	<a href="../../drivers/image_generation#griptape.drivers.image_generation.BaseImageGenerationDriver" title="BaseImageGenerationDriver (griptape.drivers.image_generation.BaseImageGenerationDriver)"><code class="language-python">BaseImageGenerationDrive	The image generation driver used to generate the image.
<a href="../../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.inpainting_image_generation.tool.InpaintingImageGenerationTool.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk in output_dir.

continues on next page

Table 406 – continued from previous page

Name	Type	Description
<a href="../../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.inpainting_image_generation.tool.InpaintingImageGenerationTool.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk as output_file.

Source Code in griptape/tools/inpainting_image_generation/tool.py

```
@define
class InpaintingImageGenerationTool(BaseImageGenerationTool):
    """A tool that can be used to generate prompted inpaintings of an image.

    Attributes:
        image_generation_driver: The image generation driver used to generate the␣
↪image.
        output_dir: If provided, the generated image will be written to disk in␣
↪output_dir.
        output_file: If provided, the generated image will be written to disk as␣
↪output_file.
    """

    image_generation_driver: BaseImageGenerationDriver = field(kw_only=True)
    image_loader: ImageLoader = field(default=ImageLoader(), kw_only=True)

    @activity(
        config={
            "description": "Modifies an image within a specified mask area using␣
↪image and mask files.",
            "schema": Schema(
                {
                    Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                    Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                    Literal(
                        "image_file",
```

(continues on next page)

(continued from previous page)

```
                        description="The path to an image file to be used as a base␣
↪to generate variations from.",
                    ): str,
                    Literal("mask_file", description="The path to mask image file."):␣
↪str,
                },
            ),
        },
    )
    def image_inpainting_from_file(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
        prompt = params["values"]["prompt"]
        negative_prompt = params["values"]["negative_prompt"]
        image_file = params["values"]["image_file"]
        mask_file = params["values"]["mask_file"]

        input_artifact = self.image_loader.load(image_file)
        mask_artifact = self.image_loader.load(mask_file)

        return self._generate_inpainting(
            prompt, negative_prompt, cast("ImageArtifact", input_artifact), cast(
↪"ImageArtifact", mask_artifact)
        )

    @activity(
        config={
            "description": "Modifies an image within a specified mask area using␣
↪image and mask artifacts in memory.",
            "schema": Schema(
                {
                    Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                    Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                    "memory_name": str,
                    "image_artifact_namespace": str,
                    "image_artifact_name": str,
                    "mask_artifact_namespace": str,
                    "mask_artifact_name": str,
                },
            ),
        },
    )
    def image_inpainting_from_memory(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
        prompt = params["values"]["prompt"]
        negative_prompt = params["values"]["negative_prompt"]
        image_artifact_namespace = params["values"]["image_artifact_namespace"]
        image_artifact_name = params["values"]["image_artifact_name"]
        mask_artifact_namespace = params["values"]["mask_artifact_namespace"]
        mask_artifact_name = params["values"]["mask_artifact_name"]
        memory = self.find_input_memory(params["values"]["memory_name"])
```

(continues on next page)

(continued from previous page)

```
        if memory is None:
            return ErrorArtifact("memory not found")

        try:
            image_artifact = load_artifact_from_memory(
                memory,
                image_artifact_namespace,
                image_artifact_name,
                ImageArtifact,
            )
            mask_artifact = load_artifact_from_memory(
                memory,
                mask_artifact_namespace,
                mask_artifact_name,
                ImageArtifact,
            )
        except ValueError as e:
            return ErrorArtifact(str(e))

        return self._generate_inpainting(
            prompt, negative_prompt, cast("ImageArtifact", image_artifact), cast(
↪"ImageArtifact", mask_artifact)
        )

    def _generate_inpainting(
        self, prompt: str, negative_prompt: str, image_artifact: ImageArtifact, mask_
↪artifact: ImageArtifact
    ) -> ImageArtifact:
        output_artifact = self.image_generation_driver.run_image_inpainting(
            prompts=[prompt], negative_prompts=[negative_prompt], image=image_
↪artifact, mask=mask_artifact
        )

        if self.output_dir or self.output_file:
            self._write_to_file(output_artifact)

        return output_artifact
```

- `image_generation_driver = field(kw_only=True)` class-attribute instance-attribute
- `image_loader = field(default=ImageLoader(), kw_only=True)` class-attribute instance-attribute

_generate_inpainting(prompt, negative_prompt, image_artifact, mask_artifact)

Source Code in griptape/tools/inpainting_image_generation/tool.py

```
def _generate_inpainting(
    self, prompt: str, negative_prompt: str, image_artifact: ImageArtifact, mask_
↪artifact: ImageArtifact
) -> ImageArtifact:
    output_artifact = self.image_generation_driver.run_image_inpainting(
        prompts=[prompt], negative_prompts=[negative_prompt], image=image_artifact,␣
```

(continues on next page)

(continued from previous page)

```
↪mask=mask_artifact
    )

    if self.output_dir or self.output_file:
        self._write_to_file(output_artifact)

    return output_artifact
```

image_inpainting_from_file(params)

Source Code in griptape/tools/inpainting_image_generation/tool.py

```
@activity(
    config={
        "description": "Modifies an image within a specified mask area using image␣
↪and mask files.",
        "schema": Schema(
            {
                Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                Literal(
                    "image_file",
                    description="The path to an image file to be used as a base to␣
↪generate variations from.",
                ): str,
                Literal("mask_file", description="The path to mask image file."): str,
            },
        ),
    },
)
def image_inpainting_from_file(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
    prompt = params["values"]["prompt"]
    negative_prompt = params["values"]["negative_prompt"]
    image_file = params["values"]["image_file"]
    mask_file = params["values"]["mask_file"]

    input_artifact = self.image_loader.load(image_file)
    mask_artifact = self.image_loader.load(mask_file)

    return self._generate_inpainting(
        prompt, negative_prompt, cast("ImageArtifact", input_artifact), cast(
↪"ImageArtifact", mask_artifact)
    )
```

image_inpainting_from_memory(params)

Source Code in griptape/tools/inpainting_image_generation/tool.py

```
@activity(
    config={
        "description": "Modifies an image within a specified mask area using image␣
↪and mask artifacts in memory.",
        "schema": Schema(
            {
                Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                "memory_name": str,
                "image_artifact_namespace": str,
                "image_artifact_name": str,
                "mask_artifact_namespace": str,
                "mask_artifact_name": str,
            },
        ),
    },
)
def image_inpainting_from_memory(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
    prompt = params["values"]["prompt"]
    negative_prompt = params["values"]["negative_prompt"]
    image_artifact_namespace = params["values"]["image_artifact_namespace"]
    image_artifact_name = params["values"]["image_artifact_name"]
    mask_artifact_namespace = params["values"]["mask_artifact_namespace"]
    mask_artifact_name = params["values"]["mask_artifact_name"]
    memory = self.find_input_memory(params["values"]["memory_name"])

    if memory is None:
        return ErrorArtifact("memory not found")

    try:
        image_artifact = load_artifact_from_memory(
            memory,
            image_artifact_namespace,
            image_artifact_name,
            ImageArtifact,
        )
        mask_artifact = load_artifact_from_memory(
            memory,
            mask_artifact_namespace,
            mask_artifact_name,
            ImageArtifact,
        )
    except ValueError as e:
        return ErrorArtifact(str(e))

    return self._generate_inpainting(
        prompt, negative_prompt, cast("ImageArtifact", image_artifact), cast(
```

(continues on next page)

(continued from previous page)

```
↪"ImageArtifact", mask_artifact)
    )
```

outpainting_image_generation

tool

Bases: *BaseImageGenerationTool* “)

Attributes

Name	Type	Description
<a href="tool.mdx#griptape.tools.outpainting_image_generation.tool.OutpaintingImageGenerationTool.image_generation_driver" title="image_generation_driver = field(kw_only=True) class-attribute instance-attribute (griptape.tools.outpainting_image_generation.tool.OutpaintingImageGenerationTool.image_generation_driver)"><code class="language-python">	<a href="../../drivers/image_generation#griptape.drivers.image_generation.BaseImageGenerationDriver" title="BaseImageGenerationDriver (griptape.drivers.image_generation.BaseImageGenerationDriver)"><code class="language-python">BaseImageGenerationDrive	The image generation driver used to generate the image.

continues on next page

Table 407 – continued from previous page

Name	Type	Description
<a href="../../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.outpainting_image_generation.tool.OutpaintingImageGenerationTool.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk in output_dir.
<a href="../../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.outpainting_image_generation.tool.OutpaintingImageGenerationTool.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk as output_file.

Source Code in griptape/tools/outpainting_image_generation/tool.py

```
@define
class OutpaintingImageGenerationTool(BaseImageGenerationTool):
    """A tool that can be used to generate prompted outpaintings of an image.

    Attributes:
        image_generation_driver: The image generation driver used to generate the␣
↪image.
```

(continues on next page)

(continued from previous page)

```
        output_dir: If provided, the generated image will be written to disk in␣
↪output_dir.
        output_file: If provided, the generated image will be written to disk as␣
↪output_file.
    """

    image_generation_driver: BaseImageGenerationDriver = field(kw_only=True)
    image_loader: ImageLoader = field(default=ImageLoader(), kw_only=True)

    @activity(
        config={
            "description": "Modifies an image outside a specified mask area using␣
↪image and mask files.",
            "schema": Schema(
                {
                    Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                    Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                    Literal(
                        "image_file",
                        description="The path to an image file to be used as a base␣
↪to generate variations from.",
                    ): str,
                    Literal("mask_file", description="The path to mask image file."):␣
↪str,
                },
            ),
        },
    )
    def image_outpainting_from_file(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
        prompt = params["values"]["prompt"]
        negative_prompt = params["values"]["negative_prompt"]
        image_file = params["values"]["image_file"]
        mask_file = params["values"]["mask_file"]

        input_artifact = self.image_loader.load(image_file)
        mask_artifact = self.image_loader.load(mask_file)

        return self._generate_outpainting(prompt, negative_prompt, input_artifact,␣
↪mask_artifact)

    @activity(
        config={
            "description": "Modifies an image outside a specified mask area using␣
↪image and mask artifacts in memory.",
            "schema": Schema(
                {
                    Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                    Literal("negative_prompt", description=BaseImageGenerationTool.
```

(continues on next page)

(continued from previous page)

```
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                    "memory_name": str,
                    "image_artifact_namespace": str,
                    "mask_artifact_namespace": str,
                },
            ),
        },
    )
    def image_outpainting_from_memory(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
        prompt = params["values"]["prompt"]
        negative_prompt = params["values"]["negative_prompt"]
        image_artifact_namespace = params["values"]["image_artifact_namespace"]
        image_artifact_name = params["values"]["image_artifact_name"]
        mask_artifact_namespace = params["values"]["mask_artifact_namespace"]
        mask_artifact_name = params["values"]["mask_artifact_name"]
        memory = self.find_input_memory(params["values"]["memory_name"])

        if memory is None:
            return ErrorArtifact("memory not found")

        try:
            image_artifact = load_artifact_from_memory(
                memory,
                image_artifact_namespace,
                image_artifact_name,
                ImageArtifact,
            )
            mask_artifact = load_artifact_from_memory(
                memory,
                mask_artifact_namespace,
                mask_artifact_name,
                ImageArtifact,
            )
        except ValueError as e:
            return ErrorArtifact(str(e))

        return self._generate_outpainting(
            prompt, negative_prompt, cast("ImageArtifact", image_artifact), cast(
↪"ImageArtifact", mask_artifact)
        )

    def _generate_outpainting(
        self, prompt: str, negative_prompt: str, image_artifact: ImageArtifact, mask_
↪artifact: ImageArtifact
    ) -> ImageArtifact | ErrorArtifact:
        output_artifact = self.image_generation_driver.run_image_outpainting(
            prompts=[prompt], negative_prompts=[negative_prompt], image=image_
↪artifact, mask=mask_artifact
        )

        if self.output_dir or self.output_file:
```

(continues on next page)

(continued from previous page)

```
            self._write_to_file(output_artifact)

        return output_artifact
```

- `image_generation_driver = field(kw_only=True)` class-attribute instance-attribute
- `image_loader = field(default=ImageLoader(), kw_only=True)` class-attribute instance-attribute

_generate_outpainting(prompt, negative_prompt, image_artifact, mask_artifact)

Source Code in griptape/tools/outpainting_image_generation/tool.py

```
def _generate_outpainting(
    self, prompt: str, negative_prompt: str, image_artifact: ImageArtifact, mask_
↪artifact: ImageArtifact
) -> ImageArtifact | ErrorArtifact:
    output_artifact = self.image_generation_driver.run_image_outpainting(
        prompts=[prompt], negative_prompts=[negative_prompt], image=image_artifact,␣
↪mask=mask_artifact
    )

    if self.output_dir or self.output_file:
        self._write_to_file(output_artifact)

    return output_artifact
```

image_outpainting_from_file(params)

Source Code in griptape/tools/outpainting_image_generation/tool.py

```
@activity(
    config={
        "description": "Modifies an image outside a specified mask area using image␣
↪and mask files.",
        "schema": Schema(
            {
                Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                Literal(
                    "image_file",
                    description="The path to an image file to be used as a base to␣
↪generate variations from.",
                ): str,
                Literal("mask_file", description="The path to mask image file."): str,
            },
        ),
    },
)
def image_outpainting_from_file(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
```

(continues on next page)

(continued from previous page)

```
    prompt = params["values"]["prompt"]
    negative_prompt = params["values"]["negative_prompt"]
    image_file = params["values"]["image_file"]
    mask_file = params["values"]["mask_file"]

    input_artifact = self.image_loader.load(image_file)
    mask_artifact = self.image_loader.load(mask_file)

    return self._generate_outpainting(prompt, negative_prompt, input_artifact, mask_
↪artifact)
```

image_outpainting_from_memory(params)

Source Code in griptape/tools/outpainting_image_generation/tool.py

```
@activity(
    config={
        "description": "Modifies an image outside a specified mask area using image␣
↪and mask artifacts in memory.",
        "schema": Schema(
            {
                Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                "memory_name": str,
                "image_artifact_namespace": str,
                "mask_artifact_namespace": str,
            },
        ),
    },
)
def image_outpainting_from_memory(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
    prompt = params["values"]["prompt"]
    negative_prompt = params["values"]["negative_prompt"]
    image_artifact_namespace = params["values"]["image_artifact_namespace"]
    image_artifact_name = params["values"]["image_artifact_name"]
    mask_artifact_namespace = params["values"]["mask_artifact_namespace"]
    mask_artifact_name = params["values"]["mask_artifact_name"]
    memory = self.find_input_memory(params["values"]["memory_name"])

    if memory is None:
        return ErrorArtifact("memory not found")

    try:
        image_artifact = load_artifact_from_memory(
            memory,
            image_artifact_namespace,
            image_artifact_name,
            ImageArtifact,
        )
```

(continues on next page)

(continued from previous page)

```
        mask_artifact = load_artifact_from_memory(
            memory,
            mask_artifact_namespace,
            mask_artifact_name,
            ImageArtifact,
        )
    except ValueError as e:
        return ErrorArtifact(str(e))

    return self._generate_outpainting(
        prompt, negative_prompt, cast("ImageArtifact", image_artifact), cast(
↪"ImageArtifact", mask_artifact)
    )
```

prompt_image_generation

tool

Bases: *BaseImageGenerationTool* “)

Attributes

Name	Type	Description
<a href="tool.mdx#griptape.tools.prompt_image_generation.tool.PromptImageGenerationTool.image_generation_driver" title="image_generation_driver = field(kw_only=True) class-attribute instance-attribute (griptape.tools.prompt_image_generation.tool.PromptImageGenerationTool.image_generation_driver)"><code class="language-python">	<a href="../../drivers/image_generation#griptape.drivers.image_generation.BaseImageGenerationDriver" title="BaseImageGenerationDriver (griptape.drivers.image_generation.BaseImageGenerationDriver)"><code class="language-python">BaseImageGenerationDrive	The image generation driver used to generate the image.

continues on next page

Table 408 – continued from previous page

Name	Type	Description
<a href="../../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.prompt_image_generation.tool.PromptImageGenerationTool.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk in output_dir.
<a href="../../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.prompt_image_generation.tool.PromptImageGenerationTool.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk as output_file.

Source Code in griptape/tools/prompt_image_generation/tool.py

```
@define
class PromptImageGenerationTool(BaseImageGenerationTool):
    """A tool that can be used to generate an image from a text prompt.

    Attributes:
        image_generation_driver: The image generation driver used to generate the␣
↪image.
        output_dir: If provided, the generated image will be written to disk in␣
↪output_dir.
```

(continues on next page)

(continued from previous page)

```
        output_file: If provided, the generated image will be written to disk as␣
↪output_file.
    """

    image_generation_driver: BaseImageGenerationDriver = field(kw_only=True)

    @activity(
        config={
            "description": "Generates an image from text prompts.",
            "schema": Schema(
                {
                    Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                    Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                }
            ),
        },
    )
    def generate_image(self, params: dict[str, dict[str, str]]) -> ImageArtifact |␣
↪ErrorArtifact:
        prompt = params["values"]["prompt"]
        negative_prompt = params["values"]["negative_prompt"]

        output_artifact = self.image_generation_driver.run_text_to_image(
            prompts=[prompt], negative_prompts=[negative_prompt]
        )

        if self.output_dir or self.output_file:
            self._write_to_file(output_artifact)

        return output_artifact
```

- `image_generation_driver = field(kw_only=True)` class-attribute instance-attribute

generate_image(params)

Source Code in griptape/tools/prompt_image_generation/tool.py

```
@activity(
    config={
        "description": "Generates an image from text prompts.",
        "schema": Schema(
            {
                Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
            }
        ),
    },
)
```

(continues on next page)

(continued from previous page)

```
def generate_image(self, params: dict[str, dict[str, str]]) -> ImageArtifact |␣
↪ErrorArtifact:
    prompt = params["values"]["prompt"]
    negative_prompt = params["values"]["negative_prompt"]

    output_artifact = self.image_generation_driver.run_text_to_image(
        prompts=[prompt], negative_prompts=[negative_prompt]
    )

    if self.output_dir or self.output_file:
        self._write_to_file(output_artifact)

    return output_artifact
```

prompt_summary

tool

Bases: *BaseTool* “) , *RuleMixin* ”)

Attributes

Name	Type	Description
<a href="tool.mdx#griptape.tools.prompt_summary.tool.PromptSummaryTool.prompt_summary_engine" title="prompt_summary_engine = field(kw_only=True, default=Factory(lambda: PromptSummaryEngine())) class-attribute instance-attribute (griptape.tools.prompt_summary.tool.PromptSummaryTool.prompt_summary_engine)"><code class="language-python">	<a href="../../engines#griptape.engines.PromptSummaryEngine" title="PromptSummaryEngine (griptape.engines.PromptSummaryEngine)"><code class="language-python">PromptSummaryEngine</co	*PromptSummaryEngine*.

Source Code in griptape/tools/prompt_summary/tool.py

```
@define(kw_only=True)
class PromptSummaryTool(BaseTool, RuleMixin):
    """Tool for using a Prompt Summary Engine.

    Attributes:
```

(continues on next page)

(continued from previous page)

```
        prompt_summary_engine: `PromptSummaryEngine`.
    """

    prompt_summary_engine: PromptSummaryEngine = field(kw_only=True,␣
↪default=Factory(lambda: PromptSummaryEngine()))

    @activity(
        config={
            "description": "Can be used to summarize text content.",
            "schema": Schema(
                {
                    Literal("summary"): Or(
                        str,
                        Schema(
                            {
                                "memory_name": str,
                                "artifact_namespace": str,
                            }
                        ),
                    ),
                }
            ),
        },
    )
    def summarize(self, params: dict) -> BaseArtifact:
        summary = params["values"]["summary"]

        if isinstance(summary, str):
            artifacts = ListArtifact([TextArtifact(summary)])
        else:
            memory = self.find_input_memory(summary["memory_name"])
            artifact_namespace = summary["artifact_namespace"]

            if memory is not None:
                artifacts = memory.load_artifacts(artifact_namespace)
            else:
                return ErrorArtifact("memory not found")

        return self.prompt_summary_engine.summarize_artifacts(artifacts,␣
↪rulesets=self.rulesets)
```

- `prompt_summary_engine = field(kw_only=True, default=Factory(lambda: PromptSummaryEngine()))` class-attribute instance-attribute

summarize(params)

Source Code in griptape/tools/prompt_summary/tool.py

```
@activity(
    config={
        "description": "Can be used to summarize text content.",
        "schema": Schema(
```

(continues on next page)

(continued from previous page)

```
            {
                Literal("summary"): Or(
                    str,
                    Schema(
                        {
                            "memory_name": str,
                            "artifact_namespace": str,
                        }
                    ),
                ),
            }
        ),
    },
)
def summarize(self, params: dict) -> BaseArtifact:
    summary = params["values"]["summary"]

    if isinstance(summary, str):
        artifacts = ListArtifact([TextArtifact(summary)])
    else:
        memory = self.find_input_memory(summary["memory_name"])
        artifact_namespace = summary["artifact_namespace"]

        if memory is not None:
            artifacts = memory.load_artifacts(artifact_namespace)
        else:
            return ErrorArtifact("memory not found")

    return self.prompt_summary_engine.summarize_artifacts(artifacts, rulesets=self.
↪rulesets)
```

query

tool

Bases: *BaseTool* “) , *RuleMixin* ”)

Source Code in griptape/tools/query/tool.py

```
@define(kw_only=True)
class QueryTool(BaseTool, RuleMixin):
    """Tool for performing a query against data."""

    prompt_driver: BasePromptDriver = field(default=Factory(lambda: Defaults.drivers_
↪config.prompt_driver))

    _rag_engine: RagEngine = field(
        default=Factory(
            lambda self: RagEngine(
                response_stage=ResponseRagStage(
                    response_modules=[
                        PromptResponseRagModule(prompt_driver=self.prompt_driver,␣
```

(continues on next page)

(continued from previous page)

```
↪rulesets=self.rulesets)
                    ],
                ),
            ),
            takes_self=True,
        ),
        alias="_rag_engine",
    )

    @activity(
        config={
            "description": "Can be used to search through textual content.",
            "schema": Schema(
                {
                    Literal("query", description="A natural language search query"):␣
↪str,
                    Literal("content"): Or(
                        str,
                        Schema(
                            {
                                "memory_name": str,
                                "artifact_namespace": str,
                            }
                        ),
                    ),
                }
            ),
        },
    )
    def query(self, params: dict) -> ListArtifact | ErrorArtifact:
        query = params["values"]["query"]
        content = params["values"]["content"]

        if isinstance(content, str):
            text_artifacts = [TextArtifact(content)]
        else:
            memory = self.find_input_memory(content["memory_name"])
            artifact_namespace = content["artifact_namespace"]

            if memory is not None:
                artifacts = memory.load_artifacts(artifact_namespace)
            else:
                return ErrorArtifact("memory not found")

            text_artifacts = [artifact for artifact in artifacts if␣
↪isinstance(artifact, TextArtifact)]

        outputs = self._rag_engine.process(RagContext(query=query, text_chunks=text_
↪artifacts)).outputs

        if len(outputs) > 0:
            return ListArtifact(outputs)
```

(continues on next page)

(continued from previous page)

```
        return ErrorArtifact("query output is empty")
```

- `_rag_engine = field(default=Factory(lambda self: RagEngine(response_stage=ResponseRagStage(respo prompt_driver, rulesets=self.rulesets)])), takes_self=True), alias='_rag_engine')` class-attribute instance-attribute
- `prompt_driver = field(default=Factory(lambda: Defaults.drivers_config. prompt_driver))` class-attribute instance-attribute

query(params)

Source Code in griptape/tools/query/tool.py

```
@activity(
    config={
        "description": "Can be used to search through textual content.",
        "schema": Schema(
            {
                Literal("query", description="A natural language search query"): str,
                Literal("content"): Or(
                    str,
                    Schema(
                        {
                            "memory_name": str,
                            "artifact_namespace": str,
                        }
                    ),
                ),
            }
        ),
    },
)
def query(self, params: dict) -> ListArtifact | ErrorArtifact:
    query = params["values"]["query"]
    content = params["values"]["content"]

    if isinstance(content, str):
        text_artifacts = [TextArtifact(content)]
    else:
        memory = self.find_input_memory(content["memory_name"])
        artifact_namespace = content["artifact_namespace"]

        if memory is not None:
            artifacts = memory.load_artifacts(artifact_namespace)
        else:
            return ErrorArtifact("memory not found")

        text_artifacts = [artifact for artifact in artifacts if isinstance(artifact,␣
↪TextArtifact)]

    outputs = self._rag_engine.process(RagContext(query=query, text_chunks=text_
↪artifacts)).outputs
```

(continues on next page)

(continued from previous page)

```
    if len(outputs) > 0:
        return ListArtifact(outputs)
    return ErrorArtifact("query output is empty")
```

rag

tool

Bases: *BaseTool* “)

Attributes

Name	Type	Description
<code class="language-python">	*str*	LLM-friendly RAG engine description.
<code class="language-python">	<code class="language-python">RagEngine</code>	*RagEngine*.

Source Code in griptape/tools/rag/tool.py

```
@define(kw_only=True)
class RagTool(BaseTool):
    """Tool for querying a RAG engine.

    Attributes:
        description: LLM-friendly RAG engine description.
        rag_engine: `RagEngine`.
    """

    description: str = field()
    rag_engine: RagEngine = field()
```

(continues on next page)

(continued from previous page)

```
    @activity(
        config={
            "description": "{{ _self.description }}",
            "schema": Schema({Literal("query", description="A natural language search query"): str}),
        },
    )
    def search(self, params: dict) -> ListArtifact | ErrorArtifact:
        query = params["values"]["query"]

        try:
            artifacts = self.rag_engine.process_query(query).outputs

            outputs = []
            for artifact in artifacts:
                if isinstance(artifact, ListArtifact):
                    outputs.extend(artifact.value)
                else:
                    outputs.append(artifact)

            if len(outputs) > 0:
                return ListArtifact(outputs)
            return ErrorArtifact("query output is empty")

        except Exception as e:
            return ErrorArtifact(f"error querying: {e}")
```

- `description = field()` class-attribute instance-attribute
- `rag_engine = field()` class-attribute instance-attribute

search(params)

Source Code in griptape/tools/rag/tool.py

```
@activity(
    config={
        "description": "{{ _self.description }}",
        "schema": Schema({Literal("query", description="A natural language search query"): str}),
    },
)
def search(self, params: dict) -> ListArtifact | ErrorArtifact:
    query = params["values"]["query"]

    try:
        artifacts = self.rag_engine.process_query(query).outputs

        outputs = []
        for artifact in artifacts:
            if isinstance(artifact, ListArtifact):
                outputs.extend(artifact.value)
            else:
```

(continues on next page)

(continued from previous page)

```
                outputs.append(artifact)

        if len(outputs) > 0:
            return ListArtifact(outputs)
        return ErrorArtifact("query output is empty")

    except Exception as e:
        return ErrorArtifact(f"error querying: {e}")
```

rest_api

tool

Bases: *BaseTool* “)

Attributes

Name	Type	Description
<a href="tool.mdx#griptape.tools.rest_api.tool.RestApiTool.base_url" title="base_url = field(kw_only=True) class-attribute instance-attribute (griptape.tools.rest_api.tool.RestApiTool.base_url)"><code class="language-python">	*str*	The base url that will be used for the request.
<a href="tool.mdx#griptape.tools.rest_api.tool.RestApiTool.path" title="path = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.rest_api.tool.RestApiTool.path)"><code class="language-python">	*Optional[str]*	The resource path that will be appended to base_url.

continues on next page

Table 411 – continued from previous page

Name	Type	Description
<a href="tool.mdx#grip-tape.tools.rest_api.tool.Re Tool.description" title="description = field(kw_only=True) class-attribute instance-attribute (grip-tape.tools.rest_api.tool.Re Tool.descrip-tion)"><code class="language-python">	*str*	A description of what the REST API does.
<a href="tool.mdx#grip-tape.tools.rest_api.tool.Re Tool.re-quest_body_schema" title="re-quest_body_schema = field(default=None, kw_only=True) class-attribute instance-attribute (grip-tape.tools.rest_api.tool.Re Tool.re-quest_body_schema)"><c class="language-python">	*Optional[str]*	A JSON schema string describing the request body. Recommended for PUT, POST, and PATCH requests.
<a href="tool.mdx#grip-tape.tools.rest_api.tool.Re Tool.re-quest_query_params_sch title="re-quest_query_params_sch = field(default=None, kw_only=True) class-attribute instance-attribute (grip-tape.tools.rest_api.tool.Re Tool.re-quest_query_params_sch class="language-python">	*Optional[str]*	A JSON schema string describing the available query parameters.

continues on next page

Table 411 – continued from previous page

Name	Type	Description
<a href="tool.mdx#griptape.tools.rest_api.tool.RestApiTool.request_path_params_schema" title="request_path_params_schema = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.rest_api.tool.RestApiTool.request_path_params_schema)"><code class="language-python">	*Optional[str]*	A JSON schema string describing the available path parameters. The schema must describe an array of string values.
<a href="tool.mdx#griptape.tools.rest_api.tool.RestApiTool.response_body_schema" title="response_body_schema = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.rest_api.tool.RestApiTool.response_body_schema)"><code class="language-python">	*Optional[str]*	A JSON schema string describing the response body.
<a href="tool.mdx#griptape.tools.rest_api.tool.RestApiTool.request_headers" title="request_headers = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.rest_api.tool.RestApiTool.request_headers)"><code class="language-python">	*Optional[dict[str, str]]*	Headers to include in the requests.

Source Code in griptape/tools/rest_api/tool.py

```
@define
class RestApiTool(BaseTool):
    """A tool for making REST API requests.
```

(continues on next page)

(continued from previous page)

```
    Attributes:
        base_url: The base url that will be used for the request.
        path: The resource path that will be appended to base_url.
        description: A description of what the REST API does.
        request_body_schema: A JSON schema string describing the request body.␣
↪Recommended for PUT, POST, and PATCH requests.
        request_query_params_schema: A JSON schema string describing the available␣
↪query parameters.
        request_path_params_schema: A JSON schema string describing the available␣
↪path parameters. The schema must describe an array of string values.
        response_body_schema: A JSON schema string describing the response body.
        request_headers: Headers to include in the requests.
    """

    base_url: str = field(kw_only=True)
    path: Optional[str] = field(default=None, kw_only=True)
    description: str = field(kw_only=True)
    request_path_params_schema: Optional[str] = field(default=None, kw_only=True)
    request_query_params_schema: Optional[str] = field(default=None, kw_only=True)
    request_body_schema: Optional[str] = field(default=None, kw_only=True)
    response_body_schema: Optional[str] = field(default=None, kw_only=True)
    request_headers: Optional[dict[str, str]] = field(default=None, kw_only=True)

    @property
    def full_url(self) -> str:
        return self._build_url(self.base_url, path=self.path)

    @activity(
        config={
            "description": dedent(
                """
                This tool can be used to make a put request to the rest api url: {{ _
↪self.full_url }}
                This rest api has the following description: {{ _self.description }}
                {% if _self.request_body_schema %}The request body must follow this␣
↪JSON schema: {{ _self.request_body_schema }}{% endif %}
                {% if _self.response_body_schema %}The response body must follow this␣
↪JSON schema: {{ _self.response_body_schema }}{% endif %}
                """,
            ),
            "schema": Schema({Literal("body", description="The request body."): dict}
↪),
        },
    )
    def put(self, params: dict) -> BaseArtifact:
        from requests import exceptions, put

        values = params["values"]
        base_url = self.base_url
        path = self.path
        body = values["body"]
        url = self._build_url(base_url, path=path)
```

(continues on next page)

(continued from previous page)

```
        try:
            response = put(url, json=body, timeout=30, headers=self.request_headers)

            return TextArtifact(response.text)
        except exceptions.RequestException as err:
            return ErrorArtifact(str(err))

    @activity(
        config={
            "description": dedent(
                """
                This tool can be used to make a patch request to the rest api url: {{ _self.full_url }}
                This rest api has the following description: {{ _self.description }}
                {% if _self.request_path_parameters %}The request path parameters must follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
                {% if _self.request_body_schema %}The request body must follow this JSON schema: {{ _self.request_body_schema }}{% endif %}
                {% if _self.response_body_schema %}The response body must follow this JSON schema: {{ _self.response_body_schema }}{% endif %}
                """,
            ),
            "schema": Schema(
                {
                    Literal("path_params", description="The request path parameters."): Schema([str]),
                    Literal("body", description="The request body."): dict,
                },
            ),
        },
    )
    def patch(self, params: dict) -> BaseArtifact:
        from requests import exceptions, patch

        values = params["values"]
        base_url = self.base_url
        path = self.path
        body = values["body"]
        path_params = values["path_params"]
        url = self._build_url(base_url, path=path, path_params=path_params)

        try:
            response = patch(url, json=body, timeout=30, headers=self.request_headers)
            return TextArtifact(response.text)
        except exceptions.RequestException as err:
            return ErrorArtifact(str(err))

    @activity(
        config={
            "description": dedent(
                """
```

(continues on next page)

(continued from previous page)

```
                This tool can be used to make a post request to the rest api url: {{ _
↪self.full_url }}
                This rest api has the following description: {{ _self.description }}
                {% if _self.request_body_schema %}The request body must follow this␣
↪JSON schema: {{ _self.request_body_schema }}{% endif %}
                {% if _self.response_body_schema %}The response body must follow this␣
↪JSON schema: {{ _self.response_body_schema }}{% endif %}
                """,
            ),
            "schema": Schema({Literal("body", description="The request body."): dict}
↪),
        },
    )
    def post(self, params: dict) -> BaseArtifact:
        from requests import exceptions, post

        values = params["values"]
        base_url = self.base_url
        path = self.path
        url = self._build_url(base_url, path=path)
        body = values["body"]

        try:
            response = post(url, json=body, timeout=30, headers=self.request_headers)
            return TextArtifact(response.text)
        except exceptions.RequestException as err:
            return ErrorArtifact(str(err))

    @activity(
        config={
            "description": dedent(
                """
                This tool can be used to make a get request to the rest api url: {{ _
↪self.full_url }}
                This rest api has the following description: {{ _self.description }}
                {% if _self.request_path_parameters %}The request path parameters␣
↪must follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
                {% if _self.request_query_parameters %}The request query parameters␣
↪must follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
                {% if _self.response_body_schema %}The response body must follow this␣
↪JSON schema: {{ _self.response_body_schema }}{% endif %}
                """,
            ),
            "schema": schema.Optional(
                Schema(
                    {
                        schema.Optional(Literal("query_params", description="The␣
↪request query parameters.")): dict,
                        schema.Optional(Literal("path_params", description="The␣
↪request path parameters.")): Schema(
                            [str]
                        ),
```

(continues on next page)

(continued from previous page)

```
                    },
                ),
            ),
        },
    )
    def get(self, params: dict) -> BaseArtifact:
        from requests import exceptions, get

        values = params["values"]
        base_url = self.base_url
        path = self.path

        query_params = {}
        path_params = []
        if values:
            query_params = values.get("query_params", {})
            path_params = values.get("path_params", [])
        url = self._build_url(base_url, path=path, path_params=path_params)

        try:
            response = get(url, params=query_params, timeout=30, headers=self.request_
↪headers)
            return TextArtifact(response.text)
        except exceptions.RequestException as err:
            return ErrorArtifact(str(err))

    @activity(
        config={
            "description": dedent(
                """
                This tool can be used to make a delete request to the rest api url: {
↪{ _self.full_url }}
                This rest api has the following description: {{ _self.description }}
                {% if _self.request_path_parameters %}The request path parameters␣
↪must follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
                {% if _self.request_query_parameters %}The request query parameters␣
↪must follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
                """,
            ),
            "schema": Schema(
                {
                    schema.Optional(Literal("query_params", description="The request␣
↪query parameters.")): dict,
                    schema.Optional(Literal("path_params", description="The request␣
↪path parameters.")): Schema([str]),
                },
            ),
        },
    )
    def delete(self, params: dict) -> BaseArtifact:
        from requests import delete, exceptions
```

(continues on next page)

(continued from previous page)

```
        values = params["values"]
        base_url = self.base_url
        path = self.path

        query_params = values.get("query_params", {})
        path_params = values.get("path_params", [])
        url = self._build_url(base_url, path=path, path_params=path_params)

        try:
            response = delete(url, params=query_params, timeout=30, headers=self.
↪request_headers)
            return TextArtifact(response.text)
        except exceptions.RequestException as err:
            return ErrorArtifact(str(err))

    def _build_url(self, base_url: str, path: Optional[str] = None, path_params:␣
↪Optional[list] = None) -> str:
        url = ""

        if path:
            url += path.strip("/")

        if path_params:
            url += f"/{str.join(/, map(str, path_params))}"

        return urljoin(base_url.strip("/"), url)
```

- `base_url = field(kw_only=True)` class-attribute instance-attribute
- `description = field(kw_only=True)` class-attribute instance-attribute
- `full_url` property
- `path = field(default=None, kw_only=True)` class-attribute instance-attribute
- `request_body_schema = field(default=None, kw_only=True)` class-attribute instance-attribute
- `request_headers = field(default=None, kw_only=True)` class-attribute instance-attribute
- `request_path_params_schema = field(default=None, kw_only=True)` class-attribute instance-attribute
- `request_query_params_schema = field(default=None, kw_only=True)` class-attribute instance-attribute
- `response_body_schema = field(default=None, kw_only=True)` class-attribute instance-attribute

_build_url(base_url, path=None, path_params=None)

Source Code in griptape/tools/rest_api/tool.py

```
def _build_url(self, base_url: str, path: Optional[str] = None, path_params:␣
↪Optional[list] = None) -> str:
    url = ""

    if path:
```

(continues on next page)

(continued from previous page)

```
        url += path.strip("/")

    if path_params:
        url += f"/{str.join(/, map(str, path_params))}"

    return urljoin(base_url.strip("/"), url)
```

delete(params)

Source Code in griptape/tools/rest_api/tool.py

```
@activity(
    config={
        "description": dedent(
            """
            This tool can be used to make a delete request to the rest api url: {{ _
↪self.full_url }}
            This rest api has the following description: {{ _self.description }}
            {% if _self.request_path_parameters %}The request path parameters must␣
↪follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
            {% if _self.request_query_parameters %}The request query parameters must␣
↪follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
            """,
        ),
        "schema": Schema(
            {
                schema.Optional(Literal("query_params", description="The request␣
↪query parameters.")): dict,
                schema.Optional(Literal("path_params", description="The request path␣
↪parameters.")): Schema([str]),
            },
        ),
    },
)
def delete(self, params: dict) -> BaseArtifact:
    from requests import delete, exceptions

    values = params["values"]
    base_url = self.base_url
    path = self.path

    query_params = values.get("query_params", {})
    path_params = values.get("path_params", [])
    url = self._build_url(base_url, path=path, path_params=path_params)

    try:
        response = delete(url, params=query_params, timeout=30, headers=self.request_
↪headers)
        return TextArtifact(response.text)
    except exceptions.RequestException as err:
        return ErrorArtifact(str(err))
```

get(params)

Source Code in griptape/tools/rest_api/tool.py

```
@activity(
    config={
        "description": dedent(
            """
            This tool can be used to make a get request to the rest api url: {{ _self.
↪full_url }}
            This rest api has the following description: {{ _self.description }}
            {% if _self.request_path_parameters %}The request path parameters must␣
↪follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
            {% if _self.request_query_parameters %}The request query parameters must␣
↪follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
            {% if _self.response_body_schema %}The response body must follow this␣
↪JSON schema: {{ _self.response_body_schema }}{% endif %}
            """,
        ),
        "schema": schema.Optional(
            Schema(
                {
                    schema.Optional(Literal("query_params", description="The request␣
↪query parameters.")): dict,
                    schema.Optional(Literal("path_params", description="The request␣
↪path parameters.")): Schema(
                        [str]
                    ),
                },
            ),
        ),
    },
)
def get(self, params: dict) -> BaseArtifact:
    from requests import exceptions, get

    values = params["values"]
    base_url = self.base_url
    path = self.path

    query_params = {}
    path_params = []
    if values:
        query_params = values.get("query_params", {})
        path_params = values.get("path_params", [])
    url = self._build_url(base_url, path=path, path_params=path_params)

    try:
        response = get(url, params=query_params, timeout=30, headers=self.request_
↪headers)
        return TextArtifact(response.text)
    except exceptions.RequestException as err:
        return ErrorArtifact(str(err))
```

patch(params)

Source Code in griptape/tools/rest_api/tool.py

```
@activity(
    config={
        "description": dedent(
            """
            This tool can be used to make a patch request to the rest api url: {{ _
↪self.full_url }}
            This rest api has the following description: {{ _self.description }}
            {% if _self.request_path_parameters %}The request path parameters must␣
↪follow this JSON schema: {{ _self.request_path_params_schema }}{% endif %}
            {% if _self.request_body_schema %}The request body must follow this JSON␣
↪schema: {{ _self.request_body_schema }}{% endif %}
            {% if _self.response_body_schema %}The response body must follow this␣
↪JSON schema: {{ _self.response_body_schema }}{% endif %}
            """,
        ),
        "schema": Schema(
            {
                Literal("path_params", description="The request path parameters."):␣
↪Schema([str]),
                Literal("body", description="The request body."): dict,
            },
        ),
    },
)
def patch(self, params: dict) -> BaseArtifact:
    from requests import exceptions, patch

    values = params["values"]
    base_url = self.base_url
    path = self.path
    body = values["body"]
    path_params = values["path_params"]
    url = self._build_url(base_url, path=path, path_params=path_params)

    try:
        response = patch(url, json=body, timeout=30, headers=self.request_headers)
        return TextArtifact(response.text)
    except exceptions.RequestException as err:
        return ErrorArtifact(str(err))
```

post(params)

Source Code in griptape/tools/rest_api/tool.py

```
@activity(
    config={
        "description": dedent(
            """
            This tool can be used to make a post request to the rest api url: {{ _
```

(continues on next page)

(continued from previous page)

```
↪self.full_url }}
            This rest api has the following description: {{ _self.description }}
            {% if _self.request_body_schema %}The request body must follow this JSON␣
↪schema: {{ _self.request_body_schema }}{% endif %}
            {% if _self.response_body_schema %}The response body must follow this␣
↪JSON schema: {{ _self.response_body_schema }}{% endif %}
            """,
        ),
        "schema": Schema({Literal("body", description="The request body."): dict}),
    },
)
def post(self, params: dict) -> BaseArtifact:
    from requests import exceptions, post

    values = params["values"]
    base_url = self.base_url
    path = self.path
    url = self._build_url(base_url, path=path)
    body = values["body"]

    try:
        response = post(url, json=body, timeout=30, headers=self.request_headers)
        return TextArtifact(response.text)
    except exceptions.RequestException as err:
        return ErrorArtifact(str(err))
```

put(params)

Source Code in griptape/tools/rest_api/tool.py

```
@activity(
    config={
        "description": dedent(
            """
            This tool can be used to make a put request to the rest api url: {{ _self.
↪full_url }}
            This rest api has the following description: {{ _self.description }}
            {% if _self.request_body_schema %}The request body must follow this JSON␣
↪schema: {{ _self.request_body_schema }}{% endif %}
            {% if _self.response_body_schema %}The response body must follow this␣
↪JSON schema: {{ _self.response_body_schema }}{% endif %}
            """,
        ),
        "schema": Schema({Literal("body", description="The request body."): dict}),
    },
)
def put(self, params: dict) -> BaseArtifact:
    from requests import exceptions, put

    values = params["values"]
    base_url = self.base_url
    path = self.path
```

(continues on next page)

(continued from previous page)

```
    body = values["body"]
    url = self._build_url(base_url, path=path)

    try:
        response = put(url, json=body, timeout=30, headers=self.request_headers)

        return TextArtifact(response.text)
    except exceptions.RequestException as err:
        return ErrorArtifact(str(err))
```

sql

tool

Bases: *BaseTool* “)

Source Code in griptape/tools/sql/tool.py

```
@define
class SqlTool(BaseTool):
    sql_loader: SqlLoader = field(kw_only=True)
    schema_name: Optional[str] = field(default=None, kw_only=True)
    table_name: str = field(kw_only=True)
    table_description: Optional[str] = field(default=None, kw_only=True)
    engine_name: Optional[str] = field(default=None, kw_only=True)

    @property
    def full_table_name(self) -> str:
        return f"{self.schema_name}.{self.table_name}" if self.schema_name else self.
↪table_name

    @property
    def table_schema(self) -> Optional[str]:
        return self.sql_loader.sql_driver.get_table_schema(self.full_table_name,␣
↪schema=self.schema_name)

    @activity(
        config={
            "description": "Can be used to execute{% if _self.engine_name %} {{ _self.
↪engine_name }}{% endif %} SQL SELECT queries "
            "in table {{ _self.full_table_name }}. "
            "Make sure the `SELECT` statement contains enough columns to get an␣
↪answer without knowing "
            "the original question. "
            "Be creative when you use `WHERE` statements: you can use wildcards,␣
↪`LOWER()`, and other functions "
            "to get better results. "
            "You can use JOINs if more tables are available in other tools.\n"
            "{{ _self.table_name }} schema: {{ _self.table_schema }}\n"
            "{% if _self.table_description %}{{ _self.table_name }} description: {{ _
↪self.table_description }}{% endif %}",
            "schema": Schema({"sql_query": str}),
```

(continues on next page)

(continued from previous page)

```
        },
    )
    def execute_query(self, params: dict) -> ListArtifact | InfoArtifact |␣
↪ErrorArtifact:
        try:
            query = params["values"]["sql_query"]
            rows = self.sql_loader.load(query)
        except Exception as e:
            return ErrorArtifact(f"error executing query: {e}")

        if len(rows) > 0:
            return rows
        return InfoArtifact("No results found")
```

- `engine_name = field(default=None, kw_only=True)` class-attribute instance-attribute
- `full_table_name` property
- `schema_name = field(default=None, kw_only=True)` class-attribute instance-attribute
- `sql_loader = field(kw_only=True)` class-attribute instance-attribute
- `table_description = field(default=None, kw_only=True)` class-attribute instance-attribute
- `table_name = field(kw_only=True)` class-attribute instance-attribute
- `table_schema` property

execute_query(params)

Source Code in griptape/tools/sql/tool.py

```
@activity(
    config={
        "description": "Can be used to execute{% if _self.engine_name %} {{ _self.
↪engine_name }}{% endif %} SQL SELECT queries "
        "in table {{ _self.full_table_name }}. "
        "Make sure the `SELECT` statement contains enough columns to get an answer␣
↪without knowing "
        "the original question. "
        "Be creative when you use `WHERE` statements: you can use wildcards,␣
↪`LOWER()`, and other functions "
        "to get better results. "
        "You can use JOINs if more tables are available in other tools.\n"
        "{{ _self.table_name }} schema: {{ _self.table_schema }}\n"
        "{% if _self.table_description %}{{ _self.table_name }} description: {{ _self.
↪table_description }}{% endif %}",
        "schema": Schema({"sql_query": str}),
    },
)
def execute_query(self, params: dict) -> ListArtifact | InfoArtifact | ErrorArtifact:
    try:
        query = params["values"]["sql_query"]
        rows = self.sql_loader.load(query)
    except Exception as e:
```

(continues on next page)

(continued from previous page)

```
        return ErrorArtifact(f"error executing query: {e}")

    if len(rows) > 0:
        return rows
    return InfoArtifact("No results found")
```

structure_run

tool

Bases: *BaseTool* “)

Attributes

Name	Type	Description
<a href="tool.mdx#griptape.tools.structure_run.tool.StructureRunTool.description" title="description = field(kw_only=True, metadata={'serializable': True})">class-attribute instance-attribute (griptape.tools.structure_run.tool.StructureRunTool.description)"><code class="language-python">	*str*	A description of what the Structure does.
<a href="tool.mdx#griptape.tools.structure_run.tool.StructureRunTool.structure_run_driver" title="structure_run_driver = field(kw_only=True, metadata={'serializable': True})">class-attribute instance-attribute (griptape.tools.structure_run.tool.StructureRunTool.structure_run_driver)"><code class="language-python">	<a href="../../drivers/structure_run#griptape.drivers.structure_run.BaseStructureRunDriver" title="BaseStructureRunDriver (griptape.drivers.structure_run.BaseStructureRunDriver)"><code class="language-python">BaseStructureRunDriver</c	Driver to run the Structure.

Source Code in griptape/tools/structure_run/tool.py

```
@define
class StructureRunTool(BaseTool):
    """Tool for running a Structure.

    Attributes:
        description: A description of what the Structure does.
        structure_run_driver: Driver to run the Structure.
    """

    description: str = field(kw_only=True, metadata={"serializable": True})
    structure_run_driver: BaseStructureRunDriver = field(kw_only=True, metadata={
↪"serializable": True})

    @activity(
        config={
            "description": "Can be used to run a Gen AI Builder Structure with the␣
↪following description: {{ _self.description }}",
            "schema": Schema(
                {
                    Literal("args", description="A list of string arguments to submit␣
↪to the Structure Run"): Schema(
                        [str]
                    )
                },
            ),
        },
    )
    def run_structure(self, params: dict) -> BaseArtifact:
        args: list[str] = params["values"]["args"]

        return self.structure_run_driver.run(*[TextArtifact(arg) for arg in args])
```

- `description = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute
- `structure_run_driver = field(kw_only=True, metadata={'serializable': True})` class-attribute instance-attribute

run_structure(params)

Source Code in griptape/tools/structure_run/tool.py

```
@activity(
    config={
        "description": "Can be used to run a Gen AI Builder Structure with the␣
↪following description: {{ _self.description }}",
        "schema": Schema(
            {
                Literal("args", description="A list of string arguments to submit to␣
↪the Structure Run"): Schema(
```

(continues on next page)

(continued from previous page)

```
                    [str]
                )
            },
        ),
    },
)
def run_structure(self, params: dict) -> BaseArtifact:
    args: list[str] = params["values"]["args"]

    return self.structure_run_driver.run(*[TextArtifact(arg) for arg in args])
```

structured_output

tool

Bases: *BaseTool* “)

Source Code in griptape/tools/structured_output/tool.py

```
@define
class StructuredOutputTool(BaseTool):
    output_schema: Union[Schema, type[BaseModel]] = field(kw_only=True)

    @activity(
        config={
            "description": "Used to provide the final response which ends this␣
↪conversation.",
            "schema": lambda _self: _self.output_schema,
        }
    )
    def provide_output(self, params: dict) -> BaseArtifact:
        return JsonArtifact(params["values"])
```

- `output_schema = field(kw_only=True)` class-attribute instance-attribute

provide_output(params)

Source Code in griptape/tools/structured_output/tool.py

```
@activity(
    config={
        "description": "Used to provide the final response which ends this␣
↪conversation.",
        "schema": lambda _self: _self.output_schema,
    }
)
def provide_output(self, params: dict) -> BaseArtifact:
    return JsonArtifact(params["values"])
```

text_to_speech

tool

Bases: *ArtifactFileOutputMixin* “) , *BaseTool* ”)

Attributes

Name	Type	Description
<a href="tool.mdx#griptape.tools.text_to_speech.ToSpeechTool.text_to_speech_driv ti- tle="text_to_speech_drive = field(kw_only=True) class-attribute instance-attribute (griptape.tools.text_to_speech.ToSpeechTool.text_to_speech_driv class="language-python">	<a href="../../drivers/text_to_speech#griptape.drivers.text_to_speech.BaseTextToSpeechDriver" title="BaseTextToSpeechDriver (griptape.drivers.text_to_speech.BaseTextToSpeechDriver)"><code class="language-python">BaseTextToSpeechDriver</c	The text to audio generation driver used to generate the speech audio.
<a href="../../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.text_to_speech.ToSpeechTool.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated audio will be written to disk in output_dir.

continues on next page

Table 413 – continued from previous page

Name	Type	Description
<a href="../../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.text_to_speech.ToSpeechTool.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated audio will be written to disk as output_file.

Source Code in griptape/tools/text_to_speech/tool.py

```
@define
class TextToSpeechTool(ArtifactFileOutputMixin, BaseTool):
    """A tool that can be used to generate speech from input text.

    Attributes:
        text_to_speech_driver: The text to audio generation driver used to generate the speech audio.
        output_dir: If provided, the generated audio will be written to disk in output_dir.
        output_file: If provided, the generated audio will be written to disk as output_file.
    """

    text_to_speech_driver: BaseTextToSpeechDriver = field(kw_only=True)

    @activity(
        config={
            "description": "Can be used to generate speech from the provided input text.",
            "schema": Schema({Literal("text", description="The literal text to be converted to speech."): str}),
        },
    )
    def text_to_speech(self, params: dict[str, Any]) -> AudioArtifact | ErrorArtifact:
        text = params["values"]["text"]

        output_artifact = self.text_to_speech_driver.run_text_to_audio(prompts=[text])

        if self.output_dir or self.output_file:
            self._write_to_file(output_artifact)
```

(continues on next page)

(continued from previous page)

```
        return output_artifact
```

- `text_to_speech_driver = field(kw_only=True)` class-attribute instance-attribute

text_to_speech(params)

Source Code in griptape/tools/text_to_speech/tool.py

```
@activity(
    config={
        "description": "Can be used to generate speech from the provided input text.",
        "schema": Schema({Literal("text", description="The literal text to be␣
↪converted to speech."): str}),
    },
)
def text_to_speech(self, params: dict[str, Any]) -> AudioArtifact | ErrorArtifact:
    text = params["values"]["text"]

    output_artifact = self.text_to_speech_driver.run_text_to_audio(prompts=[text])

    if self.output_dir or self.output_file:
        self._write_to_file(output_artifact)

    return output_artifact
```

variation_image_generation

tool

Bases: *BaseImageGenerationTool* “)

Attributes

Name	Type	Description
<a href="tool.mdx#griptape.tools.variation_image_generation.tool.VariationImageGenerationTool.image_generation_driver" title="image_generation_driver = field(kw_only=True) class-attribute instance-attribute (griptape.tools.variation_image_generation.tool.VariationImageGenerationTool.image_generation_driver)"><code class="language-python">	<a href="../../drivers/image_generation#griptape.drivers.image_generation.BaseImageGenerationDriver" title="BaseImageGenerationDriver (griptape.drivers.image_generation.BaseImageGenerationDriver)"><code class="language-python">BaseImageGenerationDrive	The image generation driver used to generate the image.
<a href="../../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_dir" title="output_dir = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.variation_image_generation.tool.VariationImageGenerationTool.output_dir)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk in output_dir.

continues on next page

Table 414 – continued from previous page

Name	Type	Description
<a href="../../mixins/artifact_file_output_mixin.mdx#griptape.mixins.artifact_file_output_mixin.ArtifactFileOutputMixin.output_file" title="output_file = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.tools.variation_image_generation.tool.VariationImageGenerationTool.output_file)"><code class="language-python">	*Optional[str]*	If provided, the generated image will be written to disk as output_file.

Source Code in griptape/tools/variation_image_generation/tool.py

```
@define
class VariationImageGenerationTool(BaseImageGenerationTool):
    """A tool that can be used to generate prompted variations of an image.

    Attributes:
        image_generation_driver: The image generation driver used to generate the␣
↪image.
        output_dir: If provided, the generated image will be written to disk in␣
↪output_dir.
        output_file: If provided, the generated image will be written to disk as␣
↪output_file.
    """

    image_generation_driver: BaseImageGenerationDriver = field(kw_only=True)
    image_loader: ImageLoader = field(default=ImageLoader(), kw_only=True)

    @activity(
        config={
            "description": "Generates a variation of a given input image file.",
            "schema": Schema(
                {
                    Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                    Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                    Literal(
                        "image_file",
                        description="The path to an image file to be used as a base␣
```

(continues on next page)

(continued from previous page)

```
↪to generate variations from.",
                    ): str,
                },
            ),
        },
    )
    def image_variation_from_file(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
        prompt = params["values"]["prompt"]
        negative_prompt = params["values"]["negative_prompt"]
        image_file = params["values"]["image_file"]

        image_artifact = self.image_loader.load(image_file)

        return self._generate_variation(prompt, negative_prompt, image_artifact)

    @activity(
        config={
            "description": "Generates a variation of a given input image artifact in␣
↪memory.",
            "schema": Schema(
                {
                    Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                    Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                    "memory_name": str,
                    "artifact_namespace": str,
                    "artifact_name": str,
                },
            ),
        },
    )
    def image_variation_from_memory(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
        prompt = params["values"]["prompt"]
        negative_prompt = params["values"]["negative_prompt"]
        artifact_namespace = params["values"]["artifact_namespace"]
        artifact_name = params["values"]["artifact_name"]
        memory = self.find_input_memory(params["values"]["memory_name"])

        if memory is None:
            return ErrorArtifact("memory not found")

        try:
            image_artifact = load_artifact_from_memory(memory, artifact_namespace,␣
↪artifact_name, ImageArtifact)
        except ValueError as e:
            return ErrorArtifact(str(e))

        return self._generate_variation(prompt, negative_prompt, cast("ImageArtifact",
↪ image_artifact))
```

(continues on next page)

(continued from previous page)

```
    def _generate_variation(
        self, prompt: str, negative_prompt: str, image_artifact: ImageArtifact
    ) -> ImageArtifact | ErrorArtifact:
        output_artifact = self.image_generation_driver.run_image_variation(
            prompts=[prompt], negative_prompts=[negative_prompt], image=image_artifact
        )

        if self.output_dir or self.output_file:
            self._write_to_file(output_artifact)

        return output_artifact
```

- `image_generation_driver = field(kw_only=True)` class-attribute instance-attribute
- `image_loader = field(default=ImageLoader(), kw_only=True)` class-attribute instance-attribute

_generate_variation(prompt, negative_prompt, image_artifact)

Source Code in griptape/tools/variation_image_generation/tool.py

```
def _generate_variation(
    self, prompt: str, negative_prompt: str, image_artifact: ImageArtifact
) -> ImageArtifact | ErrorArtifact:
    output_artifact = self.image_generation_driver.run_image_variation(
        prompts=[prompt], negative_prompts=[negative_prompt], image=image_artifact
    )

    if self.output_dir or self.output_file:
        self._write_to_file(output_artifact)

    return output_artifact
```

image_variation_from_file(params)

Source Code in griptape/tools/variation_image_generation/tool.py

```
@activity(
    config={
        "description": "Generates a variation of a given input image file.",
        "schema": Schema(
            {
                Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                Literal(
                    "image_file",
                    description="The path to an image file to be used as a base to␣
↪generate variations from.",
                ): str,
            },
```

(continues on next page)

(continued from previous page)

```
        ),
    },
)
def image_variation_from_file(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
    prompt = params["values"]["prompt"]
    negative_prompt = params["values"]["negative_prompt"]
    image_file = params["values"]["image_file"]

    image_artifact = self.image_loader.load(image_file)

    return self._generate_variation(prompt, negative_prompt, image_artifact)
```

image_variation_from_memory(params)

Source Code in griptape/tools/variation_image_generation/tool.py

```
@activity(
    config={
        "description": "Generates a variation of a given input image artifact in␣
↪memory.",
        "schema": Schema(
            {
                Literal("prompt", description=BaseImageGenerationTool.PROMPT_
↪DESCRIPTION): str,
                Literal("negative_prompt", description=BaseImageGenerationTool.
↪NEGATIVE_PROMPT_DESCRIPTION): str,
                "memory_name": str,
                "artifact_namespace": str,
                "artifact_name": str,
            },
        ),
    },
)
def image_variation_from_memory(self, params: dict[str, dict[str, str]]) ->␣
↪ImageArtifact | ErrorArtifact:
    prompt = params["values"]["prompt"]
    negative_prompt = params["values"]["negative_prompt"]
    artifact_namespace = params["values"]["artifact_namespace"]
    artifact_name = params["values"]["artifact_name"]
    memory = self.find_input_memory(params["values"]["memory_name"])

    if memory is None:
        return ErrorArtifact("memory not found")

    try:
        image_artifact = load_artifact_from_memory(memory, artifact_namespace,␣
↪artifact_name, ImageArtifact)
    except ValueError as e:
        return ErrorArtifact(str(e))

    return self._generate_variation(prompt, negative_prompt, cast("ImageArtifact",␣
```

(continues on next page)

(continued from previous page)

```
↪image_artifact))
```

vector_store

tool

Bases: *BaseTool* “)

Attributes

Name	Type	Description
<code class="language-python">	*str*	LLM-friendly vector DB description.
<code class="language-python">	<code class="language-python">BaseVectorStoreDriver</coc	*BaseVectorStoreDriver*.

continues on next page

Table 415 – continued from previous page

Name	Type	Description
<c class="language-python">	*dict[str, Any]*	Optional dictionary of vector store driver query parameters.
<code class="language-python">	<code class="language-python">Callable[[list[Entry]], BaseArtifact]</code>	Optional lambda for processing vector store driver query output `Entry`s.

Source Code in griptape/tools/vector_store/tool.py

```
@define(kw_only=True)
class VectorStoreTool(BaseTool):
    """A tool for querying a vector database.

    Attributes:
        description: LLM-friendly vector DB description.
        vector_store_driver: `BaseVectorStoreDriver`.
        query_params: Optional dictionary of vector store driver query parameters.
        process_query_output: Optional lambda for processing vector store driver␣
↪query output `Entry`s.
    """
```

(continues on next page)

(continued from previous page)

```
    DEFAULT_TOP_N = 5

    description: str = field()
    vector_store_driver: BaseVectorStoreDriver = field()
    query_params: dict[str, Any] = field(factory=dict)
    process_query_output: Callable[[list[BaseVectorStoreDriver.Entry]], BaseArtifact]␣
↪= field(
        default=Factory(lambda: lambda es: ListArtifact([e.to_artifact() for e in␣
↪es])),
    )

    @activity(
        config={
            "description": "Can be used to search a database with the following␣
↪description: {{ _self.description }}",
            "schema": Schema(
                {
                    Literal(
                        "query",
                        description="A natural language search query to run against␣
↪the vector database",
                    ): str,
                },
            ),
        },
    )
    def search(self, params: dict) -> BaseArtifact:
        query = params["values"]["query"]

        try:
            return self.process_query_output(self.vector_store_driver.query(query,␣
↪**self.query_params))
        except Exception as e:
            return ErrorArtifact(f"error querying vector store: {e}")
```

- `DEFAULT_TOP_N = 5` class-attribute instance-attribute
- `description = field()` class-attribute instance-attribute
- `process_query_output = field(default=Factory(lambda: lambda es: ListArtifact([e.to_artifact() for e in es])))` class-attribute instance-attribute
- `query_params = field(factory=dict)` class-attribute instance-attribute
- `vector_store_driver = field()` class-attribute instance-attribute

search(params)

Source Code in griptape/tools/vector_store/tool.py

```
@activity(
    config={
        "description": "Can be used to search a database with the following␣
↪description: {{ _self.description }}",
```

(continues on next page)

(continued from previous page)

```
        "schema": Schema(
            {
                Literal(
                    "query",
                    description="A natural language search query to run against the␣
↪vector database",
                ): str,
            },
        ),
    },
)
def search(self, params: dict) -> BaseArtifact:
    query = params["values"]["query"]

    try:
        return self.process_query_output(self.vector_store_driver.query(query, **self.
↪query_params))
    except Exception as e:
        return ErrorArtifact(f"error querying vector store: {e}")
```

web_scraper

tool

Bases: *BaseTool* “)

Source Code in griptape/tools/web_scraper/tool.py

```
@define
class WebScraperTool(BaseTool):
    web_loader: WebLoader = field(default=Factory(lambda: WebLoader()), kw_only=True)
    text_chunker: TextChunker = field(default=Factory(lambda: TextChunker(max_
↪tokens=400)), kw_only=True)

    @activity(
        config={
            "description": "Can be used to browse a web page and load its content",
            "schema": Schema({Literal("url", description="Valid HTTP URL"): str}),
        },
    )
    def get_content(self, params: dict) -> ListArtifact | ErrorArtifact:
        url = params["values"]["url"]

        try:
            result = self.web_loader.load(url)
            chunks = self.text_chunker.chunk(result)

            return ListArtifact(chunks)
        except Exception as e:
            return ErrorArtifact("Error getting page content: " + str(e))
```

- `text_chunker = field(default=Factory(lambda: TextChunker(max_tokens=400)), kw_only=True)` class-attribute instance-attribute

- `web_loader = field(default=Factory(lambda: WebLoader()), kw_only=True)` class-attribute instance-attribute

get_content(params)

Source Code in griptape/tools/web_scraper/tool.py

```
@activity(
    config={
        "description": "Can be used to browse a web page and load its content",
        "schema": Schema({Literal("url", description="Valid HTTP URL"): str}),
    },
)
def get_content(self, params: dict) -> ListArtifact | ErrorArtifact:
    url = params["values"]["url"]

    try:
        result = self.web_loader.load(url)
        chunks = self.text_chunker.chunk(result)

        return ListArtifact(chunks)
    except Exception as e:
        return ErrorArtifact("Error getting page content: " + str(e))
```

web_search

tool

Bases: *BaseTool* “)

Source Code in griptape/tools/web_search/tool.py

```
@define
class WebSearchTool(BaseTool):
    web_search_driver: BaseWebSearchDriver = field(kw_only=True)

    @activity(
        config={
            "description": "Can be used for searching the web via the {{ _self.web_
↪search_driver.__class__.__name__}}.",
            "schema": Schema(
                {
                    Literal(
                        "query",
                        description="Search engine request that returns a list of␣
↪pages with titles, descriptions, and URLs",
                    ): str,
                },
            ),
        },
    )
    def search(self, values: dict) -> ListArtifact | ErrorArtifact:
        # `BaseWebSearchDriver.query` already has a parameter named `query`, so we␣
↪need to pop it from the values
```

(continues on next page)

(continued from previous page)

```
        # to avoid passing it twice.
        query = values.pop("query")

        try:
            return self.web_search_driver.search(query, **values)
        except Exception as e:
            return ErrorArtifact(f"Error searching {query} with {self.web_search_
↪driver.__class__.__name__}: {e}")
```

- `web_search_driver = field(kw_only=True)` class-attribute instance-attribute

search(values)

Source Code in griptape/tools/web_search/tool.py

```
@activity(
    config={
        "description": "Can be used for searching the web via the {{ _self.web_search_
↪driver.__class__.__name__}}.",
        "schema": Schema(
            {
                Literal(
                    "query",
                    description="Search engine request that returns a list of pages␣
↪with titles, descriptions, and URLs",
                ): str,
            },
        ),
    },
)
def search(self, values: dict) -> ListArtifact | ErrorArtifact:
    # `BaseWebSearchDriver.query` already has a parameter named `query`, so we need␣
↪to pop it from the values
    # to avoid passing it twice.
    query = values.pop("query")

    try:
        return self.web_search_driver.search(query, **values)
    except Exception as e:
        return ErrorArtifact(f"Error searching {query} with {self.web_search_driver.__
↪class__.__name__}: {e}")
```

11.5.21 utils

- `__all__ = ['J2', 'Chat', 'CommandRunner', 'Conversation', 'GriptapeCloudStructure', 'ManifestValidator', 'PythonRunner', 'Stream', 'StructureVisualizer', 'TokenCounter', 'add_key_in_dict_recursively', 'build_strict_schema', 'deprecation_warn', 'dict_merge', 'execute_futures_dict', 'execute_futures_list', 'execute_futures_list_dict', 'get_mime_type', 'import_optional_dependency', 'is_dependency_installed', 'load_artifact_from_memory', 'minify_json', 'references_from_artifacts', 'remove_key_in_dict_recursively', 'remove_null_values_in_dict_recursively', 'resolve_refs', 'str_to_hash', 'with_contextvars']` module-attribute

Utility for running a chat with a Structure.

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">Structure</code>	The Structure to run.
<code class="language-python">	<code class="language-python">list[str]</code>	Keywords that will exit the chat.
<a href="./#griptape.utils.Chat.exiting_text" title="exiting_text = field(default='Exiting...', kw_only=True) class-attribute instance-attribute (griptape.utils.Chat.exiting_text)"><code class="language-python">	*str*	Text to display when exiting the chat.
<a href="./#griptape.utils.Chat.processing_text" title="processing_text = field(default='Thinking...', kw_only=True) class-attribute instance-attribute (griptape.utils.Chat.processing_text)"><code class="language-python">	*str*	Text to display while processing the user's input.

continues on next page

Table 416 – continued from previous page

Name	Type	Description
<a href="./#griptape.utils.Chat.intro_text" title="intro_text = field(default=None, kw_only=True) class-attribute instance-attribute (griptape.utils.Chat.intro_text)"><code class="language-python">	*Optional[str]*	Text to display when the chat starts.
<a href="./#griptape.utils.Chat.prompt_prefix" title="prompt_prefix = field(default='User: ', kw_only=True) class-attribute instance-attribute (griptape.utils.Chat.prompt_prefix)"><code class="language-python">	*str*	Prefix for the user's input.
<a href="./#griptape.utils.Chat.response_prefix" title="response_prefix = field(default='Assistant: ', kw_only=True) class-attribute instance-attribute (griptape.utils.Chat.response_prefix)"><code class="language-python">	*str*	Prefix for the assistant's response.
<a href="./#griptape.utils.Chat.handle_input" title="handle_input = field(default=Factory(lambda self: self.default_handle_input, takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.utils.Chat.handle_input)"><code class="language-python">	*Callable[[str], str]*	Function to get the user's input.

continues on next page

Table 416 – continued from previous page

Name	Type	Description
<a href="./#griptape.utils.Chat.handle_output" title="handle_output = field(default=Factory(lambda self: self.default_handle_output, takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.utils.Chat.handle_output)"><code class="language-python">	*Callable[..., None]*	Function to output text. Takes a *text* argument for the text to output. Also takes a *stream* argument which will be set to True when streaming Prompt Tasks are present.

Source Code in griptape/utils/chat.py

```
@define
class Chat:
    """Utility for running a chat with a Structure.

    Attributes:
        structure: The Structure to run.
        exit_keywords: Keywords that will exit the chat.
        exiting_text: Text to display when exiting the chat.
        processing_text: Text to display while processing the users input.
        intro_text: Text to display when the chat starts.
        prompt_prefix: Prefix for the users input.
        response_prefix: Prefix for the assistants response.
        handle_input: Function to get the users input.
        handle_output: Function to output text. Takes a `text` argument for the text␣
↪to output.
                       Also takes a `stream` argument which will be set to True when␣
↪streaming Prompt Tasks are present.
    """

    class ChatPrompt(Prompt):
        prompt_suffix = ""  # We dont want richs default prompt suffix

    structure: Structure = field()
    exit_keywords: list[str] = field(default=["exit"], kw_only=True)
    exiting_text: str = field(default="Exiting...", kw_only=True)
    processing_text: str = field(default="Thinking...", kw_only=True)
    intro_text: Optional[str] = field(default=None, kw_only=True)
    prompt_prefix: str = field(default="User: ", kw_only=True)
    response_prefix: str = field(default="Assistant: ", kw_only=True)
    handle_input: Callable[[str], str] = field(
        default=Factory(lambda self: self.default_handle_input, takes_self=True), kw_
↪only=True
    )
```

(continues on next page)

(continued from previous page)

```
    handle_output: Callable[..., None] = field(
        default=Factory(lambda self: self.default_handle_output, takes_self=True),
        kw_only=True,
    )
    logger_level: int = field(default=logging.ERROR, kw_only=True)

    def default_handle_input(self, prompt_prefix: str) -> str:
        return Chat.ChatPrompt.ask(prompt_prefix)

    def default_handle_output(self, text: str, *, stream: bool = False) -> None:
        if stream:
            rprint(text, end="", flush=True)
        else:
            rprint(text)

    def start(self) -> None:
        from griptape.configs import Defaults

        # Hide Gen AI Builders logging output except for errors
        old_logger_level = logging.getLogger(Defaults.logging_config.logger_name).
↪getEffectiveLevel()
        logging.getLogger(Defaults.logging_config.logger_name).setLevel(self.logger_
↪level)

        if self.intro_text:
            self._call_handle_output(self.intro_text)

        has_streaming_tasks = self._has_streaming_tasks()
        while True:
            question = self.handle_input(self.prompt_prefix)

            if question.lower() in self.exit_keywords:
                self._call_handle_output(self.exiting_text)
                break

            if has_streaming_tasks:
                self._call_handle_output(self.processing_text)
                stream = Stream(self.structure).run(question)
                first_chunk = next(stream)
                self._call_handle_output(self.response_prefix + first_chunk.value,␣
↪stream=True)
                for chunk in stream:
                    self._call_handle_output(chunk.value, stream=True)
            else:
                self._call_handle_output(self.processing_text)
                self._call_handle_output(f"{self.response_prefix}{self.structure.
↪run(question).output.to_text()}")

        # Restore the original logger level
        logging.getLogger(Defaults.logging_config.logger_name).setLevel(old_logger_
↪level)
```

(continues on next page)

(continued from previous page)

```
    def _has_streaming_tasks(self) -> bool:
        from griptape.tasks.prompt_task import PromptTask

        return any(isinstance(task, PromptTask) and task.prompt_driver.stream for␣
↪task in self.structure.tasks)

    def _call_handle_output(self, text: str, *, stream: bool = False) -> None:
        func_params = inspect.signature(self.handle_output).parameters.copy()
        has_kwargs = False
        for param in func_params.values():
            # if there is a **kwargs parameter, we can safely
            # pass all the params to the function
            if param.kind == inspect.Parameter.VAR_KEYWORD:
                has_kwargs = True
                break

        if "stream" in func_params or has_kwargs:
            self.handle_output(text, stream=stream)
        else:
            self.handle_output(text)
```

- `exit_keywords = field(default=['exit'], kw_only=True)` class-attribute instance-attribute
- `exiting_text = field(default='Exiting...', kw_only=True)` class-attribute instance-attribute
- `handle_input = field(default=Factory(lambda self: self.default_handle_input, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `handle_output = field(default=Factory(lambda self: self.default_handle_output, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `intro_text = field(default=None, kw_only=True)` class-attribute instance-attribute
- `logger_level = field(default=logging.ERROR, kw_only=True)` class-attribute instance-attribute
- `processing_text = field(default='Thinking...', kw_only=True)` class-attribute instance-attribute
- `prompt_prefix = field(default='User: ', kw_only=True)` class-attribute instance-attribute
- `response_prefix = field(default='Assistant: ', kw_only=True)` class-attribute instance-attribute
- `structure = field()` class-attribute instance-attribute

ChatPrompt

Bases:

Prompt

Source Code in griptape/utils/chat.py

```
class ChatPrompt(Prompt):
    prompt_suffix = ""  # We dont want richs default prompt suffix
```

- `prompt_suffix = ''` class-attribute instance-attribute

_call_handle_output(text, *, stream=False)

Source Code in griptape/utils/chat.py

```
def _call_handle_output(self, text: str, *, stream: bool = False) -> None:
    func_params = inspect.signature(self.handle_output).parameters.copy()
    has_kwargs = False
    for param in func_params.values():
        # if there is a **kwargs parameter, we can safely
        # pass all the params to the function
        if param.kind == inspect.Parameter.VAR_KEYWORD:
            has_kwargs = True
            break

    if "stream" in func_params or has_kwargs:
        self.handle_output(text, stream=stream)
    else:
        self.handle_output(text)
```

_has_streaming_tasks()

Source Code in griptape/utils/chat.py

```
def _has_streaming_tasks(self) -> bool:
    from griptape.tasks.prompt_task import PromptTask

    return any(isinstance(task, PromptTask) and task.prompt_driver.stream for task in␣
↪self.structure.tasks)
```

default_handle_input(prompt_prefix)

Source Code in griptape/utils/chat.py

```
def default_handle_input(self, prompt_prefix: str) -> str:
    return Chat.ChatPrompt.ask(prompt_prefix)
```

default_handle_output(text, *, stream=False)

Source Code in griptape/utils/chat.py

```
def default_handle_output(self, text: str, *, stream: bool = False) -> None:
    if stream:
        rprint(text, end="", flush=True)
    else:
        rprint(text)
```

start()

Source Code in griptape/utils/chat.py

```
def start(self) -> None:
    from griptape.configs import Defaults

    # Hide Gen AI Builders logging output except for errors
```

(continues on next page)

(continued from previous page)

```
    old_logger_level = logging.getLogger(Defaults.logging_config.logger_name).
↪getEffectiveLevel()
    logging.getLogger(Defaults.logging_config.logger_name).setLevel(self.logger_level)

    if self.intro_text:
        self._call_handle_output(self.intro_text)

    has_streaming_tasks = self._has_streaming_tasks()
    while True:
        question = self.handle_input(self.prompt_prefix)

        if question.lower() in self.exit_keywords:
            self._call_handle_output(self.exiting_text)
            break

        if has_streaming_tasks:
            self._call_handle_output(self.processing_text)
            stream = Stream(self.structure).run(question)
            first_chunk = next(stream)
            self._call_handle_output(self.response_prefix + first_chunk.value,␣
↪stream=True)
            for chunk in stream:
                self._call_handle_output(chunk.value, stream=True)
        else:
            self._call_handle_output(self.processing_text)
            self._call_handle_output(f"{self.response_prefix}{self.structure.
↪run(question).output.to_text()}")

    # Restore the original logger level
    logging.getLogger(Defaults.logging_config.logger_name).setLevel(old_logger_level)
```

CommandRunner

Source Code in griptape/utils/command_runner.py

```
@define
class CommandRunner:
    def run(self, command: str) -> BaseArtifact:
        process = subprocess.Popen(command, shell=True, stdout=subprocess.PIPE,␣
↪stderr=subprocess.PIPE)

        stdout, stderr = process.communicate()

        if len(stderr) == 0:
            return TextArtifact(stdout.strip().decode())
        return ErrorArtifact(f"error: {stderr.strip()}")
```

run(command)

Source Code in griptape/utils/command_runner.py

```
def run(self, command: str) -> BaseArtifact:
    process = subprocess.Popen(command, shell=True, stdout=subprocess.PIPE,␣
↪stderr=subprocess.PIPE)

    stdout, stderr = process.communicate()

    if len(stderr) == 0:
        return TextArtifact(stdout.strip().decode())
    return ErrorArtifact(f"error: {stderr.strip()}")
```

Conversation

Source Code in griptape/utils/conversation.py

```
@define(frozen=True)
class Conversation:
    memory: Optional[BaseConversationMemory] = field()

    @memory.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_memory(self, attribute: Attribute, value:␣
↪Optional[BaseConversationMemory]) -> None:
        if value is None:
            raise ValueError("Conversation memory must not be None.")

    def lines(self) -> list[str]:
        from griptape.memory.structure import SummaryConversationMemory

        lines = []

        for run in self.memory.runs if self.memory is not None else []:
            lines.extend((f"Q: {run.input}", f"A: {run.output}"))

        if isinstance(self.memory, SummaryConversationMemory):
            lines.append(f"Summary: {self.memory.summary}")

        return lines

    def prompt_stack(self) -> list[str]:
        from griptape.memory.structure import SummaryConversationMemory

        lines = []

        for stack in self.memory.to_prompt_stack().messages if self.memory is not␣
↪None else []:
            lines.append(f"{stack.role}: {stack.to_text()}")

        if isinstance(self.memory, SummaryConversationMemory):
            lines.append(f"Summary: {self.memory.summary}")
```

(continues on next page)

(continued from previous page)

```
        return lines

    def __str__(self) -> str:
        return str.join("\n", self.lines())
```

- `memory = field()` class-attribute instance-attribute

str()

Source Code in griptape/utils/conversation.py

```
def __str__(self) -> str:
    return str.join("\n", self.lines())
```

lines()

Source Code in griptape/utils/conversation.py

```
def lines(self) -> list[str]:
    from griptape.memory.structure import SummaryConversationMemory

    lines = []

    for run in self.memory.runs if self.memory is not None else []:
        lines.extend((f"Q: {run.input}", f"A: {run.output}"))

    if isinstance(self.memory, SummaryConversationMemory):
        lines.append(f"Summary: {self.memory.summary}")

    return lines
```

prompt_stack()

Source Code in griptape/utils/conversation.py

```
def prompt_stack(self) -> list[str]:
    from griptape.memory.structure import SummaryConversationMemory

    lines = []

    for stack in self.memory.to_prompt_stack().messages if self.memory is not None␣
↪else []:
        lines.append(f"{stack.role}: {stack.to_text()}")

    if isinstance(self.memory, SummaryConversationMemory):
        lines.append(f"Summary: {self.memory.summary}")

    return lines
```

validate_memory(attribute, value)

Source Code in griptape/utils/conversation.py

```
@memory.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_memory(self, attribute: Attribute, value:␣
↪Optional[BaseConversationMemory]) -> None:
    if value is None:
        raise ValueError("Conversation memory must not be None.")
```

GriptapeCloudStructure

Utility for working with Gen AI Builder Structures.

Attributes

Name	Type	Description
<a href="./#griptape.utils.GriptapeCloudStructure._event_listener" title="_event_listener = field(default=None, kw_only=True, alias='event_listener') class-attribute instance-attribute (griptape.utils.GriptapeCloudStructure._event_listener)"><code class="language-python">	<a href="../events#griptape.events.EventListener" title="EventListener (griptape.events.EventListener)"><code class="language-python">Optional[EventListener]</cc	Event Listener to use. Defaults to an EventListener with a GriptapeCloudEventListenerDriver.
<a href="./#griptape.utils.GriptapeCloudStructure._observability" title="_observability = field(default=None, kw_only=True, alias='observability') class-attribute instance-attribute (griptape.utils.GriptapeCloudStructure._observability)"><code class="language-python">	<a href="../observability/observability.mdx#griptape.observability.observability.Observability" title="Observability (griptape.observability.observability.Observability)"><code class="language-python">Optional[Observability]</co	Observability to use. Defaults to an Observability with a GriptapeCloudObservabilityDriver.

continues on next page

Table 417 – continued from previous page

Name	Type	Description
<a href="./#griptape.utils.GriptapeCloudStructure.observe" title="observe = field(default=False, kw_only=True) class-attribute instance-attribute (griptape.utils.GriptapeCloudStructure.observe)"><code class="language-python">	*bool*	Whether to enable observability. Enabling requires the *drivers-observability-griptape-cloud* extra.

Source Code in griptape/utils/griptape_cloud.py

```
@define()
class GriptapeCloudStructure:
    """Utility for working with Gen AI Builder Structures.

    Attributes:
        _event_listener: Event Listener to use. Defaults to an EventListener with a␣
↪GriptapeCloudEventListenerDriver.
        _observability: Observability to use. Defaults to an Observability with a␣
↪GriptapeCloudObservabilityDriver.
        observe: Whether to enable observability. Enabling requires the `drivers-
↪observability-griptape-cloud` extra.
    """

    _event_listener: Optional[EventListener] = field(default=None, kw_only=True,␣
↪alias="event_listener")
    _observability: Optional[Observability] = field(default=None, kw_only=True, alias=
↪"observability")
    observe: bool = field(default=False, kw_only=True)
    _output: Optional[BaseArtifact] = field(default=None, init=False)

    @lazy_property()
    def event_listener(self) -> EventListener:
        from griptape.drivers.event_listener.griptape_cloud import␣
↪GriptapeCloudEventListenerDriver

        return EventListener(event_listener_driver=GriptapeCloudEventListenerDriver())

    @lazy_property()
    def observability(self) -> Observability:
        from griptape.drivers.observability.griptape_cloud import␣
↪GriptapeCloudObservabilityDriver
        from griptape.observability.observability import Observability

        return Observability(observability_driver=GriptapeCloudObservabilityDriver())
```

(continues on next page)

(continued from previous page)

```
    @property
    def output(self) -> Optional[BaseArtifact]:
        return self._output

    @output.setter
    def output(self, value: BaseArtifact | Any) -> None:
        if isinstance(value, BaseArtifact):
            self._output = value
        elif isinstance(value, list):
            self._output = ListArtifact([self._to_artifact(item) for item in value])
        else:
            self._output = self._to_artifact(value)

    @property
    def structure_run_id(self) -> str:
        return os.environ["GT_CLOUD_STRUCTURE_RUN_ID"]

    @property
    def in_managed_environment(self) -> bool:
        return "GT_CLOUD_STRUCTURE_RUN_ID" in os.environ

    def __enter__(self) -> Self:
        from griptape.observability.observability import Observability

        if self.in_managed_environment:
            EventBus.add_event_listener(self.event_listener)

            if self.observe:
                Observability.set_global_driver(self.observability.observability_
↪driver)
                self.observability.observability_driver.__enter__()

        return self

    def __exit__(
        self,
        exc_type: Optional[type[BaseException]],
        exc_value: Optional[BaseException],
        exc_traceback: Optional[TracebackType],
    ) -> None:
        from griptape.observability.observability import Observability

        if self.in_managed_environment:
            if self.output is not None:
                EventBus.publish_event(FinishStructureRunEvent(output_task_
↪output=self.output), flush=True)
            EventBus.remove_event_listener(self.event_listener)

            if self.observe:
                Observability.set_global_driver(None)
                self.observability.observability_driver.__exit__(exc_type, exc_value,␣
↪exc_traceback)
```

(continues on next page)

(continued from previous page)

```
    def _to_artifact(self, value: Any) -> BaseArtifact:
        if isinstance(value, str):
            return TextArtifact(value)
        if isinstance(value, bool):
            return BooleanArtifact(value)
        if isinstance(value, dict):
            return JsonArtifact(value)
        if isinstance(value, bytes):
            return BlobArtifact(value)
        return GenericArtifact(value)
```

- `_event_listener = field(default=None, kw_only=True, alias='event_listener')` class-attribute instance-attribute
- `_observability = field(default=None, kw_only=True, alias='observability')` class-attribute instance-attribute
- `_output = field(default=None, init=False)` class-attribute instance-attribute
- `in_managed_environment` property
- `observe = field(default=False, kw_only=True)` class-attribute instance-attribute
- `output` property writable
- `structure_run_id` property

enter()

Source Code in griptape/utils/griptape_cloud.py

```
def __enter__(self) -> Self:
    from griptape.observability.observability import Observability

    if self.in_managed_environment:
        EventBus.add_event_listener(self.event_listener)

        if self.observe:
            Observability.set_global_driver(self.observability.observability_driver)
            self.observability.observability_driver.__enter__()

    return self
```

exit(exc_type, exc_value, exc_traceback)

Source Code in griptape/utils/griptape_cloud.py

```
def __exit__(
    self,
    exc_type: Optional[type[BaseException]],
    exc_value: Optional[BaseException],
    exc_traceback: Optional[TracebackType],
) -> None:
    from griptape.observability.observability import Observability
```

(continues on next page)

(continued from previous page)

```
    if self.in_managed_environment:
        if self.output is not None:
            EventBus.publish_event(FinishStructureRunEvent(output_task_output=self.
↪output), flush=True)
        EventBus.remove_event_listener(self.event_listener)

        if self.observe:
            Observability.set_global_driver(None)
            self.observability.observability_driver.__exit__(exc_type, exc_value, exc_
↪traceback)
```

_to_artifact(value)

Source Code in griptape/utils/griptape_cloud.py

```
def _to_artifact(self, value: Any) -> BaseArtifact:
    if isinstance(value, str):
        return TextArtifact(value)
    if isinstance(value, bool):
        return BooleanArtifact(value)
    if isinstance(value, dict):
        return JsonArtifact(value)
    if isinstance(value, bytes):
        return BlobArtifact(value)
    return GenericArtifact(value)
```

event_listener()

Source Code in griptape/utils/griptape_cloud.py

```
@lazy_property()
def event_listener(self) -> EventListener:
    from griptape.drivers.event_listener.griptape_cloud import␣
↪GriptapeCloudEventListenerDriver

    return EventListener(event_listener_driver=GriptapeCloudEventListenerDriver())
```

observability()

Source Code in griptape/utils/griptape_cloud.py

```
@lazy_property()
def observability(self) -> Observability:
    from griptape.drivers.observability.griptape_cloud import␣
↪GriptapeCloudObservabilityDriver
    from griptape.observability.observability import Observability

    return Observability(observability_driver=GriptapeCloudObservabilityDriver())
```

J2

Source Code in griptape/utils/j2.py

```
@define(frozen=True)
class J2:
    template_name: Optional[str] = field(default=None)
    templates_dir: str = field(default=abs_path("templates"), kw_only=True)
    environment: Environment = field(
        default=Factory(
            lambda self: Environment(loader=FileSystemLoader(self.templates_dir),␣
↪trim_blocks=True, lstrip_blocks=True),
            takes_self=True,
        ),
        kw_only=True,
    )

    def render(self, **kwargs) -> str:
        if self.template_name is None:
            raise ValueError("template_name is required.")
        return self.environment.get_template(self.template_name).render(kwargs).
↪rstrip()

    def render_from_string(self, value: str, **kwargs) -> str:
        return self.environment.from_string(value).render(kwargs)
```

- `environment = field(default=Factory(lambda self: Environment(loader=FileSystemLoader(self.templates_dir), trim_blocks=True, lstrip_blocks=True), takes_self=True), kw_only=True)` class-attribute instance-attribute
- `template_name = field(default=None)` class-attribute instance-attribute
- `templates_dir = field(default=abs_path('templates'), kw_only=True)` class-attribute instance-attribute

render(**kwargs)

Source Code in griptape/utils/j2.py

```
def render(self, **kwargs) -> str:
    if self.template_name is None:
        raise ValueError("template_name is required.")
    return self.environment.get_template(self.template_name).render(kwargs).rstrip()
```

render_from_string(value, **kwargs)

Source Code in griptape/utils/j2.py

```
def render_from_string(self, value: str, **kwargs) -> str:
    return self.environment.from_string(value).render(kwargs)
```

ManifestValidator

Source Code in griptape/utils/manifest_validator.py

```
class ManifestValidator:
    def validate(self, manifest: dict) -> dict:
        return self.schema().validate(manifest)

    def schema(self) -> Schema:
        return Schema({"version": "v1", "name": str, "description": str, "contact_
↪email": str, "legal_info_url": str})
```

schema()

Source Code in griptape/utils/manifest_validator.py

```
def schema(self) -> Schema:
    return Schema({"version": "v1", "name": str, "description": str, "contact_email":␣
↪str, "legal_info_url": str})
```

validate(manifest)

Source Code in griptape/utils/manifest_validator.py

```
def validate(self, manifest: dict) -> dict:
    return self.schema().validate(manifest)
```

PythonRunner

Source Code in griptape/utils/python_runner.py

```
@define
class PythonRunner:
    libs: dict[str, str] = field(factory=dict, kw_only=True)

    def run(self, code: str) -> str:
        global_stdout = sys.stdout
        sys.stdout = local_stdout = StringIO()

        try:
            for lib, alias in self.libs.items():
                imported_lib = importlib.import_module(lib)
                globals()[alias] = imported_lib

            exec(f"print({code})", {}, {alias: eval(alias) for alias in self.libs.
↪values()})

            output = local_stdout.getvalue()
        except Exception as e:
            output = str(e)
        finally:
            sys.stdout = global_stdout

        return output.strip()
```

- `libs = field(factory=dict, kw_only=True)` class-attribute instance-attribute

run(code)

Source Code in griptape/utils/python_runner.py

```
def run(self, code: str) -> str:
    global_stdout = sys.stdout
    sys.stdout = local_stdout = StringIO()

    try:
        for lib, alias in self.libs.items():
            imported_lib = importlib.import_module(lib)
            globals()[alias] = imported_lib

        exec(f"print({code})", {}, {alias: eval(alias) for alias in self.libs.
↪values()})

        output = local_stdout.getvalue()
    except Exception as e:
        output = str(e)
    finally:
        sys.stdout = global_stdout

    return output.strip()
```

Stream

A wrapper for Structures filters Events relevant to text output and converts them to TextArtifacts.

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">Structure</code>	The Structure to wrap.

Source Code in griptape/utils/stream.py

```
@define
class Stream:
    """A wrapper for Structures filters Events relevant to text output and converts␣
↪them to TextArtifacts.

    Attributes:
        structure: The Structure to wrap.
    """
```

(continues on next page)

(continued from previous page)

```
    structure: Structure = field()
    event_types: list[type[BaseEvent]] = field(
        default=Factory(lambda: [TextChunkEvent, ActionChunkEvent, FinishPromptEvent,␣
↪FinishStructureRunEvent])
    )

    def run(self, *args) -> Iterator[TextArtifact]:
        action_str = ""

        for event in self.structure.run_stream(*args, event_types=self.event_types):
            if isinstance(event, FinishPromptEvent):
                yield TextArtifact(value="\n")
            elif isinstance(event, TextChunkEvent):
                yield TextArtifact(value=event.token)
            elif isinstance(event, ActionChunkEvent):
                if event.tag is not None and event.name is not None and event.path is␣
↪not None:
                    yield TextArtifact(f"{event.name}.{event.tag} ({event.path})")
                if event.partial_input is not None:
                    action_str += event.partial_input
                    try:
                        yield TextArtifact(json.dumps(json.loads(action_str),␣
↪indent=2))
                        action_str = ""
                    except Exception:
                        pass
```

- `event_types = field(default=Factory(lambda: [TextChunkEvent, ActionChunkEvent, FinishPromptEvent, FinishStructureRunEvent]))` class-attribute instance-attribute
- `structure = field()` class-attribute instance-attribute

run(*args)

Source Code in griptape/utils/stream.py

```
def run(self, *args) -> Iterator[TextArtifact]:
    action_str = ""

    for event in self.structure.run_stream(*args, event_types=self.event_types):
        if isinstance(event, FinishPromptEvent):
            yield TextArtifact(value="\n")
        elif isinstance(event, TextChunkEvent):
            yield TextArtifact(value=event.token)
        elif isinstance(event, ActionChunkEvent):
            if event.tag is not None and event.name is not None and event.path is not␣
↪None:
                yield TextArtifact(f"{event.name}.{event.tag} ({event.path})")
            if event.partial_input is not None:
                action_str += event.partial_input
                try:
                    yield TextArtifact(json.dumps(json.loads(action_str), indent=2))
```

(continues on next page)

(continued from previous page)

```
                    action_str = ""
                except Exception:
                    pass
```

StructureVisualizer

Source Code in griptape/utils/structure_visualizer.py

```
@define
class StructureVisualizer:
    """Utility class to visualize a Structure structure."""

    structure: Structure = field()
    header: str = field(default="graph TD;", kw_only=True)
    build_node_id: Callable[[BaseTask], str] = field(default=lambda task: task.id.
↪title(), kw_only=True)
    query_params: dict[str, str] = field(factory=dict, kw_only=True)
    base_url: str = field(default="https://mermaid.ink", kw_only=True)

    def to_url(self) -> str:
        """Generates a url that renders the Workflow structure as a Mermaid flowchart.

        Reference: https://mermaid.js.org/ecosystem/tutorials#jupyter-integration-
↪with-mermaid-js.

        Returns:
            str: URL to the rendered image
        """
        self.structure.resolve_relationships()

        tasks = self.__render_tasks(self.structure.tasks)
        graph = f"{self.header}{tasks}"

        graph_bytes = graph.encode("utf-8")
        base64_string = base64.b64encode(graph_bytes).decode("utf-8")

        url = urllib.parse.urljoin(self.base_url, f"svg/{base64_string}")
        req = PreparedRequest()
        req.prepare_url(url, self.query_params)

        if req.url is None:
            raise ValueError("Failed to generate the URL")

        return req.url

    def __render_tasks(self, tasks: list[BaseTask]) -> str:
        return "\n\t" + "\n\t".join([self.__render_task(task) for task in tasks])

    def __render_task(self, task: BaseTask) -> str:
        from griptape.drivers.structure_run.local import LocalStructureRunDriver
        from griptape.tasks import BranchTask, StructureRunTask
```

(continues on next page)

(continued from previous page)

```
        parts = []
        if task.children:
            children = " & ".join([f"{self.build_node_id(child)}" for child in task.
↪children])
            if isinstance(task, BranchTask):
                parts.append(f"{self.build_node_id(task)}{{ {self.build_node_id(task)}
↪ }}-.-> {children};")
            else:
                parts.append(f"{self.build_node_id(task)}--> {children};")
        else:
            parts.append(f"{self.build_node_id(task)};")

        if isinstance(task, StructureRunTask) and isinstance(task.structure_run_
↪driver, LocalStructureRunDriver):
            sub_structure = task.structure_run_driver.create_structure()
            sub_tasks = self.__render_tasks(sub_structure.tasks)
            parts.append(f"subgraph {self.build_node_id(task)}{sub_tasks}\n\tend")

        return "\n\t".join(parts)
```

- `base_url = field(default='https://mermaid.ink', kw_only=True)` class-attribute instance-attribute
- `build_node_id = field(default=lambda task: task.id.title(), kw_only=True)` class-attribute instance-attribute
- `header = field(default='graph TD;', kw_only=True)` class-attribute instance-attribute
- `query_params = field(factory=dict, kw_only=True)` class-attribute instance-attribute
- `structure = field()` class-attribute instance-attribute

__render_task(task)

Source Code in griptape/utils/structure_visualizer.py

```
def __render_task(self, task: BaseTask) -> str:
    from griptape.drivers.structure_run.local import LocalStructureRunDriver
    from griptape.tasks import BranchTask, StructureRunTask

    parts = []
    if task.children:
        children = " & ".join([f"{self.build_node_id(child)}" for child in task.
↪children])
        if isinstance(task, BranchTask):
            parts.append(f"{self.build_node_id(task)}{{ {self.build_node_id(task)} }}-
↪.-> {children};")
        else:
            parts.append(f"{self.build_node_id(task)}--> {children};")
    else:
        parts.append(f"{self.build_node_id(task)};")

    if isinstance(task, StructureRunTask) and isinstance(task.structure_run_driver,␣
↪LocalStructureRunDriver):
```

(continues on next page)

(continued from previous page)

```
        sub_structure = task.structure_run_driver.create_structure()
        sub_tasks = self.__render_tasks(sub_structure.tasks)
        parts.append(f"subgraph {self.build_node_id(task)}{sub_tasks}\n\tend")

    return "\n\t".join(parts)
```

__render_tasks(tasks)

Source Code in griptape/utils/structure_visualizer.py

```
def __render_tasks(self, tasks: list[BaseTask]) -> str:
    return "\n\t" + "\n\t".join([self.__render_task(task) for task in tasks])
```

to_url()

Generates a url that renders the Workflow structure as a Mermaid flowchart.

Reference: https://mermaid.js.org/ecosystem/tutorials#jupyter-integration-with-mermaid-js.

Returns

Name	Type	Description
str	*str*	URL to the rendered image

Source Code in griptape/utils/structure_visualizer.py

```
def to_url(self) -> str:
    """Generates a url that renders the Workflow structure as a Mermaid flowchart.

    Reference: https://mermaid.js.org/ecosystem/tutorials#jupyter-integration-with-
↪mermaid-js.

    Returns:
        str: URL to the rendered image
    """
    self.structure.resolve_relationships()

    tasks = self.__render_tasks(self.structure.tasks)
    graph = f"{self.header}{tasks}"

    graph_bytes = graph.encode("utf-8")
    base64_string = base64.b64encode(graph_bytes).decode("utf-8")

    url = urllib.parse.urljoin(self.base_url, f"svg/{base64_string}")
    req = PreparedRequest()
    req.prepare_url(url, self.query_params)

    if req.url is None:
        raise ValueError("Failed to generate the URL")
```

(continues on next page)

(continued from previous page)

```
    return req.url
```

TokenCounter

Source Code in griptape/utils/token_counter.py

```
@define
class TokenCounter:
    tokens: int = field(default=0, kw_only=True)

    def add_tokens(self, new_tokens: float) -> int:
        self.tokens += int(new_tokens)

        return self.tokens
```

- `tokens = field(default=0, kw_only=True)` class-attribute instance-attribute

add_tokens(new_tokens)

Source Code in griptape/utils/token_counter.py

```
def add_tokens(self, new_tokens: float) -> int:
    self.tokens += int(new_tokens)

    return self.tokens
```

add_key_in_dict_recursively(d, key, value, criteria=None)

Add a key in a dictionary recursively.

Parameters

Name	Type	Description	Default
d	*Any*	The dictionary to add the key to. 	*required*
key	*str*	The key to add. 	*required*
value	*Any*	The value to add. 	*required*
criteria	*Optional[Callable[[dict], bool]]*	An optional function to determine if the key should be added. 	*None*

Source Code in griptape/utils/dict_utils.py

```
def add_key_in_dict_recursively(
    d: Any, key: str, value: Any, criteria: Optional[Callable[[dict], bool]] = None
) -> dict:
    """Add a key in a dictionary recursively.
```

(continues on next page)

(continued from previous page)

```
    Args:
        d: The dictionary to add the key to.
        key: The key to add.
        value: The value to add.
        criteria: An optional function to determine if the key should be added.
    """
    if isinstance(d, dict):
        if criteria is None or criteria(d):
            d[key] = value
        return {k: add_key_in_dict_recursively(v, key, value, criteria) for k, v in d.
↪items()}
    return d
```

build_strict_schema(json_schema, schema_id)

Performs a series of post-processing steps to ensure a JSON schema is compatible with LLMs.

1. Adds the `$id` and `$schema` keys.
2. Sets `additionalProperties` to `False` for objects without this key.
3. Resolves `$ref` s and removes `$defs` .

Parameters

Name	Type	Description	Default
json_schema	*dict*	The JSON schema to ensure is strict. 	*required*
schema_id	*str*	The ID of the schema. 	*required*

Source Code in griptape/utils/json_schema_utils.py

```
def build_strict_schema(json_schema: dict, schema_id: str) -> dict:
    """Performs a series of post-processing steps to ensure a JSON schema is␣
↪compatible with LLMs.

    1. Adds the `$id` and `$schema` keys.
    2. Sets `additionalProperties` to `False` for objects without this key.
    3. Resolves `$ref`s and removes `$defs`.

    Args:
        json_schema: The JSON schema to ensure is strict.
        schema_id: The ID of the schema.

    Returns: The strict JSON schema.
    """
    from griptape.utils.dict_utils import add_key_in_dict_recursively

    json_schema.setdefault("$id", schema_id)
```

(continues on next page)

(continued from previous page)

```
    json_schema.setdefault("$schema", "http://json-schema.org/draft-07/schema#")
    json_schema = add_key_in_dict_recursively(
        json_schema,
        key="additionalProperties",
        value=False,
        criteria=lambda d: d.get("type") == "object" and "additionalProperties" not␣
↪in d,
    )
    json_schema = resolve_refs(json_schema)
    if "$defs" in json_schema:
        del json_schema["$defs"]

    return json_schema
```

deprecation_warn(message, stacklevel=2)

Source Code in griptape/utils/deprecation.py

```
def deprecation_warn(message: str, stacklevel: int = 2) -> None:
    warnings.simplefilter("always", DeprecationWarning)
    warnings.warn(message, category=DeprecationWarning, stacklevel=stacklevel)
    warnings.simplefilter("default", DeprecationWarning)
```

dict_merge(dct, merge_dct, *, add_keys=True)

Recursive dict merge.

Inspired by :meth:`dict.update()` , instead of updating only top-level keys, dict_merge recurses down into dicts nested to an arbitrary depth, updating keys. The `merge_dct` is merged into `dct` .

This version will return a copy of the dictionary and leave the original arguments untouched.

The optional argument `add_keys` , determines whether keys which are present in `merge_dict` but not `dct` should be included in the new dict.

Parameters

Name	Type	Description	Default
dct	*Optional[dict]*	onto which the merge is executed 	*required*
merge_dct	*Optional[dict]*	dct merged into dct 	*required*
add_keys	*bool*	whether to add new keys 	*True*

Returns

Name	Type	Description
dict	*dict*	updated dict

Source Code in griptape/utils/dict_utils.py

```
def dict_merge(dct: Optional[dict], merge_dct: Optional[dict], *, add_keys: bool =␣
↪True) -> dict:
    """Recursive dict merge.

    Inspired by :meth:``dict.update()``, instead of
    updating only top-level keys, dict_merge recurses down into dicts nested
    to an arbitrary depth, updating keys. The ``merge_dct`` is merged into
    ``dct``.

    This version will return a copy of the dictionary and leave the original
    arguments untouched.

    The optional argument ``add_keys``, determines whether keys which are
    present in ``merge_dict`` but not ``dct`` should be included in the
    new dict.

    Args:
        dct: onto which the merge is executed
        merge_dct: dct merged into dct
        add_keys: whether to add new keys

    Returns:
        dict: updated dict
    """
    dct = {} if dct is None else dct
    merge_dct = {} if merge_dct is None else merge_dct

    dct = dct.copy()

    if not add_keys:
        merge_dct = {k: merge_dct[k] for k in set(dct).intersection(set(merge_dct))}

    for key in merge_dct:
        if key in dct and isinstance(dct[key], dict):
            dct[key] = dict_merge(dct[key], merge_dct[key], add_keys=add_keys)
        else:
            dct[key] = merge_dct[key]

    return dct
```

execute_futures_dict(fs_dict)

Source Code in griptape/utils/futures.py

```
def execute_futures_dict(fs_dict: dict[str, futures.Future[T]]) -> dict[str, T]:
    futures.wait(fs_dict.values(), timeout=None, return_when=futures.ALL_COMPLETED)

```

(continues on next page)

(continued from previous page)

```
    return {key: future.result() for key, future in fs_dict.items()}
```

execute_futures_list(fs_list)

Source Code in griptape/utils/futures.py

```
def execute_futures_list(fs_list: list[futures.Future[T]]) -> list[T]:
    futures.wait(fs_list, timeout=None, return_when=futures.ALL_COMPLETED)

    return [future.result() for future in fs_list]
```

execute_futures_list_dict(fs_dict)

Source Code in griptape/utils/futures.py

```
def execute_futures_list_dict(fs_dict: dict[str, list[futures.Future[T]]]) ->␣
↪dict[str, list[T]]:
    execute_futures_list([item for sublist in fs_dict.values() for item in sublist])

    return {key: [f.result() for f in fs] for key, fs in fs_dict.items()}
```

get_mime_type(file_path_or_bytes)

Attempt to determine the MIME type of a file or bytes.

If the input is a file path, we use the built-in `mimetypes` package to guess the MIME type.

If the input is bytes, we use the `filetype` library to determine the MIME type. If the library cannot determine the MIME type (data missing magic bytes), we use a few heuristics to guess the type.

Parameters

Name	Type	Description	Default
file_path_or_bytes	*str I bytes*	The path to the file or the bytes to check. 	*required*

Source Code in griptape/utils/file_utils.py

```
def get_mime_type(file_path_or_bytes: str | bytes) -> str:
    """Attempt to determine the MIME type of a file or bytes.

    If the input is a file path, we use the built-in `mimetypes` package to guess the␣
↪MIME type.

    If the input is bytes, we use the `filetype` library to determine the MIME type.
    If the library cannot determine the MIME type (data missing magic bytes), we use␣
↪a few heuristics to guess the type.

    Args:
```

(continues on next page)

(continued from previous page)

```
        file_path_or_bytes: The path to the file or the bytes to check.

    Returns: The MIME type of the file or bytes.
    """
    if isinstance(file_path_or_bytes, bytes):
        filetype_guess = filetype.guess(file_path_or_bytes)

        if filetype_guess is None:
            if _is_text(file_path_or_bytes):
                if _is_json(file_path_or_bytes):
                    return "application/json"
                if _is_csv(file_path_or_bytes):
                    return "text/csv"
                return "text/plain"
            return "application/octet-stream"
        return filetype_guess.mime
    type_, _ = mimetypes.guess_type(file_path_or_bytes)
    if type_ is None:
        return "application/octet-stream"
    return type_
```

import_optional_dependency(name)

Import an optional dependency.

If a dependency is missing, an ImportError with a nice message will be raised.

Parameters

Name	Type	Description	Default
name	*str*	The module name. 	*required*

Returns

Type	Description
ModuleType	The imported module, when found.

Source Code in griptape/utils/import_utils.py

```
def import_optional_dependency(name: str) -> ModuleType:
    """Import an optional dependency.

    If a dependency is missing, an ImportError with a nice message will be raised.

    Args:
        name: The module name.

    Returns:
```

(continues on next page)

(continued from previous page)

```
        The imported module, when found.
    """
    package_name = INSTALL_MAPPING.get(name)
    install_name = package_name if package_name is not None else name

    msg = (
        f"Missing optional dependency: {install_name}. "
        f"Please install the appropriate extra: https://docs.griptape.ai/stable/
↪griptape-framework/#extras."
    )
    try:
        module = import_module(name)
    except ImportError as exc:
        raise ImportError(msg) from exc

    return module
```

is_dependency_installed(name)

Check if an optional dependency is available.

Parameters

Name	Type	Description	Default
name	*str*	The module name. 	*required*

Returns

Type	Description
bool	True if the dependency is available.
bool	False if the dependency is not available.

Source Code in griptape/utils/import_utils.py

```
def is_dependency_installed(name: str) -> bool:
    """Check if an optional dependency is available.

    Args:
        name: The module name.

    Returns:
        True if the dependency is available.
        False if the dependency is not available.
    """
    try:
        import_optional_dependency(name)
    except ImportError:
```

(continues on next page)

(continued from previous page)

```
        return False

    return True
```

minify_json(value)

Source Code in griptape/utils/**init**.py

```
def minify_json(value: str) -> str:
    return json.dumps(json.loads(value), separators=(",", ":"))
```

references_from_artifacts(artifacts)

Source Code in griptape/utils/reference_utils.py

```
def references_from_artifacts(artifacts: list[TextArtifact]) -> list[Reference]:
    references = []

    for a in artifacts:
        if a.reference is not None and a.reference not in references:
            references.append(a.reference)

    return references
```

remove_key_in_dict_recursively(d, key)

Source Code in griptape/utils/dict_utils.py

```
def remove_key_in_dict_recursively(d: dict, key: str) -> dict:
    if isinstance(d, dict):
        return {k: remove_key_in_dict_recursively(v, key) for k, v in d.items() if k !
↪= key}
    return d
```

remove_null_values_in_dict_recursively(d)

Source Code in griptape/utils/dict_utils.py

```
def remove_null_values_in_dict_recursively(d: dict) -> dict:
    if isinstance(d, dict):
        return {k: remove_null_values_in_dict_recursively(v) for k, v in d.items() if␣
↪v is not None}
    return d
```

resolve_refs(schema)

Recursively resolve all local $refs in the given JSON Schema using $defs as the source.

Required since pydantic does not support nested schemas without $refs. https://github.com/pydantic/pydantic/issues/889

Parameters

Name	Type	Description	Default
schema	*dict[str, Any]*	A JSON Schema as a dictionary, which may contain "$refs" and "$defs". 	*required*

Source Code in griptape/utils/json_schema_utils.py

```
def resolve_refs(schema: dict[str, Any]) -> dict[str, Any]:
    """Recursively resolve all local $refs in the given JSON Schema using $defs as the source.

    Required since pydantic does not support nested schemas without $refs. https://github.com/pydantic/pydantic/issues/889

    Args:
         schema: A JSON Schema as a dictionary, which may contain "$refs" and "$defs".
    Returns:  A new dictionary with all local $refs resolved against $defs.
    """
    defs = schema.get("$defs", {})
    # Work on a deep copy so we dont mutate the original schema.
    schema_copy = deepcopy(schema)

    def _resolve(node: Any) -> Any:
        """Recursively walk through the node, resolving any local $refs to the corresponding definitions in defs."""
        if isinstance(node, dict):
            # If this node is a reference to something in #/$defs/...
            ref = node.get("$ref")
            if isinstance(ref, str) and ref.startswith("#/$defs/"):
                def_name = ref.replace("#/$defs/", "")
                if def_name in defs:
                    # Replace the entire node with the referenced definition.
                    return _resolve(deepcopy(defs[def_name]))
                raise KeyError(f"Definition {def_name} not found in $defs.")

            # If not a ref, or doesnt start with #/$defs/, just walk deeper.
            return {key: _resolve(value) for key, value in node.items()}

        if isinstance(node, list):
            # Recurse into each item of the list
            return [_resolve(item) for item in node]

        # For scalars (str, int, bool, None, etc.), return them as-is.
        return node

    return _resolve(schema_copy)
```

str_to_hash(text, hash_algorithm='sha256')

Source Code in griptape/utils/hash.py

```
def str_to_hash(text: str, hash_algorithm: str = "sha256") -> str:
    m = hashlib.new(hash_algorithm)

    m.update(text.encode())

    return m.hexdigest()
```

with_contextvars(wrapped)

Source Code in griptape/utils/contextvars_utils.py

```
def with_contextvars(wrapped: Callable) -> Callable:
    ctx = contextvars.copy_context()

    def wrapper(*args, **kwargs) -> Any:
        return ctx.run(wrapped, *args, **kwargs)

    return wrapper
```

chat

Utility for running a chat with a Structure.

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">Structure</code>	The Structure to run.
<code class="language-python">	<code class="language-python">list[str]</code>	Keywords that will exit the chat.

continues on next page

Table 430 – continued from previous page

Name	Type	Description
<a href="chat.mdx#grip-tape.utils.chat.Chat.exiting_text" title="exiting_text = field(default='Exiting...', kw_only=True) class-attribute instance-attribute (grip-tape.utils.chat.Chat.exiting_text)"><code class="language-python">	*str*	Text to display when exiting the chat.
<a href="chat.mdx#grip-tape.utils.chat.Chat.processing_text" title="processing_text = field(default='Thinking...', kw_only=True) class-attribute instance-attribute (grip-tape.utils.chat.Chat.processing_text)"><code class="language-python">	*str*	Text to display while processing the user's input.
<a href="chat.mdx#grip-tape.utils.chat.Chat.intro_text" title="intro_text = field(default=None, kw_only=True) class-attribute instance-attribute (grip-tape.utils.chat.Chat.intro_text)"><code class="language-python">	*Optional[str]*	Text to display when the chat starts.
<a href="chat.mdx#grip-tape.utils.chat.Chat.prompt_prefix" title="prompt_prefix = field(default='User: ', kw_only=True) class-attribute instance-attribute (grip-tape.utils.chat.Chat.prompt_prefix)"><code class="language-python">	*str*	Prefix for the user's input.

continues on next page

Table 430 – continued from previous page

Name	Type	Description
<a href="chat.mdx#griptape.utils.chat.Chat.response_prefix" title="response_prefix = field(default='Assistant: ', kw_only=True) class-attribute instance-attribute (griptape.utils.chat.Chat.response_prefix)"><code class="language-python">	*str*	Prefix for the assistant's response.
<a href="chat.mdx#griptape.utils.chat.Chat.handle_input" title="handle_input = field(default=Factory(lambda self: self.default_handle_input, takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.utils.chat.Chat.handle_input)"><code class="language-python">	*Callable[[str], str]*	Function to get the user's input.
<a href="chat.mdx#griptape.utils.chat.Chat.handle_output" title="handle_output = field(default=Factory(lambda self: self.default_handle_output, takes_self=True), kw_only=True) class-attribute instance-attribute (griptape.utils.chat.Chat.handle_output)"><code class="language-python">	*Callable[..., None]*	Function to output text. Takes a *text* argument for the text to output. Also takes a *stream* argument which will be set to True when streaming Prompt Tasks are present.

Source Code in griptape/utils/chat.py

```
@define
class Chat:
    """Utility for running a chat with a Structure.

    Attributes:
```

(continues on next page)

(continued from previous page)

```
        structure: The Structure to run.
        exit_keywords: Keywords that will exit the chat.
        exiting_text: Text to display when exiting the chat.
        processing_text: Text to display while processing the users input.
        intro_text: Text to display when the chat starts.
        prompt_prefix: Prefix for the users input.
        response_prefix: Prefix for the assistants response.
        handle_input: Function to get the users input.
        handle_output: Function to output text. Takes a `text` argument for the text␣
↪to output.
                    Also takes a `stream` argument which will be set to True when␣
↪streaming Prompt Tasks are present.
    """

    class ChatPrompt(Prompt):
        prompt_suffix = ""  # We dont want richs default prompt suffix

    structure: Structure = field()
    exit_keywords: list[str] = field(default=["exit"], kw_only=True)
    exiting_text: str = field(default="Exiting...", kw_only=True)
    processing_text: str = field(default="Thinking...", kw_only=True)
    intro_text: Optional[str] = field(default=None, kw_only=True)
    prompt_prefix: str = field(default="User: ", kw_only=True)
    response_prefix: str = field(default="Assistant: ", kw_only=True)
    handle_input: Callable[[str], str] = field(
        default=Factory(lambda self: self.default_handle_input, takes_self=True), kw_
↪only=True
    )
    handle_output: Callable[..., None] = field(
        default=Factory(lambda self: self.default_handle_output, takes_self=True),
        kw_only=True,
    )
    logger_level: int = field(default=logging.ERROR, kw_only=True)

    def default_handle_input(self, prompt_prefix: str) -> str:
        return Chat.ChatPrompt.ask(prompt_prefix)

    def default_handle_output(self, text: str, *, stream: bool = False) -> None:
        if stream:
            rprint(text, end="", flush=True)
        else:
            rprint(text)

    def start(self) -> None:
        from griptape.configs import Defaults

        # Hide Gen AI Builders logging output except for errors
        old_logger_level = logging.getLogger(Defaults.logging_config.logger_name).
↪getEffectiveLevel()
        logging.getLogger(Defaults.logging_config.logger_name).setLevel(self.logger_
↪level)
```

(continues on next page)

(continued from previous page)

```
        if self.intro_text:
            self._call_handle_output(self.intro_text)

        has_streaming_tasks = self._has_streaming_tasks()
        while True:
            question = self.handle_input(self.prompt_prefix)

            if question.lower() in self.exit_keywords:
                self._call_handle_output(self.exiting_text)
                break

            if has_streaming_tasks:
                self._call_handle_output(self.processing_text)
                stream = Stream(self.structure).run(question)
                first_chunk = next(stream)
                self._call_handle_output(self.response_prefix + first_chunk.value,␣
↪stream=True)
                for chunk in stream:
                    self._call_handle_output(chunk.value, stream=True)
            else:
                self._call_handle_output(self.processing_text)
                self._call_handle_output(f"{self.response_prefix}{self.structure.
↪run(question).output.to_text()}")

        # Restore the original logger level
        logging.getLogger(Defaults.logging_config.logger_name).setLevel(old_logger_
↪level)

    def _has_streaming_tasks(self) -> bool:
        from griptape.tasks.prompt_task import PromptTask

        return any(isinstance(task, PromptTask) and task.prompt_driver.stream for␣
↪task in self.structure.tasks)

    def _call_handle_output(self, text: str, *, stream: bool = False) -> None:
        func_params = inspect.signature(self.handle_output).parameters.copy()
        has_kwargs = False
        for param in func_params.values():
            # if there is a **kwargs parameter, we can safely
            # pass all the params to the function
            if param.kind == inspect.Parameter.VAR_KEYWORD:
                has_kwargs = True
                break

        if "stream" in func_params or has_kwargs:
            self.handle_output(text, stream=stream)
        else:
            self.handle_output(text)
```

- `exit_keywords = field(default=['exit'], kw_only=True)` class-attribute instance-attribute
- `exiting_text = field(default='Exiting...', kw_only=True)` class-attribute instance-attribute
- `handle_input = field(default=Factory(lambda self: self.default_handle_input,`

`takes_self=True), kw_only=True)` class-attribute instance-attribute

- `handle_output = field(default=Factory(lambda self: self.default_handle_output, takes_self=True), kw_only=True)` class-attribute instance-attribute
- `intro_text = field(default=None, kw_only=True)` class-attribute instance-attribute
- `logger_level = field(default=logging.ERROR, kw_only=True)` class-attribute instance-attribute
- `processing_text = field(default='Thinking...', kw_only=True)` class-attribute instance-attribute
- `prompt_prefix = field(default='User: ', kw_only=True)` class-attribute instance-attribute
- `response_prefix = field(default='Assistant: ', kw_only=True)` class-attribute instance-attribute
- `structure = field()` class-attribute instance-attribute

ChatPrompt

Bases:

Prompt

Source Code in griptape/utils/chat.py

```
class ChatPrompt(Prompt):
    prompt_suffix = ""  # We dont want richs default prompt suffix
```

- `prompt_suffix = ''` class-attribute instance-attribute

_call_handle_output(text, *, stream=False)

Source Code in griptape/utils/chat.py

```
def _call_handle_output(self, text: str, *, stream: bool = False) -> None:
    func_params = inspect.signature(self.handle_output).parameters.copy()
    has_kwargs = False
    for param in func_params.values():
        # if there is a **kwargs parameter, we can safely
        # pass all the params to the function
        if param.kind == inspect.Parameter.VAR_KEYWORD:
            has_kwargs = True
            break

    if "stream" in func_params or has_kwargs:
        self.handle_output(text, stream=stream)
    else:
        self.handle_output(text)
```

_has_streaming_tasks()

Source Code in griptape/utils/chat.py

```
def _has_streaming_tasks(self) -> bool:
    from griptape.tasks.prompt_task import PromptTask
```

(continues on next page)

(continued from previous page)

```
    return any(isinstance(task, PromptTask) and task.prompt_driver.stream for task in␣
↪self.structure.tasks)
```

default_handle_input(prompt_prefix)

Source Code in griptape/utils/chat.py

```
def default_handle_input(self, prompt_prefix: str) -> str:
    return Chat.ChatPrompt.ask(prompt_prefix)
```

default_handle_output(text, *, stream=False)

Source Code in griptape/utils/chat.py

```
def default_handle_output(self, text: str, *, stream: bool = False) -> None:
    if stream:
        rprint(text, end="", flush=True)
    else:
        rprint(text)
```

start()

Source Code in griptape/utils/chat.py

```
def start(self) -> None:
    from griptape.configs import Defaults

    # Hide Gen AI Builders logging output except for errors
    old_logger_level = logging.getLogger(Defaults.logging_config.logger_name).
↪getEffectiveLevel()
    logging.getLogger(Defaults.logging_config.logger_name).setLevel(self.logger_level)

    if self.intro_text:
        self._call_handle_output(self.intro_text)

    has_streaming_tasks = self._has_streaming_tasks()
    while True:
        question = self.handle_input(self.prompt_prefix)

        if question.lower() in self.exit_keywords:
            self._call_handle_output(self.exiting_text)
            break

        if has_streaming_tasks:
            self._call_handle_output(self.processing_text)
            stream = Stream(self.structure).run(question)
            first_chunk = next(stream)
            self._call_handle_output(self.response_prefix + first_chunk.value,␣
↪stream=True)
            for chunk in stream:
                self._call_handle_output(chunk.value, stream=True)
```

(continues on next page)

(continued from previous page)

```
        else:
            self._call_handle_output(self.processing_text)
            self._call_handle_output(f"{self.response_prefix}{self.structure.
↪run(question).output.to_text()}")

    # Restore the original logger level
    logging.getLogger(Defaults.logging_config.logger_name).setLevel(old_logger_level)
```

command_runner

Source Code in griptape/utils/command_runner.py

```
@define
class CommandRunner:
    def run(self, command: str) -> BaseArtifact:
        process = subprocess.Popen(command, shell=True, stdout=subprocess.PIPE,␣
↪stderr=subprocess.PIPE)

        stdout, stderr = process.communicate()

        if len(stderr) == 0:
            return TextArtifact(stdout.strip().decode())
        return ErrorArtifact(f"error: {stderr.strip()}")
```

run(command)

Source Code in griptape/utils/command_runner.py

```
def run(self, command: str) -> BaseArtifact:
    process = subprocess.Popen(command, shell=True, stdout=subprocess.PIPE,␣
↪stderr=subprocess.PIPE)

    stdout, stderr = process.communicate()

    if len(stderr) == 0:
        return TextArtifact(stdout.strip().decode())
    return ErrorArtifact(f"error: {stderr.strip()}")
```

constants

contextvars_utils

Source Code in griptape/utils/contextvars_utils.py

```
def with_contextvars(wrapped: Callable) -> Callable:
    ctx = contextvars.copy_context()

    def wrapper(*args, **kwargs) -> Any:
        return ctx.run(wrapped, *args, **kwargs)

    return wrapper
```

conversation

Source Code in griptape/utils/conversation.py

```
@define(frozen=True)
class Conversation:
    memory: Optional[BaseConversationMemory] = field()

    @memory.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
    def validate_memory(self, attribute: Attribute, value:␣
↪Optional[BaseConversationMemory]) -> None:
        if value is None:
            raise ValueError("Conversation memory must not be None.")

    def lines(self) -> list[str]:
        from griptape.memory.structure import SummaryConversationMemory

        lines = []

        for run in self.memory.runs if self.memory is not None else []:
            lines.extend((f"Q: {run.input}", f"A: {run.output}"))

        if isinstance(self.memory, SummaryConversationMemory):
            lines.append(f"Summary: {self.memory.summary}")

        return lines

    def prompt_stack(self) -> list[str]:
        from griptape.memory.structure import SummaryConversationMemory

        lines = []

        for stack in self.memory.to_prompt_stack().messages if self.memory is not␣
↪None else []:
            lines.append(f"{stack.role}: {stack.to_text()}")

        if isinstance(self.memory, SummaryConversationMemory):
            lines.append(f"Summary: {self.memory.summary}")

        return lines

    def __str__(self) -> str:
        return str.join("\n", self.lines())
```

- `memory = field()` class-attribute instance-attribute

str()

Source Code in griptape/utils/conversation.py

```
def __str__(self) -> str:
    return str.join("\n", self.lines())
```

lines()

Source Code in griptape/utils/conversation.py

```
def lines(self) -> list[str]:
    from griptape.memory.structure import SummaryConversationMemory

    lines = []

    for run in self.memory.runs if self.memory is not None else []:
        lines.extend((f"Q: {run.input}", f"A: {run.output}"))

    if isinstance(self.memory, SummaryConversationMemory):
        lines.append(f"Summary: {self.memory.summary}")

    return lines
```

prompt_stack()

Source Code in griptape/utils/conversation.py

```
def prompt_stack(self) -> list[str]:
    from griptape.memory.structure import SummaryConversationMemory

    lines = []

    for stack in self.memory.to_prompt_stack().messages if self.memory is not None␣
↪else []:
        lines.append(f"{stack.role}: {stack.to_text()}")

    if isinstance(self.memory, SummaryConversationMemory):
        lines.append(f"Summary: {self.memory.summary}")

    return lines
```

validate_memory(attribute, value)

Source Code in griptape/utils/conversation.py

```
@memory.validator  # pyright: ignore[reportAttributeAccessIssue,␣
↪reportOptionalMemberAccess]
def validate_memory(self, attribute: Attribute, value:␣
↪Optional[BaseConversationMemory]) -> None:
    if value is None:
        raise ValueError("Conversation memory must not be None.")
```

decorators

- `CONFIG_SCHEMA = Schema({schema.Optional('name'): str, 'description': str, schema.Optional('schema'): lambda data: isinstance(data, (Schema, Callable)) or isinstance(data, type) and issubclass(data, BaseModel)})` module-attribute
- `P = ParamSpec('P')` module-attribute

- `R = TypeVar('R')` module-attribute

Source Code in griptape/utils/decorators.py

```
def _build_kwargs(func: Callable, params: dict) -> dict:
    func_params = cast("OrderedDict", inspect.signature(func).parameters.copy())
    func_params.popitem(last=False)

    kwarg_var = None
    for param in func_params.values():
        # if there is a **kwargs parameter, we can safely
        # pass all the params to the function
        if param.kind == inspect.Parameter.VAR_KEYWORD:
            kwarg_var = func_params.pop(param.name).name
            break

    # only pass the values that are in the function signature
    # or if there is a **kwargs parameter, pass all the values
    kwargs = {k: v for k, v in params.get("values", {}).items() if k in func_params␣
↪or kwarg_var is not None}

    # add params and values if they are in the signature
    # or if there is a **kwargs parameter
    if "params" in func_params or kwarg_var is not None:
        kwargs["params"] = params
    if "values" in func_params or kwarg_var is not None:
        kwargs["values"] = params.get("values")

    # set any missing parameters to None
    for param_name, param in func_params.items():
        if param_name not in kwargs and param.default == inspect.Parameter.empty:
            kwargs[param_name] = None

    return kwargs
```

activity(config)

Source Code in griptape/utils/decorators.py

```
def activity(config: dict) -> Any:
    validated_config = CONFIG_SCHEMA.validate(config)

    validated_config.update({k: v for k, v in config.items() if k not in validated_
↪config})

    if not validated_config.get("schema"):
        validated_config["schema"] = None

    def decorator(func: Callable) -> Any:
        @functools.wraps(func)
        def wrapper(self: Any, params: dict) -> Any:
            return func(self, **_build_kwargs(func, params))

        setattr(wrapper, "name", validated_config.get("name", func.__name__))
```

(continues on next page)

(continued from previous page)

```
        setattr(wrapper, "config", validated_config)
        setattr(wrapper, "is_activity", True)

        return wrapper

    return decorator
```

lazy_property(attr_name=None)

Source Code in griptape/utils/decorators.py

```
def lazy_property(attr_name: Optional[str] = None) -> Callable[[Callable[P, R]], R]:
    def decorator(func: Callable[P, R]) -> R:
        actual_attr_name = f"_{func.__name__}" if attr_name is None else attr_name

        @property
        @functools.wraps(func)
        def wrapper(self: Any) -> R:
            if getattr(self, actual_attr_name) is None:
                setattr(self, actual_attr_name, func(self))  # pyright:␣
↪ignore[reportCallIssue]
            return getattr(self, actual_attr_name)

        @wrapper.setter
        def wrapper(self: Any, value: Any) -> None:
            setattr(self, actual_attr_name, value)

        return cast("R", wrapper)

    return decorator
```

deprecation

Bases:

ModuleType

Source Code in griptape/utils/deprecation.py

```
class DeprecationModuleWrapper(ModuleType):
    """Module wrapper that issues a deprecation warning when importing."""

    __ignore_attrs__ = {
        "__file__",
        "__package__",
        "__path__",
        "__doc__",
        "__all__",
        "__name__",
        "__loader__",
        "__spec__",
    }
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, real_module: Any, deprecation_message: str) -> None:
        self._real_module = real_module
        self._deprecation_message = deprecation_message

    def __getattr__(self, name: str) -> Any:
        if name not in self.__ignore_attrs__:
            warnings.warn(
                self._deprecation_message,
                DeprecationWarning,
                stacklevel=2,
            )
        return getattr(self._real_module, name)
```

- `__ignore_attrs__ = {'__file__', '__package__', '__path__', '__doc__', '__all__', '__name__', '__loader__', '__spec__'}` class-attribute instance-attribute
- `_deprecation_message = deprecation_message` instance-attribute
- `_real_module = real_module` instance-attribute

getattr(name)

Source Code in griptape/utils/deprecation.py

```
def __getattr__(self, name: str) -> Any:
    if name not in self.__ignore_attrs__:
        warnings.warn(
            self._deprecation_message,
            DeprecationWarning,
            stacklevel=2,
        )
    return getattr(self._real_module, name)
```

init(real_module, deprecation_message)

Source Code in griptape/utils/deprecation.py

```
def __init__(self, real_module: Any, deprecation_message: str) -> None:
    self._real_module = real_module
    self._deprecation_message = deprecation_message
```

deprecation_warn(message, stacklevel=2)

Source Code in griptape/utils/deprecation.py

```
def deprecation_warn(message: str, stacklevel: int = 2) -> None:
    warnings.simplefilter("always", DeprecationWarning)
    warnings.warn(message, category=DeprecationWarning, stacklevel=stacklevel)
    warnings.simplefilter("default", DeprecationWarning)
```

dict_utils

Add a key in a dictionary recursively.

Parameters

Name	Type	Description	Default
d	*Any*	The dictionary to add the key to. 	*required*
key	*str*	The key to add. 	*required*
value	*Any*	The value to add. 	*required*
criteria	*Optional[Callable[[dict], bool]]*	An optional function to determine if the key should be added. 	*None*

Source Code in griptape/utils/dict_utils.py

```
def add_key_in_dict_recursively(
    d: Any, key: str, value: Any, criteria: Optional[Callable[[dict], bool]] = None
) -> dict:
    """Add a key in a dictionary recursively.

    Args:
        d: The dictionary to add the key to.
        key: The key to add.
        value: The value to add.
        criteria: An optional function to determine if the key should be added.
    """
    if isinstance(d, dict):
        if criteria is None or criteria(d):
            d[key] = value
        return {k: add_key_in_dict_recursively(v, key, value, criteria) for k, v in d.
↪items()}
    return d
```

dict_merge(dct, merge_dct, *, add_keys=True)

Recursive dict merge.

Inspired by :meth:`dict.update()` , instead of updating only top-level keys, dict_merge recurses down into dicts nested to an arbitrary depth, updating keys. The `merge_dct` is merged into `dct` .

This version will return a copy of the dictionary and leave the original arguments untouched.

The optional argument `add_keys` , determines whether keys which are present in `merge_dict` but not `dct` should be included in the new dict.

Parameters

Name	Type	Description	Default
dct	*Optional[dict]*	onto which the merge is executed 	*required*
merge_dct	*Optional[dict]*	dct merged into dct 	*required*
add_keys	*bool*	whether to add new keys 	*True*

Returns

Name	Type	Description
dict	*dict*	updated dict

Source Code in griptape/utils/dict_utils.py

```
def dict_merge(dct: Optional[dict], merge_dct: Optional[dict], *, add_keys: bool =␣
↪True) -> dict:
    """Recursive dict merge.

    Inspired by :meth:``dict.update()``, instead of
    updating only top-level keys, dict_merge recurses down into dicts nested
    to an arbitrary depth, updating keys. The ``merge_dct`` is merged into
    ``dct``.

    This version will return a copy of the dictionary and leave the original
    arguments untouched.

    The optional argument ``add_keys``, determines whether keys which are
    present in ``merge_dict`` but not ``dct`` should be included in the
    new dict.

    Args:
        dct: onto which the merge is executed
        merge_dct: dct merged into dct
        add_keys: whether to add new keys

    Returns:
        dict: updated dict
    """
    dct = {} if dct is None else dct
    merge_dct = {} if merge_dct is None else merge_dct

    dct = dct.copy()

    if not add_keys:
        merge_dct = {k: merge_dct[k] for k in set(dct).intersection(set(merge_dct))}
```

(continues on next page)

(continued from previous page)

```
    for key in merge_dct:
        if key in dct and isinstance(dct[key], dict):
            dct[key] = dict_merge(dct[key], merge_dct[key], add_keys=add_keys)
        else:
            dct[key] = merge_dct[key]

    return dct
```

remove_key_in_dict_recursively(d, key)

Source Code in griptape/utils/dict_utils.py

```
def remove_key_in_dict_recursively(d: dict, key: str) -> dict:
    if isinstance(d, dict):
        return {k: remove_key_in_dict_recursively(v, key) for k, v in d.items() if k !
↪= key}
    return d
```

remove_null_values_in_dict_recursively(d)

Source Code in griptape/utils/dict_utils.py

```
def remove_null_values_in_dict_recursively(d: dict) -> dict:
    if isinstance(d, dict):
        return {k: remove_null_values_in_dict_recursively(v) for k, v in d.items() if␣
↪v is not None}
    return d
```

file_utils

Source Code in griptape/utils/file_utils.py

```
def _is_csv(data: bytes) -> bool:
    """Check if data appears to be CSV-like."""
    try:
        text = data.decode("utf-8")
        sample = io.StringIO(text).read(1024)
        dialect = csv.Sniffer().sniff(sample, delimiters=";,\t|")
        return bool(dialect)
    except (UnicodeDecodeError, csv.Error):
        return False
```

_is_json(data)

Source Code in griptape/utils/file_utils.py

```
def _is_json(data: bytes) -> bool:
    """Check if data is valid JSON."""
    try:
        json.loads(data.decode("utf-8"))
        return True
```

(continues on next page)

(continued from previous page)

```
    except (UnicodeDecodeError, json.JSONDecodeError):
        return False
```

_is_text(data)

Source Code in griptape/utils/file_utils.py

```
def _is_text(data: bytes) -> bool:
    """Check if bytes are decodable as text.

    Required since filetypes does not support this: https://github.com/h2non/filetype.
↪py/issues/30

    """
    try:
        text = data.decode("utf-8")
        return all(c.isprintable() or c.isspace() for c in text)
    except UnicodeDecodeError:
        return False
```

get_mime_type(file_path_or_bytes)

Attempt to determine the MIME type of a file or bytes.

If the input is a file path, we use the built-in `mimetypes` package to guess the MIME type.

If the input is bytes, we use the `filetype` library to determine the MIME type. If the library cannot determine the MIME type (data missing magic bytes), we use a few heuristics to guess the type.

Parameters

Name	Type	Description	Default
file_path_or_bytes	*str I bytes*	The path to the file or the bytes to check. 	*required*

Source Code in griptape/utils/file_utils.py

```
def get_mime_type(file_path_or_bytes: str | bytes) -> str:
    """Attempt to determine the MIME type of a file or bytes.

    If the input is a file path, we use the built-in `mimetypes` package to guess the␣
↪MIME type.

    If the input is bytes, we use the `filetype` library to determine the MIME type.
    If the library cannot determine the MIME type (data missing magic bytes), we use␣
↪a few heuristics to guess the type.

    Args:
        file_path_or_bytes: The path to the file or the bytes to check.
```

(continues on next page)

(continued from previous page)

```
    Returns: The MIME type of the file or bytes.
    """
    if isinstance(file_path_or_bytes, bytes):
        filetype_guess = filetype.guess(file_path_or_bytes)

        if filetype_guess is None:
            if _is_text(file_path_or_bytes):
                if _is_json(file_path_or_bytes):
                    return "application/json"
                if _is_csv(file_path_or_bytes):
                    return "text/csv"
                return "text/plain"
            return "application/octet-stream"
        return filetype_guess.mime
    type_, _ = mimetypes.guess_type(file_path_or_bytes)
    if type_ is None:
        return "application/octet-stream"
    return type_
```

futures

- `T = TypeVar('T')` module-attribute

Source Code in griptape/utils/futures.py

```
def execute_futures_dict(fs_dict: dict[str, futures.Future[T]]) -> dict[str, T]:
    futures.wait(fs_dict.values(), timeout=None, return_when=futures.ALL_COMPLETED)

    return {key: future.result() for key, future in fs_dict.items()}
```

execute_futures_list(fs_list)

Source Code in griptape/utils/futures.py

```
def execute_futures_list(fs_list: list[futures.Future[T]]) -> list[T]:
    futures.wait(fs_list, timeout=None, return_when=futures.ALL_COMPLETED)

    return [future.result() for future in fs_list]
```

execute_futures_list_dict(fs_dict)

Source Code in griptape/utils/futures.py

```
def execute_futures_list_dict(fs_dict: dict[str, list[futures.Future[T]]]) ->␣
↪dict[str, list[T]]:
    execute_futures_list([item for sublist in fs_dict.values() for item in sublist])

    return {key: [f.result() for f in fs] for key, fs in fs_dict.items()}
```

griptape_cloud

- `P = ParamSpec('P')` module-attribute
- `T = TypeVar('T')` module-attribute

Utility for working with Gen AI Builder Structures.

Attributes

Name	Type	Description
<a href="griptape_cloud.mdx#griptape.utils.griptape_cloud.GriptapeCloudStructure._event_listener" title="_event_listener = field(default=None, kw_only=True, alias='event_listener') class-attribute instance-attribute (griptape.utils.griptape_cloud.GriptapeCloudStructure._event_listener)"><code class="language-python">	<a href="../events#griptape.events.EventListener" title="EventListener (griptape.events.EventListener)"><code class="language-python">Optional[EventListener]</cc	Event Listener to use. Defaults to an EventListener with a GriptapeCloudEventListenerDriver.
<a href="griptape_cloud.mdx#griptape.utils.griptape_cloud.GriptapeCloudStructure._observability" title="_observability = field(default=None, kw_only=True, alias='observability') class-attribute instance-attribute (griptape.utils.griptape_cloud.GriptapeCloudStructure._observability)"><code class="language-python">	<a href="../observability/observability.mdx#griptape.observability.observability.Observability" title="Observability (griptape.observability.observability.Observability)"><code class="language-python">Optional[Observability]</co	Observability to use. Defaults to an Observability with a GriptapeCloudObservabilityDriver.

continues on next page

Table 435 – continued from previous page

Name	Type	Description
<a href="griptape_cloud.mdx#griptape.utils.griptape_cloud.GriptapeCloudStructure.observe" title="observe = field(default=False, kw_only=True) class-attribute instance-attribute (griptape.utils.griptape_cloud.GriptapeCloudStructure.observe)"><code class="language-python">	*bool*	Whether to enable observability. Enabling requires the *drivers-observability-griptape-cloud* extra.

Source Code in griptape/utils/griptape_cloud.py

```
@define()
class GriptapeCloudStructure:
    """Utility for working with Gen AI Builder Structures.

    Attributes:
        _event_listener: Event Listener to use. Defaults to an EventListener with a␣
↪GriptapeCloudEventListenerDriver.
        _observability: Observability to use. Defaults to an Observability with a␣
↪GriptapeCloudObservabilityDriver.
        observe: Whether to enable observability. Enabling requires the `drivers-
↪observability-griptape-cloud` extra.
    """

    _event_listener: Optional[EventListener] = field(default=None, kw_only=True,␣
↪alias="event_listener")
    _observability: Optional[Observability] = field(default=None, kw_only=True, alias=
↪"observability")
    observe: bool = field(default=False, kw_only=True)
    _output: Optional[BaseArtifact] = field(default=None, init=False)

    @lazy_property()
    def event_listener(self) -> EventListener:
        from griptape.drivers.event_listener.griptape_cloud import␣
↪GriptapeCloudEventListenerDriver

        return EventListener(event_listener_driver=GriptapeCloudEventListenerDriver())

    @lazy_property()
    def observability(self) -> Observability:
        from griptape.drivers.observability.griptape_cloud import␣
↪GriptapeCloudObservabilityDriver
        from griptape.observability.observability import Observability
```

(continues on next page)

(continued from previous page)

```
        return Observability(observability_driver=GriptapeCloudObservabilityDriver())

    @property
    def output(self) -> Optional[BaseArtifact]:
        return self._output

    @output.setter
    def output(self, value: BaseArtifact | Any) -> None:
        if isinstance(value, BaseArtifact):
            self._output = value
        elif isinstance(value, list):
            self._output = ListArtifact([self._to_artifact(item) for item in value])
        else:
            self._output = self._to_artifact(value)

    @property
    def structure_run_id(self) -> str:
        return os.environ["GT_CLOUD_STRUCTURE_RUN_ID"]

    @property
    def in_managed_environment(self) -> bool:
        return "GT_CLOUD_STRUCTURE_RUN_ID" in os.environ

    def __enter__(self) -> Self:
        from griptape.observability.observability import Observability

        if self.in_managed_environment:
            EventBus.add_event_listener(self.event_listener)

            if self.observe:
                Observability.set_global_driver(self.observability.observability_
↪driver)
                self.observability.observability_driver.__enter__()

        return self

    def __exit__(
        self,
        exc_type: Optional[type[BaseException]],
        exc_value: Optional[BaseException],
        exc_traceback: Optional[TracebackType],
    ) -> None:
        from griptape.observability.observability import Observability

        if self.in_managed_environment:
            if self.output is not None:
                EventBus.publish_event(FinishStructureRunEvent(output_task_
↪output=self.output), flush=True)
            EventBus.remove_event_listener(self.event_listener)

            if self.observe:
```

(continues on next page)

(continued from previous page)

```
                Observability.set_global_driver(None)
                self.observability.observability_driver.__exit__(exc_type, exc_value,␣
↪exc_traceback)

    def _to_artifact(self, value: Any) -> BaseArtifact:
        if isinstance(value, str):
            return TextArtifact(value)
        if isinstance(value, bool):
            return BooleanArtifact(value)
        if isinstance(value, dict):
            return JsonArtifact(value)
        if isinstance(value, bytes):
            return BlobArtifact(value)
        return GenericArtifact(value)
```

- `_event_listener = field(default=None, kw_only=True, alias='event_listener')` class-attribute instance-attribute
- `_observability = field(default=None, kw_only=True, alias='observability')` class-attribute instance-attribute
- `_output = field(default=None, init=False)` class-attribute instance-attribute
- `in_managed_environment` property
- `observe = field(default=False, kw_only=True)` class-attribute instance-attribute
- `output` property writable
- `structure_run_id` property

enter()

Source Code in griptape/utils/griptape_cloud.py

```
def __enter__(self) -> Self:
    from griptape.observability.observability import Observability

    if self.in_managed_environment:
        EventBus.add_event_listener(self.event_listener)

        if self.observe:
            Observability.set_global_driver(self.observability.observability_driver)
            self.observability.observability_driver.__enter__()

    return self
```

exit(exc_type, exc_value, exc_traceback)

Source Code in griptape/utils/griptape_cloud.py

```
def __exit__(
    self,
    exc_type: Optional[type[BaseException]],
    exc_value: Optional[BaseException],
```

(continues on next page)

(continued from previous page)

```
    exc_traceback: Optional[TracebackType],
) -> None:
    from griptape.observability.observability import Observability

    if self.in_managed_environment:
        if self.output is not None:
            EventBus.publish_event(FinishStructureRunEvent(output_task_output=self.
↪output), flush=True)
        EventBus.remove_event_listener(self.event_listener)

        if self.observe:
            Observability.set_global_driver(None)
            self.observability.observability_driver.__exit__(exc_type, exc_value, exc_
↪traceback)
```

_to_artifact(value)

Source Code in griptape/utils/griptape_cloud.py

```
def _to_artifact(self, value: Any) -> BaseArtifact:
    if isinstance(value, str):
        return TextArtifact(value)
    if isinstance(value, bool):
        return BooleanArtifact(value)
    if isinstance(value, dict):
        return JsonArtifact(value)
    if isinstance(value, bytes):
        return BlobArtifact(value)
    return GenericArtifact(value)
```

event_listener()

Source Code in griptape/utils/griptape_cloud.py

```
@lazy_property()
def event_listener(self) -> EventListener:
    from griptape.drivers.event_listener.griptape_cloud import␣
↪GriptapeCloudEventListenerDriver

    return EventListener(event_listener_driver=GriptapeCloudEventListenerDriver())
```

observability()

Source Code in griptape/utils/griptape_cloud.py

```
@lazy_property()
def observability(self) -> Observability:
    from griptape.drivers.observability.griptape_cloud import␣
↪GriptapeCloudObservabilityDriver
    from griptape.observability.observability import Observability
```

(continues on next page)

(continued from previous page)

```
    return Observability(observability_driver=GriptapeCloudObservabilityDriver())
```

griptape_cloud_url(base_url, route)

Source Code in griptape/utils/griptape_cloud.py

```
def griptape_cloud_url(base_url: str, route: str) -> str:
    if not base_url.endswith("/"):
        base_url += "/"
    return urljoin(base_url, route.lstrip("/"))
```

hash

Source Code in griptape/utils/hash.py

```
def bytes_to_hash(data: bytes, hash_algorithm: str = "sha256") -> str:
    m = hashlib.new(hash_algorithm)

    m.update(data)

    return m.hexdigest()
```

str_to_hash(text, hash_algorithm='sha256')

Source Code in griptape/utils/hash.py

```
def str_to_hash(text: str, hash_algorithm: str = "sha256") -> str:
    m = hashlib.new(hash_algorithm)

    m.update(text.encode())

    return m.hexdigest()
```

import_utils

- `INSTALL_MAPPING = {'huggingface_hub': 'huggingface-hub', 'pinecone': 'pinecone-client', 'opensearchpy': 'opensearch-py', 'google.generativeai': 'google-generativeai'}` module-attribute

Import an optional dependency.

If a dependency is missing, an ImportError with a nice message will be raised.

Parameters

Name	Type	Description	Default
name	*str*	The module name. 	*required*

Returns

Type	Description
ModuleType	The imported module, when found.

Source Code in griptape/utils/import_utils.py

```
def import_optional_dependency(name: str) -> ModuleType:
    """Import an optional dependency.

    If a dependency is missing, an ImportError with a nice message will be raised.

    Args:
        name: The module name.

    Returns:
        The imported module, when found.
    """
    package_name = INSTALL_MAPPING.get(name)
    install_name = package_name if package_name is not None else name

    msg = (
        f"Missing optional dependency: {install_name}. "
        f"Please install the appropriate extra: https://docs.griptape.ai/stable/
↪griptape-framework/#extras."
    )
    try:
        module = import_module(name)
    except ImportError as exc:
        raise ImportError(msg) from exc

    return module
```

is_dependency_installed(name)

Check if an optional dependency is available.

Parameters

Name	Type	Description	Default
name	*str*	The module name. 	*required*

Returns

Type	Description
bool	True if the dependency is available.

continues on next page

Table 439 – continued from previous page

Type	Description
bool	False if the dependency is not available.

Source Code in griptape/utils/import_utils.py

```
def is_dependency_installed(name: str) -> bool:
    """Check if an optional dependency is available.

    Args:
        name: The module name.

    Returns:
        True if the dependency is available.
        False if the dependency is not available.
    """
    try:
        import_optional_dependency(name)
    except ImportError:
        return False

    return True
```

j2

Source Code in griptape/utils/j2.py

```
@define(frozen=True)
class J2:
    template_name: Optional[str] = field(default=None)
    templates_dir: str = field(default=abs_path("templates"), kw_only=True)
    environment: Environment = field(
        default=Factory(
            lambda self: Environment(loader=FileSystemLoader(self.templates_dir),␣
↪trim_blocks=True, lstrip_blocks=True),
            takes_self=True,
        ),
        kw_only=True,
    )

    def render(self, **kwargs) -> str:
        if self.template_name is None:
            raise ValueError("template_name is required.")
        return self.environment.get_template(self.template_name).render(kwargs).
↪rstrip()

    def render_from_string(self, value: str, **kwargs) -> str:
        return self.environment.from_string(value).render(kwargs)
```

- `environment = field(default=Factory(lambda self: Environment(loader=FileSystemLoader(self.templates_dir), trim_blocks=True, lstrip_blocks=True), takes_self=True), kw_only=True)` class-attribute instance-attribute

- `template_name = field(default=None)` class-attribute instance-attribute
- `templates_dir = field(default=abs_path('templates'), kw_only=True)` class-attribute instance-attribute

render(**kwargs)

Source Code in griptape/utils/j2.py

```
def render(self, **kwargs) -> str:
    if self.template_name is None:
        raise ValueError("template_name is required.")
    return self.environment.get_template(self.template_name).render(kwargs).rstrip()
```

render_from_string(value, **kwargs)

Source Code in griptape/utils/j2.py

```
def render_from_string(self, value: str, **kwargs) -> str:
    return self.environment.from_string(value).render(kwargs)
```

json_schema_utils

Performs a series of post-processing steps to ensure a JSON schema is compatible with LLMs.

1. Adds the `$id` and `$schema` keys.
2. Sets `additionalProperties` to `False` for objects without this key.
3. Resolves `$ref` s and removes `$defs` .

Parameters

Name	Type	Description	Default
json_schema	*dict*	The JSON schema to ensure is strict. 	*required*
schema_id	*str*	The ID of the schema. 	*required*

Source Code in griptape/utils/json_schema_utils.py

```
def build_strict_schema(json_schema: dict, schema_id: str) -> dict:
    """Performs a series of post-processing steps to ensure a JSON schema is␣
↪compatible with LLMs.

    1. Adds the `$id` and `$schema` keys.
    2. Sets `additionalProperties` to `False` for objects without this key.
    3. Resolves `$ref`s and removes `$defs`.

    Args:
        json_schema: The JSON schema to ensure is strict.
        schema_id: The ID of the schema.
```

(continues on next page)

(continued from previous page)

```
    Returns: The strict JSON schema.
    """
    from griptape.utils.dict_utils import add_key_in_dict_recursively

    json_schema.setdefault("$id", schema_id)
    json_schema.setdefault("$schema", "http://json-schema.org/draft-07/schema#")
    json_schema = add_key_in_dict_recursively(
        json_schema,
        key="additionalProperties",
        value=False,
        criteria=lambda d: d.get("type") == "object" and "additionalProperties" not
↪in d,
    )
    json_schema = resolve_refs(json_schema)
    if "$defs" in json_schema:
        del json_schema["$defs"]

    return json_schema
```

resolve_refs(schema)

Recursively resolve all local $refs in the given JSON Schema using $defs as the source.

Required since pydantic does not support nested schemas without $refs. https://github.com/pydantic/pydantic/issues/889

Parameters

Name	Type	Description	Default
schema	*dict[str, Any]*	A JSON Schema as a dictionary, which may contain "$refs" and "$defs". 	*required*

Source Code in griptape/utils/json_schema_utils.py

```
def resolve_refs(schema: dict[str, Any]) -> dict[str, Any]:
    """Recursively resolve all local $refs in the given JSON Schema using $defs as
↪the source.

    Required since pydantic does not support nested schemas without $refs. https://
↪github.com/pydantic/pydantic/issues/889

    Args:
        schema: A JSON Schema as a dictionary, which may contain "$refs" and "$defs".
    Returns:  A new dictionary with all local $refs resolved against $defs.
    """
    defs = schema.get("$defs", {})
    # Work on a deep copy so we dont mutate the original schema.
    schema_copy = deepcopy(schema)
```

(continues on next page)

(continued from previous page)

```
    def _resolve(node: Any) -> Any:
        """Recursively walk through the node, resolving any local $refs to the␣
↪corresponding definitions in defs."""
        if isinstance(node, dict):
            # If this node is a reference to something in #/$defs/...
            ref = node.get("$ref")
            if isinstance(ref, str) and ref.startswith("#/$defs/"):
                def_name = ref.replace("#/$defs/", "")
                if def_name in defs:
                    # Replace the entire node with the referenced definition.
                    return _resolve(deepcopy(defs[def_name]))
                raise KeyError(f"Definition {def_name} not found in $defs.")

            # If not a ref, or doesnt start with #/$defs/, just walk deeper.
            return {key: _resolve(value) for key, value in node.items()}

        if isinstance(node, list):
            # Recurse into each item of the list
            return [_resolve(item) for item in node]

        # For scalars (str, int, bool, None, etc.), return them as-is.
        return node

    return _resolve(schema_copy)
```

load_artifact_from_memory

Source Code in griptape/utils/load_artifact_from_memory.py

```
def load_artifact_from_memory(
    memory: Optional[TaskMemory],
    artifact_namespace: str,
    artifact_name: str,
    artifact_type: type,
) -> BaseArtifact:
    if memory is None:
        raise ValueError("memory not found")

    artifacts = memory.load_artifacts(namespace=artifact_namespace)
    if len(artifacts) == 0:
        raise ValueError("no artifacts found in namespace")

    try:
        artifact = [a for a in artifacts if a.name == artifact_name][0]
    except IndexError as exc:
        raise ValueError(f"artifact {artifact_name} not found in namespace {artifact_
↪namespace}") from exc

    if not isinstance(artifact, artifact_type):
        raise ValueError(f"{artifact.name} is not of type {artifact_type}")
```

(continues on next page)

(continued from previous page)

```
    return artifact
```

manifest_validator

Source Code in griptape/utils/manifest_validator.py

```
class ManifestValidator:
    def validate(self, manifest: dict) -> dict:
        return self.schema().validate(manifest)

    def schema(self) -> Schema:
        return Schema({"version": "v1", "name": str, "description": str, "contact_
↪email": str, "legal_info_url": str})
```

schema()

Source Code in griptape/utils/manifest_validator.py

```
def schema(self) -> Schema:
    return Schema({"version": "v1", "name": str, "description": str, "contact_email":␣
↪str, "legal_info_url": str})
```

validate(manifest)

Source Code in griptape/utils/manifest_validator.py

```
def validate(self, manifest: dict) -> dict:
    return self.schema().validate(manifest)
```

paths

Source Code in griptape/utils/paths.py

```
def abs_path(path: str) -> str:
    return os.path.abspath(os.path.join(os.path.dirname(os.path.abspath(__file__)),␣
↪os.pardir, path))
```

python_runner

Source Code in griptape/utils/python_runner.py

```
@define
class PythonRunner:
    libs: dict[str, str] = field(factory=dict, kw_only=True)

    def run(self, code: str) -> str:
        global_stdout = sys.stdout
        sys.stdout = local_stdout = StringIO()

        try:
```

(continues on next page)

(continued from previous page)

```
            for lib, alias in self.libs.items():
                imported_lib = importlib.import_module(lib)
                globals()[alias] = imported_lib

            exec(f"print({code})", {}, {alias: eval(alias) for alias in self.libs.
↪values()})

            output = local_stdout.getvalue()
        except Exception as e:
            output = str(e)
        finally:
            sys.stdout = global_stdout

        return output.strip()
```

- `libs = field(factory=dict, kw_only=True)` class-attribute instance-attribute

run(code)

Source Code in griptape/utils/python_runner.py

```
def run(self, code: str) -> str:
    global_stdout = sys.stdout
    sys.stdout = local_stdout = StringIO()

    try:
        for lib, alias in self.libs.items():
            imported_lib = importlib.import_module(lib)
            globals()[alias] = imported_lib

        exec(f"print({code})", {}, {alias: eval(alias) for alias in self.libs.
↪values()})

        output = local_stdout.getvalue()
    except Exception as e:
        output = str(e)
    finally:
        sys.stdout = global_stdout

    return output.strip()
```

reference_utils

Source Code in griptape/utils/reference_utils.py

```
def references_from_artifacts(artifacts: list[TextArtifact]) -> list[Reference]:
    references = []

    for a in artifacts:
        if a.reference is not None and a.reference not in references:
            references.append(a.reference)
```

(continues on next page)

(continued from previous page)

```
    return references
```

stream

A wrapper for Structures filters Events relevant to text output and converts them to TextArtifacts.

Attributes

Name	Type	Description
<code class="language-python">	<code class="language-python">Structure</code>	The Structure to wrap.

Source Code in griptape/utils/stream.py

```
@define
class Stream:
    """A wrapper for Structures filters Events relevant to text output and converts them to TextArtifacts.

    Attributes:
        structure: The Structure to wrap.
    """

    structure: Structure = field()
    event_types: list[type[BaseEvent]] = field(
        default=Factory(lambda: [TextChunkEvent, ActionChunkEvent, FinishPromptEvent, FinishStructureRunEvent])
    )

    def run(self, *args) -> Iterator[TextArtifact]:
        action_str = ""

        for event in self.structure.run_stream(*args, event_types=self.event_types):
            if isinstance(event, FinishPromptEvent):
                yield TextArtifact(value="\n")
            elif isinstance(event, TextChunkEvent):
                yield TextArtifact(value=event.token)
            elif isinstance(event, ActionChunkEvent):
                if event.tag is not None and event.name is not None and event.path is not None:
                    yield TextArtifact(f"{event.name}.{event.tag} ({event.path})")
                if event.partial_input is not None:
```

(continues on next page)

(continued from previous page)

```
                    action_str += event.partial_input
                    try:
                        yield TextArtifact(json.dumps(json.loads(action_str),␣
↪indent=2))
                        action_str = ""
                    except Exception:
                        pass
```

- `event_types = field(default=Factory(lambda: [TextChunkEvent, ActionChunkEvent, FinishPromptEvent, FinishStructureRunEvent]))` class-attribute instance-attribute
- `structure = field()` class-attribute instance-attribute

run(*args)

Source Code in griptape/utils/stream.py

```
def run(self, *args) -> Iterator[TextArtifact]:
    action_str = ""

    for event in self.structure.run_stream(*args, event_types=self.event_types):
        if isinstance(event, FinishPromptEvent):
            yield TextArtifact(value="\n")
        elif isinstance(event, TextChunkEvent):
            yield TextArtifact(value=event.token)
        elif isinstance(event, ActionChunkEvent):
            if event.tag is not None and event.name is not None and event.path is not␣
↪None:
                yield TextArtifact(f"{event.name}.{event.tag} ({event.path})")
            if event.partial_input is not None:
                action_str += event.partial_input
                try:
                    yield TextArtifact(json.dumps(json.loads(action_str), indent=2))
                    action_str = ""
                except Exception:
                    pass
```

structure_visualizer

Source Code in griptape/utils/structure_visualizer.py

```
@define
class StructureVisualizer:
    """Utility class to visualize a Structure structure."""

    structure: Structure = field()
    header: str = field(default="graph TD;", kw_only=True)
    build_node_id: Callable[[BaseTask], str] = field(default=lambda task: task.id.
↪title(), kw_only=True)
    query_params: dict[str, str] = field(factory=dict, kw_only=True)
    base_url: str = field(default="https://mermaid.ink", kw_only=True)
```

(continues on next page)

(continued from previous page)

```
    def to_url(self) -> str:
        """Generates a url that renders the Workflow structure as a Mermaid flowchart.

        Reference: https://mermaid.js.org/ecosystem/tutorials#jupyter-integration-
↪with-mermaid-js.

        Returns:
            str: URL to the rendered image
        """
        self.structure.resolve_relationships()

        tasks = self.__render_tasks(self.structure.tasks)
        graph = f"{self.header}{tasks}"

        graph_bytes = graph.encode("utf-8")
        base64_string = base64.b64encode(graph_bytes).decode("utf-8")

        url = urllib.parse.urljoin(self.base_url, f"svg/{base64_string}")
        req = PreparedRequest()
        req.prepare_url(url, self.query_params)

        if req.url is None:
            raise ValueError("Failed to generate the URL")

        return req.url

    def __render_tasks(self, tasks: list[BaseTask]) -> str:
        return "\n\t" + "\n\t".join([self.__render_task(task) for task in tasks])

    def __render_task(self, task: BaseTask) -> str:
        from griptape.drivers.structure_run.local import LocalStructureRunDriver
        from griptape.tasks import BranchTask, StructureRunTask

        parts = []
        if task.children:
            children = " & ".join([f"{self.build_node_id(child)}" for child in task.
↪children])
            if isinstance(task, BranchTask):
                parts.append(f"{self.build_node_id(task)}{{ {self.build_node_id(task)}
↪ }}-.-> {children};")
            else:
                parts.append(f"{self.build_node_id(task)}--> {children};")
        else:
            parts.append(f"{self.build_node_id(task)};")

        if isinstance(task, StructureRunTask) and isinstance(task.structure_run_
↪driver, LocalStructureRunDriver):
            sub_structure = task.structure_run_driver.create_structure()
            sub_tasks = self.__render_tasks(sub_structure.tasks)
            parts.append(f"subgraph {self.build_node_id(task)}{sub_tasks}\n\tend")

        return "\n\t".join(parts)
```

- `base_url = field(default='https://mermaid.ink', kw_only=True)` class-attribute instance-attribute
- `build_node_id = field(default=lambda task: task.id.title(), kw_only=True)` class-attribute instance-attribute
- `header = field(default='graph TD;', kw_only=True)` class-attribute instance-attribute
- `query_params = field(factory=dict, kw_only=True)` class-attribute instance-attribute
- `structure = field()` class-attribute instance-attribute

__render_task(task)

Source Code in griptape/utils/structure_visualizer.py

```
def __render_task(self, task: BaseTask) -> str:
    from griptape.drivers.structure_run.local import LocalStructureRunDriver
    from griptape.tasks import BranchTask, StructureRunTask

    parts = []
    if task.children:
        children = " & ".join([f"{self.build_node_id(child)}" for child in task.
↪children])
        if isinstance(task, BranchTask):
            parts.append(f"{self.build_node_id(task)}{{ {self.build_node_id(task)} }}-
↪.-> {children};")
        else:
            parts.append(f"{self.build_node_id(task)}--> {children};")
    else:
        parts.append(f"{self.build_node_id(task)};")

    if isinstance(task, StructureRunTask) and isinstance(task.structure_run_driver,␣
↪LocalStructureRunDriver):
        sub_structure = task.structure_run_driver.create_structure()
        sub_tasks = self.__render_tasks(sub_structure.tasks)
        parts.append(f"subgraph {self.build_node_id(task)}{sub_tasks}\n\tend")

    return "\n\t".join(parts)
```

__render_tasks(tasks)

Source Code in griptape/utils/structure_visualizer.py

```
def __render_tasks(self, tasks: list[BaseTask]) -> str:
    return "\n\t" + "\n\t".join([self.__render_task(task) for task in tasks])
```

to_url()

Generates a url that renders the Workflow structure as a Mermaid flowchart.

Reference: https://mermaid.js.org/ecosystem/tutorials#jupyter-integration-with-mermaid-js.

Returns

Name	Type	Description
str	*str*	URL to the rendered image

Source Code in griptape/utils/structure_visualizer.py

```
def to_url(self) -> str:
    """Generates a url that renders the Workflow structure as a Mermaid flowchart.

    Reference: https://mermaid.js.org/ecosystem/tutorials#jupyter-integration-with-
↪mermaid-js.

    Returns:
        str: URL to the rendered image
    """
    self.structure.resolve_relationships()

    tasks = self.__render_tasks(self.structure.tasks)
    graph = f"{self.header}{tasks}"

    graph_bytes = graph.encode("utf-8")
    base64_string = base64.b64encode(graph_bytes).decode("utf-8")

    url = urllib.parse.urljoin(self.base_url, f"svg/{base64_string}")
    req = PreparedRequest()
    req.prepare_url(url, self.query_params)

    if req.url is None:
        raise ValueError("Failed to generate the URL")

    return req.url
```

token_counter

Source Code in griptape/utils/token_counter.py

```
@define
class TokenCounter:
    tokens: int = field(default=0, kw_only=True)

    def add_tokens(self, new_tokens: float) -> int:
        self.tokens += int(new_tokens)

        return self.tokens
```

- `tokens = field(default=0, kw_only=True)` class-attribute instance-attribute

add_tokens(new_tokens)

Source Code in griptape/utils/token_counter.py

```
def add_tokens(self, new_tokens: float) -> int:
    self.tokens += int(new_tokens)

    return self.tokens
```

11.6 Agents

11.6.1 Overview

An *agent* is the quickest way to get started with Gen AI Builder. Agents take in *Tools* and *run(prompt_input)* directly, which the agent uses to add a *Prompt Task* .

You can access the final output of the Agent by using the *Structured Output* attribute.

Agent Tools

```
from griptape.structures import Agent
from griptape.tools import CalculatorTool

agent = Agent(input="Calculate the following: {{ args[0] }}",␣
↪tools=[CalculatorTool()])

agent.run("whats 13^7?")
print("Answer:", agent.output)
```

Agent Input

```
from griptape.structures import Agent

agent = Agent(
    input="Write me a {{ args[0] }} about {{ args[1] }} and {{ args[2] }}",
)

agent.run("Haiku", "Skateboards", "Programming")
```

Configs

Overview

Gen AI Builder exposes a global singleton, *defaults_config* , which can be used to access and modify the default configurations of the framework.

To update the default configurations, simply update the fields on the `Defaults` object. Framework objects will be created with the currently set default configurations, but you can always override at the individual class level.

```
from griptape.configs import Defaults
from griptape.configs.drivers import AnthropicDriversConfig, OpenAiDriversConfig
from griptape.drivers.prompt.anthropic_prompt_driver import AnthropicPromptDriver
```

(continues on next page)

(continued from previous page)

```
from griptape.structures import Agent

Defaults.drivers_config = OpenAiDriversConfig()  # Default
openai_agent = Agent()

Defaults.drivers_config = AnthropicDriversConfig()
anthropic_agent = Agent(
    prompt_driver=AnthropicPromptDriver(model="claude-3-7-sonnet-latest"),  #␣
↪Override the default prompt driver
)
```

Drivers Configs

The *AnthropicDriversConfig* class allows for the customization of Structures within Gen AI Builder, enabling specific settings such as Drivers to be defined for Tasks.

Gen AI Builder provides predefined *AnthropicDriversConfig* 's for widely used services that provide APIs for most Driver types Gen AI Builder offers.

`DriversConfig` s can be used as a Python Context Manager using the `with` statement to temporarily change the default configurations for a block of code.

```
import schema

from griptape.configs.drivers import AnthropicDriversConfig, OpenAiDriversConfig
from griptape.drivers.prompt.anthropic import AnthropicPromptDriver
from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.engines import JsonExtractionEngine
from griptape.structures import Agent
from griptape.tasks import ToolTask
from griptape.tools import ExtractionTool

with OpenAiDriversConfig():  # Agent will be created with OpenAi Drivers
    openai_agent = Agent()

with AnthropicDriversConfig():  # Agent will be created with Anthropic Drivers
    anthropic_agent = Agent(
        tasks=[
            ToolTask(
                "Extract sentiment from this text: {{ args[0] }}",
                prompt_driver=OpenAiChatPromptDriver(model="gpt-4.1"),  # Override␣
↪this particular Tasks prompt driver
                tool=ExtractionTool(
                    extraction_engine=JsonExtractionEngine(
                        prompt_driver=AnthropicPromptDriver(  # Override this␣
↪particular Engines prompt driver
                            model="claude-3-opus-20240229"
                        ),
                        template_schema=schema.Schema({"sentiment": str}).json_schema(
↪"Output"),
                    ),
                ),
            )
```

(continues on next page)

(continued from previous page)

```
        ]
    )

anthropic_agent.run("Hello, I am happy!")
```

OpenAI

The *OpenAI Driver config* provides default Drivers for OpenAI's APIs. This is the default config for all Structures.

```
from griptape.configs import Defaults
from griptape.configs.drivers import OpenAiDriversConfig
from griptape.structures import Agent

Defaults.drivers_config = OpenAiDriversConfig()

agent = Agent()
```

Azure OpenAI

The *Azure OpenAI Driver config* provides default Drivers for Azure's OpenAI APIs.

```
import os

from griptape.configs import Defaults
from griptape.configs.drivers import AzureOpenAiDriversConfig
from griptape.structures import Agent

Defaults.drivers_config = AzureOpenAiDriversConfig(
    azure_endpoint=os.environ["AZURE_OPENAI_ENDPOINT_1"], api_key=os.environ["AZURE_
↪OPENAI_API_KEY_1"]
)

agent = Agent()
```

Amazon Bedrock

The *Amazon Bedrock Driver config* provides default Drivers for Amazon Bedrock's APIs.

```
import os

import boto3

from griptape.configs import Defaults
from griptape.configs.drivers import AmazonBedrockDriversConfig
from griptape.structures import Agent

Defaults.drivers_config = AmazonBedrockDriversConfig(
    session=boto3.Session(
        region_name=os.environ["AWS_DEFAULT_REGION"],
        aws_access_key_id=os.environ["AWS_ACCESS_KEY_ID"],
        aws_secret_access_key=os.environ["AWS_SECRET_ACCESS_KEY"],
```

(continues on next page)

(continued from previous page)

```
    )
)

agent = Agent()
```

Google

The *Google Driver config* provides default Drivers for Google's Gemini APIs.

```
from griptape.configs import Defaults
from griptape.configs.drivers import GoogleDriversConfig
from griptape.structures import Agent

Defaults.drivers_config = GoogleDriversConfig()

agent = Agent()
```

Anthropic

The *Anthropic Driver config* provides default Drivers for Anthropic's APIs.

```
from griptape.configs import Defaults
from griptape.configs.drivers import AnthropicDriversConfig
from griptape.structures import Agent

Defaults.drivers_config = AnthropicDriversConfig()

agent = Agent()
```

Cohere

The *Cohere Driver config* provides default Drivers for Cohere's APIs.

```
import os

from griptape.configs import Defaults
from griptape.configs.drivers import CohereDriversConfig
from griptape.structures import Agent

Defaults.drivers_config = CohereDriversConfig(api_key=os.environ["COHERE_API_KEY"])

agent = Agent()
```

Custom

You can create your own *AnthropicDriversConfig* by overriding relevant Drivers. The *AnthropicDriversConfig* class includes "Dummy" Drivers for all types, which throw a *DummyError* if invoked without being overridden. This approach ensures that you are informed through clear error messages if you attempt to use Structures without proper Driver configurations.

```
import os

from griptape.configs import Defaults
from griptape.configs.drivers import DriversConfig
from griptape.drivers.embedding.openai import OpenAiEmbeddingDriver
from griptape.drivers.prompt.anthropic import AnthropicPromptDriver
from griptape.structures import Agent

Defaults.drivers_config = DriversConfig(
    prompt_driver=AnthropicPromptDriver(
        model="claude-3-sonnet-20240229",
        api_key=os.environ["ANTHROPIC_API_KEY"],
    ),
    embedding_driver=OpenAiEmbeddingDriver(),
)

agent = Agent()
```

Logging Config

Gen AI Builder provides a predefined *LoggingConfig* 's for easily customizing the logging events that the framework emits. In order to customize the logger, the logger can be fetched by using the `Defaults.logging.logger_name`.

```
import logging

from griptape.configs import Defaults
from griptape.configs.drivers import OpenAiDriversConfig
from griptape.configs.logging import TruncateLoggingFilter
from griptape.structures import Agent

Defaults.drivers_config = OpenAiDriversConfig()

logger = logging.getLogger(Defaults.logging_config.logger_name)
logger.setLevel(logging.ERROR)
logger.addFilter(TruncateLoggingFilter(max_log_length=100))

agent = Agent()
```

Debug Logs

You can enable debug logs to view more granular information such as request/response payloads.

```
import logging

from griptape.configs import Defaults
from griptape.configs.logging import JsonFormatter
from griptape.structures import Agent

logger = logging.getLogger(Defaults.logging_config.logger_name)
logger.setLevel(logging.DEBUG)
logger.handlers[0].setFormatter(JsonFormatter())
```

(continues on next page)

(continued from previous page)

```
agent = Agent()

agent.run("Hello world!")
```

Loading/Saving Configs

You can serialize and deserialize Driver Configs using the *actions_to_json()* and *from_json()* methods.

```
from pathlib import Path

from griptape.configs import Defaults
from griptape.configs.drivers import DriversConfig
from griptape.structures import Agent

config_file = "config.json"

#  Save config

config_text = Defaults.drivers_config.to_json()
Path(config_file).write_text(config_text)

#  Load config

config_text = Path(config_file).read_text()
Defaults.drivers_config = DriversConfig.from_json(config_text)

agent = Agent()
```

Conversation Memory

Overview

You can use Conversation Memory to give Gen AI Builder Structures the ability to keep track of the conversation across runs. All structures are created with *AmazonDynamoDbConversationMemoryDriver* by default.

Example

```
from griptape.structures import Agent

agent = Agent()

agent.run("My favorite animal is a Liger.")
agent.run("What is my favorite animal?")
```

You can disable conversation memory in any structure by setting it to `None` :

```
from griptape.structures import Agent

Agent(conversation_memory=None)
```

Interaction With Structures

Per Structure

By default, Conversation Memory *store(runs, metadata)* are created for each run of the structure. Gen AI Builder takes the Structure's *base_audio_input_task* 's input and the *output_task* 's output, storing them in the Run. Tasks that are neither the input task nor the output task are not stored in the Run.

```
from griptape.structures import Pipeline
from griptape.tasks import PromptTask

pipeline = Pipeline(
    tasks=[
        PromptTask("Respond to this request: {{ args[0] }}", id="input"),
        PromptTask("Improve the writing of this: {{ parent_output }}", id="improve"),
        PromptTask("Output this as a pirate: {{ parent_output }}", id="output"),
    ]
)

pipeline.run("My favorite animal is a Liger.")

if pipeline.conversation_memory is not None:
    for run in pipeline.conversation_memory.runs:
        print("Input", run.input.value)
        print("Output", run.output.value)
        print("\n\n")
```

In this example, the `improve` Task is "forgotten" after the Structure's run is finished. This approach allows you to perform intermediary work within a Structure without it being stored in, and potentially cluttering, Conversation Memory.

Per Task

You can change when Conversation Memory Runs are created by modifying *Structure.conversation_memory_strategy* from the default `per_structure` to `per_task` .

```
from griptape.structures import Pipeline
from griptape.tasks import PromptTask

pipeline = Pipeline(
    conversation_memory_strategy="per_task",
    tasks=[
        PromptTask("Respond to this request: {{ args[0] }}", id="input"),
        PromptTask("Improve the writing", id="improve"),
        PromptTask("Respond as a pirate", id="output"),
    ],
)

pipeline.run("My favorite animal is a Liger.")

if pipeline.conversation_memory is not None:
    for run in pipeline.conversation_memory.runs:
        print("Input", run.input.value)
        print("Output", run.output.value)
```

(continues on next page)

(continued from previous page)

```
        print("\n\n")
```

Now, each *Task* creates a Conversation Memory Run when it runs. This eliminates the need to feed the output of Tasks into each other using context variables like `{{ parent_output }}` since the output of the previous Task is stored in Conversation Memory and loaded when the next Task runs.

To blend the two approaches, you can disable Conversation Memory on individual tasks by setting *PromptTask.conversation_memory* to `None`.

```
from griptape.structures import Pipeline
from griptape.tasks import PromptTask

pipeline = Pipeline(
    conversation_memory_strategy="per_task",
    tasks=[
        PromptTask("Respond to this request: {{ args[0] }}", id="input"),
        PromptTask("Improve the writing of this: {{ parent_output }}", id="improve",␣
↪conversation_memory=None),
        PromptTask("Respond as a pirate", id="output"),
    ],
)

pipeline.run("My favorite animal is a Liger.")

if pipeline.conversation_memory is not None:
    for run in pipeline.conversation_memory.runs:
        print("Input", run.input.value)
        print("Output", run.output.value)
        print("\n\n")
```

Types of Memory

Gen AI Builder provides several types of Conversation Memory to fit various use-cases.

Conversation Memory

AmazonDynamoDbConversationMemoryDriver will keep track of the full task input and output for all runs.

```
from griptape.memory.structure import ConversationMemory
from griptape.structures import Agent

agent = Agent(conversation_memory=ConversationMemory())

agent.run("Hello!")

print(agent.conversation_memory)
```

You can set the *max_runs* parameter to limit how many runs are kept in memory.

```
from griptape.memory.structure import ConversationMemory
from griptape.structures import Agent

```

(continues on next page)

(continued from previous page)

```
conversation_memory = ConversationMemory(max_runs=2)
agent = Agent(conversation_memory=conversation_memory)

agent.run("Run 1")
agent.run("Run 2")
agent.run("Run 3")
agent.run("Run 4")
agent.run("Run 5")

print(conversation_memory.runs[0].input == "run4")
print(conversation_memory.runs[1].input == "run5")
```

Summary Conversation Memory

SummaryConversationMemory will progressively summarize task input and output of runs.

You can choose to offset which runs are summarized with the query_vector(vector, *, count=None, namespace=None, include_vectors=False, offset=None, **kwargs) parameter.

```
from griptape.memory.structure import SummaryConversationMemory
from griptape.structures import Agent
from griptape.utils import Conversation

agent = Agent(conversation_memory=SummaryConversationMemory(offset=2))

agent.run("Hello my name is John?")
agent.run("What is my name?")
agent.run("My favorite color is blue.")

print(Conversation(agent.conversation_memory))
```

Observability

Overview

The *Observability* context manager sends telemetry (metrics and traces) for all functions and methods annotated with the `@observable` decorator to a destination of your choice. This is useful for monitoring and debugging your application.

Observability is completely optional. To opt in, wrap your application code with the *Observability* context manager, for example:

```
from griptape.drivers.observability.griptape_cloud import␣
↪GriptapeCloudObservabilityDriver
from griptape.observability import Observability
from griptape.structures import Agent

observability_driver = GriptapeCloudObservabilityDriver()

with Observability(observability_driver=observability_driver):
    # Important! Only code within this block is subject to observability
    agent = Agent()
    agent.run("Name the five greatest rappers of all time")
```

Tracing Custom Code

All functions and methods annotated with the `@observable` decorator will be traced when invoked within the context of the *Observability* context manager, including functions and methods defined outside of the Gen AI Builder SDK. Thus to trace custom code, you just need to add the `@observable` decorator to your function or method, then invoke it within the *Observability* context manager.

For example:

```
import time

from griptape.common import observable
from griptape.drivers.observability.griptape_cloud import␣
↪GriptapeCloudObservabilityDriver
from griptape.observability import Observability

#  Decorate a function

@observable
def my_function() -> None:
    time.sleep(3)

class MyClass:
    # Decorate a method
    @observable
    def my_method(self) -> None:
        time.sleep(1)
        my_function()
        time.sleep(2)

observability_driver = GriptapeCloudObservabilityDriver()

#  When invoking the instrumented code from within the Observability context manager,␣
↪the

#  telemetry for the custom code will be sent to the destination specified by the␣
↪driver.

with Observability(observability_driver=observability_driver):
    my_function()
    MyClass().my_method()
```

Pipelines

Overview

A *HuggingFace Pipelines* is very similar to an *agent* , but allows for multiple tasks.

You can access the final output of the Pipeline by using the *Structured Output* attribute.

Context

Pipelines have access to the following *RAG Context* variables in addition to the *base context* .

- `task_outputs` : dictionary containing mapping of all task IDs to their outputs.
- `parent_output` : output from the parent task if one exists, otherwise `None` .
- `parent` : parent task if one exists, otherwise `None` .
- `child` : child task if one exists, otherwise `None` .

Pipeline

```
from griptape.structures import Pipeline
from griptape.tasks import PromptTask

pipeline = Pipeline()

pipeline.add_tasks(
    # take the first argument from the pipeline `run` method
    PromptTask("{{ args[0] }}"),
    # take the output from the previous task and insert it into the prompt
    PromptTask("Say the following like a pirate: {{ parent_output }}"),
)

pipeline.run("Write me a haiku about sailing.")
```

Rulesets

Overview

A *Ruleset* can be used to define *Ruleset* s for *Structures* and add_tasks(*tasks) . Gen AI Builder places Rules into the LLM's system prompt for strong control over the output.

Types of Rules

Rule

Ruleset s shape the LLM's behavior by defining specific guidelines or instructions for how it should interpret and respond to inputs. Rules can be used to modify language style, tone, or even behavior based on what you define.

> **Tip**
>
> Avoid writing large amounts of text in a single Rule. Breaking down your Rules generally helps the LLM follow them more effectively. Additionally, it makes it easier to evaluate the Rule's effectiveness using tools like the *Eval Engine* . If you have an existing system prompt, consider *overriding the default system prompt* instead.

```
from griptape.rules import Rule, Ruleset
from griptape.structures import Agent

pipeline = Agent(
    rulesets=[
```

(continues on next page)

(continued from previous page)

```
        Ruleset(
            name="Personality",
            rules=[Rule("Talk like a pirate.")],
        ),
    ]
)

pipeline.run("Hi there! How are you?")
```

Json Schema

> **Tip**
>
> *Structured Output* provides a more robust solution for having the LLM generate structured output.

JsonSchemaRule s defines a structured format for the LLM's output by providing a JSON schema. This is particularly useful when you need the LLM to return well-formed data, such as JSON objects, with specific fields and data types.

```
import json

import schema

from griptape.rules.json_schema_rule import JsonSchemaRule
from griptape.structures import Agent

agent = Agent(
    rules=[
        JsonSchemaRule(
            schema.Schema({"answer": str, "relevant_emojis": schema.Schema(["str"])}).
↪json_schema("Output Format")
        )
    ]
)

output = agent.run("What is the sentiment of this message?: I am so happy!").output

print(json.dumps(json.loads(output.value), indent=2))
```

Although Gen AI Builder leverages the `schema` library, you're free to use any JSON schema generation library to define your schema!

For example, using `pydantic` :

```
from __future__ import annotations

import pydantic

from griptape.rules.json_schema_rule import JsonSchemaRule
from griptape.structures import Agent

class SentimentModel(pydantic.BaseModel):
```

(continues on next page)

(continued from previous page)

```
    answer: str
    relevant_emojis: list[str]

agent = Agent(rules=[JsonSchemaRule(SentimentModel.model_json_schema())])

output = agent.run("What is the sentiment of this message?: I am so happy!").output

sentiment_analysis = SentimentModel.model_validate_json(output.value)

#  Autocomplete via dot notation 

print(sentiment_analysis.answer)
print(sentiment_analysis.relevant_emojis)
```

Structure

Rulesets

You can define a Ruleset at the Structure level if you need to have certain behaviors across all Tasks.

```
from griptape.rules import Rule, Ruleset
from griptape.structures import Pipeline
from griptape.tasks import PromptTask

pipeline = Pipeline(
    rulesets=[
        Ruleset(
            name="Employment",
            rules=[
                Rule("Behave like a polite customer support agent"),
                Rule("Act like you work for company SkaterWorld, Inc."),
                Rule("Discuss only topics related to skateboarding"),
                Rule("Limit your response to fewer than 5 sentences."),
            ],
        ),
        Ruleset(
            name="Background",
            rules=[
                Rule("Your name is Todd"),
            ],
        ),
    ]
)

pipeline.add_tasks(
    PromptTask(input="Respond to this users question: {{ args[0] }}"),
    PromptTask(input="Extract keywords from this response: {{ parent_output }}"),
)

pipeline.run("How do I do a kickflip?")
```

Rules

You can pass *Ruleset* directly to the Structure to have a Ruleset created for you.

```
from griptape.rules import Rule
from griptape.structures import Pipeline
from griptape.tasks import PromptTask

pipeline = Pipeline(
    rules=[
        Rule("Respond only using emojis"),
    ],
)

pipeline.add_tasks(
    PromptTask("Respond to this question from the user: {{ args[0] }}"),
    PromptTask("How would you rate your response (1-5)? 1 being bad, 5 being good.␣
↪Response: {{parent_output}}"),
)

pipeline.run("How do I bake a cake?")
```

Task

Rulesets

You can define a Ruleset at the Task level if you need to have different behaviors per Task.

```
from griptape.rules import Rule, Ruleset
from griptape.structures import Pipeline
from griptape.tasks import PromptTask

pipeline = Pipeline()

pipeline.add_tasks(
    PromptTask(
        input="Respond to the following prompt: {{ args[0] }}",
        rulesets=[
            Ruleset(
                name="Emojis",
                rules=[
                    Rule("Respond using uppercase characters only."),
                ],
            )
        ],
    ),
    PromptTask(
        input="Determine the sentiment of the following text: {{ parent_output }}",
        rulesets=[
            Ruleset(
                name="Diacritic",
                rules=[
                    Rule("Respond using diacritic characters only."),
```

(continues on next page)

(continued from previous page)

```
                ],
            )
        ],
    ),
)

pipeline.run("I love skateboarding!")
```

Rules

You can pass *Ruleset* directly to the Task to have a Ruleset created for you.

```
from griptape.rules import Rule
from griptape.structures import Pipeline
from griptape.tasks import PromptTask

pipeline = Pipeline()

pipeline.add_tasks(
    PromptTask(
        rules=[
            Rule("Write your answer in json with a single key emoji_response"),
            Rule("Respond only using emojis"),
        ],
    ),
)

pipeline.run("How are you?")
```

Overriding System Prompts

While Rulesets are the preferred way to steer LLM output, sometimes you may want to fully override the system prompt.

```
from textwrap import dedent

from griptape.rules import Rule
from griptape.structures import Pipeline
from griptape.tasks.prompt_task import PromptTask

pipeline = Pipeline(
    tasks=[
        PromptTask(
            rules=[Rule("Your favorite color is blue")],  # Will be ignored!
            generate_system_template=lambda _: dedent("""
                Employment:
                    - Behave like a polite customer support agent
                    - Act like you work for company SkaterWorld, Inc.
                    - Discuss only topics related to skateboarding
                    - Limit your response to fewer than 5 sentences.
```

(continues on next page)

(continued from previous page)

```
                Background:
                    - Your name is Todd
                """),
        )
    ]
)

pipeline.run("What is your name and favorite color?")
```

Note that when overriding the system prompt, it is your responsibility to integrate anything that goes in *by default* . You can achieve this by appending the default system prompt to your custom prompt like so:

```
from textwrap import dedent

from griptape.rules import Rule
from griptape.structures import Pipeline
from griptape.tasks.prompt_task import PromptTask

pipeline = Pipeline(
    tasks=[
        PromptTask(
            rules=[Rule("Your favorite color is blue")],
            generate_system_template=lambda task: dedent(
                """
                Employment:
                    - Behave like a polite customer support agent
                    - Act like you work for company SkaterWorld, Inc.
                    - Discuss only topics related to skateboarding
                    - Limit your response to fewer than 5 sentences.

                Background:
                    - Your name is Todd
                """
                + task.default_generate_system_template(task)
            ),
        )
    ]
)

pipeline.run("What is your name and favorite color?")
```

Task Memory and Off Prompt

Overview

Task Memory is a powerful feature of Gen AI Builder that allows you to control where the data returned by *Tools* is stored. This is useful in the following scenarios:

- **Security requirements**: many organizations don't want data to leave their cloud for regulatory and security reasons.
- **Long textual content**: when textual content returned by Tools can't fit in the token limit, it's often useful to perform actions on it as a separate operation, not through the main LLM.

- **Non-textual content**: Tools can generate images, videos, PDFs, and other non-textual content that can be stored in Task Memory and acted upon later by other Tools.

> **Tip**
>
> Running into issue with Task Memory? Check out the *Task Memory Considerations* section for some common pitfalls.

Off Prompt

You can enable or disable sending a Tool's results to Task Memory with the `off_prompt` parameter. By default, all Tools have `off_prompt` set to `False` making this an opt-in feature. When `off_prompt` is set to `True`, the Tool will store its output in Task Memory. When `off_prompt` is set to `False`, the Tool will return its output directly to the LLM.

Lets look at a simple example where `off_prompt` is set to `False`:

```
from griptape.structures import Agent
from griptape.tools import CalculatorTool

#  Create an agent with the CalculatorTool tool

agent = Agent(tools=[CalculatorTool(off_prompt=False)])

agent.run("What is the square root of 12345?")
```

Since the result of the CalculatorTool Tool is neither sensitive nor too large, we can set `off_prompt` to `False` and not use Task Memory.

Let's explore what happens when `off_prompt` is set to `True`:

```
from griptape.structures import Agent
from griptape.tools import CalculatorTool

#  Create an agent with the CalculatorTool tool

agent = Agent(tools=[CalculatorTool(off_prompt=True)])

agent.run("What is the square root of 12345?")
```

When we set `off_prompt` to `True`, the Agent does not function as expected, even generating an error. This is because the Calculator output is being stored in Task Memory but the Agent has no way to access it. To fix this, we need a *Tool that can read from Task Memory* such as the `PromptSummaryTool`. This is an example of *Not Providing a Task Memory Compatible Tool*.

Prompt Summary Tool

The *PromptSummaryTool* is a Tool that allows an Agent to summarize the Artifacts in Task Memory. It has the following methods:

Let's add `PromptSummaryTool` to the Agent and run the same task. Note that on the `PromptSummaryTool` we've set `off_prompt` to `False` so that the results of the query can be returned directly to the LLM. If we had kept it as `True`, the results would have been stored back Task Memory which would've put us back to square one. See *Task Memory Looping* for more information on this scenario.

```
from griptape.structures import Agent
from griptape.tools import CalculatorTool, PromptSummaryTool

#  Create an agent with the Calculator tool

agent = Agent(tools=[CalculatorTool(off_prompt=True), PromptSummaryTool(off_
↪prompt=False)])

agent.run("What is the square root of 12345?")
```

While this fixed the problem, it took a handful more steps than when we just had `CalculatorTool()`. Something like a basic calculation is an instance of where *Task Memory May Not Be Necessary*. Let's look at a more complex example where Task Memory shines.

Large Data

Let's say we want to query the contents of a very large webpage.

```
from griptape.structures import Agent
from griptape.tools import WebScraperTool

#  Create an agent with the WebScraperTool tool

agent = Agent(tools=[WebScraperTool()])

agent.run(
    "According to this page https://en.wikipedia.org/wiki/Elden_Ring, how many copies␣
↪of Elden Ring have been sold?"
)
```

When running this example, we get the following error:

```
[04/26/24 13:20:02] ERROR    PromptTask 67e2f907f95d4850ae79f9da67df54c1
                             Error code: 400 - {error: {message: "This models maximum␣
↪context length is 8192 tokens. However, your messages resulted in 73874 tokens.
                             Please reduce the length of the messages.", type:␣
↪invalid_request_error, param: messages, code: context_length_exceeded}}
```

This is because the content of the webpage is too large to fit in the LLM's input token limit. We can fix this by storing the content in Task Memory, and then querying it with the `QueryTool`. Note that we're setting `off_prompt` to `False` on the `QueryTool` so that the *queried* content can be returned directly to the LLM.

```
from griptape.structures import Agent
from griptape.tools import QueryTool, WebScraperTool

agent = Agent(
    tools=[
        WebScraperTool(off_prompt=True),
        QueryTool(off_prompt=False),
    ]
)

agent.run(
```

(continues on next page)

(continued from previous page)

```
    "According to this page https://en.wikipedia.org/wiki/Elden_Ring, how many copies␣
↪of Elden Ring have been sold?"
)
```

And now we get the expected output.

```
[08/12/24 14:56:29] INFO     Subtask 8669ee523bb64550850566011bcd14e2
                             Response: "Elden Ring" sold 13.4 million copies␣
↪worldwide by the end of March 2022 and 25 million by June 2024. The downloadable␣
↪content (DLC)
                             "Shadow of the Erdtree" sold five million copies within␣
↪three days of its release.
[08/12/24 14:56:30] INFO     PromptTask d3ce58587dc944b0a30a205631b82944
                             Output: Elden Ring sold 13.4 million copies worldwide by␣
↪the end of March 2022 and 25 million by June 2024.
```

Sensitive Data

Because Task Memory splits up the storage and retrieval of data, you can use different models for each step.

Here is an example where we use GPT-4 to orchestrate the Tools and store the data in Task Memory, and Anthropic's Claude 3 Haiku model to query the raw content. In this example, GPT-4 *never* sees the contents of the page, only that it was stored in Task Memory. Even the query results generated by the Haiku model are stored in Task Memory so that the `FileManagerTool` can save the results to disk without GPT-4 ever seeing them.

```
from griptape.configs import Defaults
from griptape.configs.drivers import OpenAiDriversConfig
from griptape.drivers.embedding.openai import OpenAiEmbeddingDriver
from griptape.drivers.prompt.amazon_bedrock import AmazonBedrockPromptDriver
from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.drivers.vector.local import LocalVectorStoreDriver
from griptape.structures import Agent
from griptape.tools import FileManagerTool, QueryTool, WebScraperTool

Defaults.drivers_config = OpenAiDriversConfig(
    prompt_driver=OpenAiChatPromptDriver(model="gpt-4"),
)

vector_store_driver = LocalVectorStoreDriver(embedding_driver=OpenAiEmbeddingDriver())

agent = Agent(
    tools=[
        WebScraperTool(off_prompt=True),
        QueryTool(
            off_prompt=True,
            prompt_driver=AmazonBedrockPromptDriver(model="anthropic.claude-3-haiku-
↪20240307-v1:0"),
        ),
        FileManagerTool(off_prompt=True),
    ],
)
```

(continues on next page)

(continued from previous page)

```
agent.run(
    "Use this page https://en.wikipedia.org/wiki/Elden_Ring to find how many copies␣
↪of Elden Ring have been sold, and then save the result to a file."
)
```

Tools That Can Read From Task Memory

As seen in the previous example, certain Tools are designed to read directly from Task Memory. This means that you can use these Tools to interact with the data stored in Task Memory without needing to pass it through the LLM.

Today, these include:

- `:ref:`PromptSummaryTool <PromptSummaryTool>``
- `:ref:`ExtractionTool <ExtractionTool>``
- `:ref:`RagTool <RagTool>``
- `:ref:`FileManagerTool <FileManagerTool>``
- `:ref:`ImageQueryTool <ImageQueryTool>``

Task Memory Considerations

Task Memory is a powerful feature of Gen AI Builder, but with great power comes great responsibility. Here are some things to keep in mind when using Task Memory:

Tool Return Types

Gen AI Builder will only store Artifacts in Task Memory that have been explicitly defined in the `artifact_storages` parameter of the `TaskMemory` object. If you try to store an Artifact that is not defined in `artifact_storages`, Gen AI Builder will raise an error. The exception to this is `InfoArtifact` s and `ErrorArtifact` s. Gen AI Builder will never store these Artifacts store in Task Memory. By default, Gen AI Builder will store `TextArtifact` 's, `BlobArtifact` 's in Task Memory. Additionally, Gen AI Builder will also store the elements of `ListArtifact` 's as long as they are of a supported Artifact type.

Not Providing a Task Memory Compatible Tool

When using Task Memory, make sure that you have at least one Tool that can read from Task Memory. If you don't, the data stored in Task Memory will be inaccessible to the Agent and it may hallucinate Tool Activities.

```
from griptape.structures import Agent
from griptape.tools import WebScraperTool

agent = Agent(
    tools=[
        WebScraperTool(off_prompt=True)  # `off_prompt=True` will store the data in␣
↪Task Memory
        # Missing a Tool that can read from Task Memory
    ]
```

(continues on next page)

(continued from previous page)

```
)
agent.run(
    "According to this page https://en.wikipedia.org/wiki/San_Francisco, what is the␣
↪population of San Francisco?"
)
```

Task Memory Looping

An improper configuration of Tools can lead to the LLM using the Tools in a loop. For example, if you have a Tool that stores data in Task Memory and another Tool that queries that data from Task Memory (*Tools That Can Read From Task Memory*), make sure that the query Tool does not store the data back in Task Memory. This can create a loop where the same data is stored and queried over and over again.

```
from griptape.structures import Agent
from griptape.tools import PromptSummaryTool, WebScraperTool

agent = Agent(
    tools=[
        WebScraperTool(off_prompt=True),  # This tool will store the data in Task␣
↪Memory
        PromptSummaryTool(
            off_prompt=True
        ),  # This tool will store the data back in Task Memory with no way to get it␣
↪out
    ]
)
agent.run(
    "According to this page https://en.wikipedia.org/wiki/Dark_forest_hypothesis,␣
↪what is the Dark Forest Hypothesis?"
)
```

Task Memory May Not Be Necessary

Task Memory may not be necessary for all use cases. If the data returned by a Tool is not sensitive, not too large, and does not need to be acted upon by another Tool, you can leave the default of `off_prompt` to `False` and return the data directly to the LLM.

```
from griptape.structures import Agent
from griptape.tools import CalculatorTool

agent = Agent(
    tools=[
        CalculatorTool()  # Default value of `off_prompt=False` will return the data␣
↪directly to the LLM
    ]
)
agent.run("What is 10 ^ 3, 55 / 23, and 12345 * 0.5?")
```

Tasks

Overview

A add_tasks(*tasks) is a purpose-built abstraction for the Large Language Model (LLM). Gen AI Builder offers various types of Tasks, each suitable for specific use cases.

Context

Tasks that take input have a field *run(prompt_input)* which lets you define the Task objective. Within the *run(prompt_input)* , you can access the following *RAG Context* variables:

- `args` : an array of arguments passed to the `.run()` method.
- `structure` : the structure that the task belongs to.
- user defined context variables

Additional *RAG Context* variables may be added based on the Structure running the task.

```
from griptape.structures import Agent
from griptape.tasks import PromptTask

agent = Agent()
agent.add_task(
    PromptTask(
        "Respond to the users following question {{ args[0] }} in the language {
↪{preferred_language}} and tone {{tone}}.",
        context={"preferred_language": "ENGLISH", "tone": "PLAYFUL"},
    )
)

agent.run("How do I bake a cake?")
```

Hooks

All Tasks implement *RunnableMixin* which provides `on_before_run` and `on_after_run` hooks for the Task lifecycle.

These hooks can be used to perform actions before and after the Task is run. For example, you can mask sensitive information before running the Task, and transform the output after the Task is run.

```
import json
import re

from griptape.structures import Agent
from griptape.tasks import PromptTask
from griptape.tasks.base_task import BaseTask

SSN_PATTERN = re.compile(r"\b\d{3}-\d{2}-\d{4}\b")

original_input = None

def on_before_run(task: BaseTask) -> None:
    global original_input  # noqa: PLW0603
```

(continues on next page)

(continued from previous page)

```
    original_input = task.input.value

    if isinstance(task, PromptTask):
        task.input = SSN_PATTERN.sub("xxx-xx-xxxx", task.input.value)

def on_after_run(task: BaseTask) -> None:
    if task.output is not None:
        task.output.value = json.dumps(
            {"original_input": original_input, "masked_input": task.input.value,
↪"output": task.output.value}, indent=2
        )

agent = Agent(
    tasks=[
        PromptTask(
            "Respond to this user: {{ args[0] }}",
            on_before_run=on_before_run,
            on_after_run=on_after_run,
        )
    ]
)
agent.run("Hello! My favorite color is blue, and my social security number is 123-45-
↪6789.")
```

Prompt Task

For general-purpose interaction with LLMs, use the *PromptTask* :

```
from griptape.structures import Agent
from griptape.tasks import PromptTask

agent = Agent()
agent.add_task(
    # take the first argument from the agent `run` method
    PromptTask("Respond to the following request: {{ args[0] }}"),
)

agent.run("Write me a haiku")
```

Tools

You can pass in one or more Tools which the LLM will decide to use through Chain of Thought (CoT) reasoning. Because tool execution uses CoT, it is recommended to only use with very capable models.

```
from griptape.structures import Agent
from griptape.tasks import PromptTask
from griptape.tools import FileManagerTool, PromptSummaryTool, WebScraperTool

agent = Agent()
agent.add_task(
    PromptTask(
```

(continues on next page)

(continued from previous page)

```
        "Load https://www.griptape.ai, summarize it, and store it in a file called␣
↪griptape.txt",
        tools=[WebScraperTool(off_prompt=True), FileManagerTool(off_prompt=True),␣
↪PromptSummaryTool(off_prompt=True)],
    ),
)

agent.run()
```

Reflect On Tool Use

By default, Gen AI Builder will pass the results of Tool runs back to the LLM for reflection. This enables the LLM to reason about the results and potentially use additional tools.

However, there may be times where you may want the LLM to give you back the results directly, without reflection. You can disable this behavior by setting *reflect_on_tool_use* to `False` .

```
from griptape.artifacts.list_artifact import ListArtifact
from griptape.drivers.web_search.duck_duck_go import DuckDuckGoWebSearchDriver
from griptape.tasks import PromptTask
from griptape.tools import WebScraperTool, WebSearchTool

search_task = PromptTask(
    tools=[WebSearchTool(web_search_driver=DuckDuckGoWebSearchDriver())],
    reflect_on_tool_use=False,
)
search_results = search_task.run("Do two searches, one for vim and one for emacs.")

#  When disabling `reflect_on_tool_use`, the Tasks results will be returned as a␣
↪ListArtifact.

#  Each item in the ListArtifact will be the result of a single tool execution.

if isinstance(search_results, ListArtifact):
    for result in search_results:
        print(result)

scrape_task = PromptTask(
    tools=[WebScraperTool()],
    reflect_on_tool_use=True,
)

#  If we dont care about the individual results, we can join them back together␣
↪before passing to the next task.

answer = scrape_task.run(["Compare and contrast vim and emacs: ", search_results.to_
↪text()])
```

Important

Disabling reflection will prevent the LLM from using one Tool to inform the use of another Tool. Instead, you

must coordinate the Tool uses yourself.

Images

If the model supports it, you can also pass image inputs:

```
from griptape.loaders import ImageLoader
from griptape.structures import Agent

agent = Agent()

image_artifact = ImageLoader().load("tests/resources/mountain.jpg")

agent.run([image_artifact, "Whats in this image?"])
```

Tool Task

Warning

ToolTask is deprecated and will be removed in a future version. Use *Prompt Task* with *reflect_on_tool_use* set to *False* instead.

Another way to use *Gen AI Builder Tools* , is with a *Tool Task* . This Task takes in a single Tool which the LLM will use without Chain of Thought (CoT) reasoning. Because this Task does not use CoT, it is better suited for less capable models.

```
from griptape.structures import Agent
from griptape.tasks import ToolTask
from griptape.tools import CalculatorTool

#  Initialize the agent and add a task

agent = Agent()
agent.add_task(ToolTask(tool=CalculatorTool()))

#  Run the agent with a prompt

agent.run("Give me the answer for 5*4.")
```

Extraction Task

To extract information from text, use an *ExtractionTask* . This Task takes an *Extraction Engines* , and a set of arguments specific to the Engine.

CSV Extraction

```
from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.engines import CsvExtractionEngine
from griptape.structures import Agent
```

(continues on next page)

(continued from previous page)

```
from griptape.tasks import ExtractionTask

#  Instantiate the CSV extraction engine

csv_extraction_engine = CsvExtractionEngine(
    prompt_driver=OpenAiChatPromptDriver(model="gpt-3.5-turbo"), column_names=["Name",
↪ "Age", "Address"]
)

#  Define some unstructured data and columns

csv_data = """
Alice, 28, lives in New York.
Bob, 35 lives in California.
Charlie is 40 and lives in Texas.
"""

#  Create an agent and add the ExtractionTask to it

agent = Agent()
agent.add_task(
    ExtractionTask(
        extraction_engine=csv_extraction_engine,
    )
)

#  Run the agent

agent.run(csv_data)
```

JSON Extraction

```
from schema import Schema

from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.engines import JsonExtractionEngine
from griptape.structures import Agent
from griptape.tasks import ExtractionTask

#  Instantiate the json extraction engine

json_extraction_engine = JsonExtractionEngine(
    prompt_driver=OpenAiChatPromptDriver(model="gpt-3.5-turbo"),
    template_schema=Schema({"users": [{"name": str, "age": int, "location": str}]}).
↪json_schema("UserSchema"),
)

#  Define some unstructured data and a schema

json_data = """
Alice (Age 28) lives in New York.
```

(continues on next page)

(continued from previous page)

```
Bob (Age 35) lives in California.
"""

agent = Agent()
agent.add_task(
    ExtractionTask(
        extraction_engine=json_extraction_engine,
    )
)

#  Run the agent

agent.run(json_data)
```

Text Summary Task

To summarize a text, use the *TextSummaryTask* . This Task takes an *Summarization Engine* , and a set of arguments to the engine.

```
from griptape.structures import Agent
from griptape.tasks import TextSummaryTask

#  Create a new agent

agent = Agent()

#  Add the TextSummaryTask to the agent

agent.add_task(TextSummaryTask())

#  Run the agent

agent.run(
    "Artificial Intelligence (AI) is a branch of computer science that deals with "
    "creating machines capable of thinking and learning. It encompasses various␣
↪fields "
    "such as machine learning, neural networks, and deep learning. AI has the␣
↪potential "
    "to revolutionize many sectors, including healthcare, finance, and transportation.
↪ "
    "Our life in this modern age depends largely on computers. It is almost␣
↪impossible "
    "to think about life without computers. We need computers in everything that we␣
↪use "
    "in our daily lives. So it becomes very important to make computers intelligent␣
↪so "
    "that our lives become easy. Artificial Intelligence is the theory and␣
↪development "
    "of computers, which imitates the human intelligence and senses, such as visual "
    "perception, speech recognition, decision-making, and translation between␣
↪languages."
    " Artificial Intelligence has brought a revolution in the world of technology. "
```

(continues on next page)

(continued from previous page)

```
)
```

RAG Task

To query text, use the *RagTask* . This task takes a *RAG Engine* , and a set of arguments specific to the engine.

```
from griptape.artifacts import TextArtifact
from griptape.drivers.embedding.openai import OpenAiEmbeddingDriver
from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.drivers.vector.local import LocalVectorStoreDriver
from griptape.engines.rag import RagEngine
from griptape.engines.rag.modules import PromptResponseRagModule,␣
↪VectorStoreRetrievalRagModule
from griptape.engines.rag.stages import ResponseRagStage, RetrievalRagStage
from griptape.structures import Agent
from griptape.tasks import RagTask

#  Initialize Embedding Driver and Vector Store Driver

vector_store_driver = LocalVectorStoreDriver(embedding_driver=OpenAiEmbeddingDriver())

artifacts = [
    TextArtifact("Griptape builds AI-powered applications that connect securely to␣
↪your enterprise data and APIs."),
    TextArtifact("Griptape Agents provide incredible power and flexibility when␣
↪working with large language models."),
]
vector_store_driver.upsert_collection({"griptape": artifacts})

#  Instantiate the agent and add RagTask with the RagEngine

agent = Agent()
agent.add_task(
    RagTask(
        "Respond to the following query: {{ args[0] }}",
        rag_engine=RagEngine(
            retrieval_stage=RetrievalRagStage(
                retrieval_modules=[
                    VectorStoreRetrievalRagModule(
                        vector_store_driver=vector_store_driver, query_params={
↪"namespace": "griptape", "top_n": 20}
                    )
                ]
            ),
            response_stage=ResponseRagStage(
                response_modules=[PromptResponseRagModule(prompt_
↪driver=OpenAiChatPromptDriver(model="gpt-4.1"))]
            ),
        ),
    )
)
```

(continues on next page)

(continued from previous page)

```
#  Run the agent with a query string

agent.run("Give me information about Griptape")
```

Code Execution Task

To execute an arbitrary Python function, use the *CodeExecutionTask* . This task takes a python function, and authors can elect to return a custom artifact.

In this example, the `generate_title` function combines a hero's name and setting from previous tasks with a random title, returning a `TextArtifact` that contains the generated fantasy title. The output of this task can then be referenced by subsequent tasks using the `parent_outputs` templating variable, as shown in the final `PromptTask` .

```
import random

from griptape.artifacts import BaseArtifact, TextArtifact
from griptape.structures import Workflow
from griptape.tasks import CodeExecutionTask, PromptTask
from griptape.utils import StructureVisualizer

#  Define a function that will generate a fantasy title by combining inputs

def generate_title(task: CodeExecutionTask) -> BaseArtifact:
    # Make sure this is part of a valid structure
    structure = task.structure
    if structure is None:
        raise ValueError("Task must be part of a structure to run")

    # Retrieve outputs from previous tasks using their task IDs
    hero = structure.find_task("hero").output  # Gets the hero name from first prompt
    setting = structure.find_task("setting").output  # Gets the setting from second␣
↪prompt

    # List of epic titles to randomly choose from
    titles = ["the Brave", "the Shadow", "the Flame", "the Eternal", "the Cursed"]
    # Combine the hero name, random title, and setting into a full title
    generated_title = f"{hero} {random.choice(titles)} of {setting}"

    # Return the generated title as a TextArtifact that can be used by other tasks
    return TextArtifact(generated_title)

#  Create a new workflow to organize our tasks

workflow = Workflow()

#  Add a sequence of connected tasks to the workflow

workflow.add_tasks(
    # First task: Prompt for a hero name
    # The child_ids parameter connects this to the generate_title task
    PromptTask("Name a fantasy hero (e.g., Eldric)", id="hero", child_ids=["generate_
↪title"]),
```

(continues on next page)

(continued from previous page)

```
    # Second task: Prompt for a setting
    # Also connected to generate_title task
    PromptTask(
        "Describe a mystical setting in a couple sentences (e.g., the Shattered Isles)
↪",
        id="setting",
        child_ids=["generate_title"],
    ),
    # Third task: Execute our custom code to generate the title
    # The on_run parameter specifies which function to execute
    CodeExecutionTask(on_run=generate_title, id="generate_title", child_ids=["story
↪"]),
    # Fourth task: Prompt for a story using the generated title
    # {{ parent_outputs[generate_title] }} references the output from the previous␣
↪task
    PromptTask(
        "Write a brief, yet heroic tale about {{ parent_outputs[generate_title] }}. ",
        id="story",
    ),
)

#  Visualize the workflow structure (helpful for debugging)

print(StructureVisualizer(workflow).to_url())

#  Execute the entire workflow

workflow.run()
```

Branch Task

By default, a *Workflow* will only run a Task when all the Tasks it depends on have finished.

You can make use of *BranchTask* in order to tell the Workflow not to run all dependent tasks, but instead to pick and choose one or more paths to go down.

The `BranchTask` 's *python_runner* function can return one of three things:

1. An `InfoArtifact` containing the `id` of the Task to run next.
2. A `ListArtifact` of `InfoArtifact` s containing the `id` s of the Tasks to run next.
3. An *empty* `ListArtifact` to indicate that no Tasks should be run next.

```
from griptape.artifacts import InfoArtifact
from griptape.loaders import ImageLoader
from griptape.structures import Workflow
from griptape.tasks import BranchTask, PromptTask

def on_run(task: BranchTask) -> InfoArtifact:
    if "hot dog" in task.input.value:
        return InfoArtifact("hot_dog")
    return InfoArtifact("not_hot_dog")
```

(continues on next page)

(continued from previous page)

```
image_artifact = ImageLoader().load("tests/resources/mountain.png")

workflow = Workflow(
    tasks=[
        PromptTask(["What is in this image?", image_artifact], child_ids=["branch"]),
        BranchTask(
            "{{ parents_output_text }}",
            on_run=on_run,
            id="branch",
            child_ids=["hot_dog", "not_hot_dog"],
        ),
        PromptTask("Tell me about hot dogs", id="hot_dog"),
        PromptTask("Tell me about anything but hotdogs", id="not_hot_dog"),
    ]
)

workflow.run()
```

Image Generation Tasks

To generate an image, use one of the following add_tasks(*tasks) . All Image Generation Tasks accept an *Image Generation Drivers* .

All successful Image Generation Tasks will always output an *Image Artifact* . Each task can be configured to additionally write the generated image to disk by providing either the `output_file` or `output_dir` field. The `output_file` field supports file names in the current directory (`my_image.png`), relative directory prefixes (`images/my_image.png`), or absolute paths (`/usr/var/my_image.png`). By setting `output_dir` , the task will generate a file name and place the image in the requested directory.

Prompt Image Generation Task

The *Prompt Image Generation Task* generates an image from a text prompt.

```
from griptape.drivers.image_generation.openai import OpenAiImageGenerationDriver
from griptape.structures import Pipeline
from griptape.tasks import PromptImageGenerationTask

#  Create a driver configured to use OpenAIs DALL-E 3 model.

driver = OpenAiImageGenerationDriver(
    model="dall-e-3",
    quality="hd",
    style="natural",
)

#  Instantiate a pipeline.

pipeline = Pipeline()

#  Add a PromptImageGenerationTask to the pipeline.

pipeline.add_tasks(
```

(continues on next page)

(continued from previous page)

```
    PromptImageGenerationTask(
        input="{{ args[0] }}",
        image_generation_driver=driver,
        output_dir="images/",
    )
)

pipeline.run("An image of a mountain on a summer day")
```

Variation Image Generation Task

The *Variation Image Generation Task* generates an image using an input image and a text prompt. The input image is used as a basis for generating a new image as requested by the text prompt.

```
from griptape.drivers.image_generation.amazon_bedrock import␣
↪AmazonBedrockImageGenerationDriver
from griptape.drivers.image_generation_model.bedrock_stable_diffusion import (
    BedrockStableDiffusionImageGenerationModelDriver,
)
from griptape.loaders import ImageLoader
from griptape.structures import Pipeline
from griptape.tasks import VariationImageGenerationTask

#  Create a driver configured to use Stable Diffusion via Bedrock.

driver = AmazonBedrockImageGenerationDriver(
    image_generation_model_driver=BedrockStableDiffusionImageGenerationModelDriver(),
    model="stability.stable-diffusion-xl-v1",
)

#  Load input image artifact.

image_artifact = ImageLoader().load("tests/resources/mountain.png")

#  Instantiate a pipeline.

pipeline = Pipeline()

#  Add a VariationImageGenerationTask to the pipeline.

pipeline.add_task(
    VariationImageGenerationTask(
        input=("{{ args[0] }}", image_artifact),
        image_generation_driver=driver,
        output_dir="images/",
    )
)

pipeline.run("An image of a mountain landscape on a snowy winter day")
```

Inpainting Image Generation Task

The *Inpainting Image Generation Task* generates an image using an input image, a mask image, and a text prompt. The input image will be modified within the bounds of the mask image as requested by the text prompt.

```
from griptape.drivers.image_generation.amazon_bedrock import␣
↪AmazonBedrockImageGenerationDriver
from griptape.drivers.image_generation_model.bedrock_stable_diffusion import (
    BedrockStableDiffusionImageGenerationModelDriver,
)
from griptape.loaders import ImageLoader
from griptape.structures import Pipeline
from griptape.tasks import InpaintingImageGenerationTask

#  Create a driver configured to use Stable Diffusion via Bedrock.

driver = AmazonBedrockImageGenerationDriver(
    image_generation_model_driver=BedrockStableDiffusionImageGenerationModelDriver(),
    model="stability.stable-diffusion-xl-v1",
)

#  Load input image artifacts.

image_artifact = ImageLoader().load("tests/resources/mountain.png")

mask_artifact = ImageLoader().load("tests/resources/mountain-mask.png")

#  Instantiate a pipeline.

pipeline = Pipeline()

#  Add an InpaintingImageGenerationTask to the pipeline.

pipeline.add_task(
    InpaintingImageGenerationTask(
        input=("{{ args[0] }}", image_artifact, mask_artifact), image_generation_
↪driver=driver, output_dir="images/"
    )
)

pipeline.run("An image of a castle built into the side of a mountain")
```

Outpainting Image Generation Task

The *Outpainting Image Generation Task* generates an image using an input image, a mask image, and a text prompt. The input image will be modified outside the bounds of a mask image as requested by the text prompt.

```
from griptape.drivers.image_generation.amazon_bedrock import␣
↪AmazonBedrockImageGenerationDriver
from griptape.drivers.image_generation_model.bedrock_stable_diffusion import (
    BedrockStableDiffusionImageGenerationModelDriver,
)
from griptape.loaders import ImageLoader
```

(continues on next page)

(continued from previous page)

```
from griptape.structures import Pipeline
from griptape.tasks import OutpaintingImageGenerationTask

#  Create a driver configured to use Stable Diffusion via Bedrock.

driver = AmazonBedrockImageGenerationDriver(
    image_generation_model_driver=BedrockStableDiffusionImageGenerationModelDriver(),
    model="stability.stable-diffusion-xl-v1",
)

#  Load input image artifacts.

image_artifact = ImageLoader().load("tests/resources/mountain.png")

mask_artifact = ImageLoader().load("tests/resources/mountain-mask.png")

#  Instantiate a pipeline.

pipeline = Pipeline()

#  Add an OutpaintingImageGenerationTask to the pipeline.

pipeline.add_task(
    OutpaintingImageGenerationTask(
        input=("{{ args[0] }}", image_artifact, mask_artifact),
        image_generation_driver=driver,
        output_dir="images/",
    )
)

pipeline.run("An image of a mountain shrouded by clouds")
```

Structure Run Task

The *Structure Run Task* runs another Structure with a given input. This Task is useful for orchestrating multiple specialized Structures in a single run. Note that the input to the Task is a tuple of arguments that will be passed to the Structure.

```
import os

from griptape.drivers.structure_run.local import LocalStructureRunDriver
from griptape.drivers.web_search.google import GoogleWebSearchDriver
from griptape.rules import Rule, Ruleset
from griptape.structures import Agent, Pipeline
from griptape.tasks import StructureRunTask
from griptape.tools import (
    PromptSummaryTool,
    WebScraperTool,
    WebSearchTool,
)

def build_researcher() -> Agent:
```

(continues on next page)

(continued from previous page)

```
    return Agent(
        tools=[
            WebSearchTool(
                web_search_driver=GoogleWebSearchDriver(
                    api_key=os.environ["GOOGLE_API_KEY"],
                    search_id=os.environ["GOOGLE_API_SEARCH_ID"],
                ),
            ),
            WebScraperTool(
                off_prompt=True,
            ),
            PromptSummaryTool(off_prompt=False),
        ],
        rulesets=[
            Ruleset(
                name="Position",
                rules=[
                    Rule(
                        value="Senior Research Analyst",
                    )
                ],
            ),
            Ruleset(
                name="Objective",
                rules=[
                    Rule(
                        value="Uncover cutting-edge developments in AI and data␣
↪science",
                    )
                ],
            ),
            Ruleset(
                name="Background",
                rules=[
                    Rule(
                        value="""You work at a leading tech think tank.,
                        Your expertise lies in identifying emerging trends.
                        You have a knack for dissecting complex data and presenting␣
↪actionable insights."""
                    )
                ],
            ),
            Ruleset(
                name="Desired Outcome",
                rules=[
                    Rule(
                        value="Full analysis report in bullet points",
                    )
                ],
            ),
        ],
    )
```

(continues on next page)

(continued from previous page)

```
def build_writer() -> Agent:
    return Agent(
        input="Instructions: {{args[0]}}\nContext: {{args[1]}}",
        rulesets=[
            Ruleset(
                name="Position",
                rules=[
                    Rule(
                        value="Tech Content Strategist",
                    )
                ],
            ),
            Ruleset(
                name="Objective",
                rules=[
                    Rule(
                        value="Craft compelling content on tech advancements",
                    )
                ],
            ),
            Ruleset(
                name="Backstory",
                rules=[
                    Rule(
                        value="""You are a renowned Content Strategist, known for␣
↪your insightful and engaging articles.
                        You transform complex concepts into compelling narratives."""
                    )
                ],
            ),
            Ruleset(
                name="Desired Outcome",
                rules=[
                    Rule(
                        value="Full blog post of at least 4 paragraphs",
                    )
                ],
            ),
        ],
    )

team = Pipeline(
    tasks=[
        StructureRunTask(
            (
                """Perform a detailed examination of the newest developments in AI as␣
↪of 2024.
                Pinpoint major trends, breakthroughs, and their implications for␣
↪various industries.""",
            ),
            structure_run_driver=LocalStructureRunDriver(create_structure=build_
```

(continues on next page)

(continued from previous page)

```
↪researcher),
        ),
        StructureRunTask(
            (
                """Utilize the gathered insights to craft a captivating blog
                article showcasing the key AI innovations.
                Ensure the content is engaging yet straightforward, appealing to a␣
↪tech-aware readership.
                Keep the tone appealing and use simple language to make it less␣
↪technical.""",
                "{{parent_output}}",
            ),
            structure_run_driver=LocalStructureRunDriver(create_structure=build_
↪writer),
        ),
    ],
)

team.run()
```

Assistant Task

The *Assistant Task* enables Structures to interact with various "assistant" services using *Assistant Drivers* .

```
import os

from griptape.drivers.assistant.griptape_cloud import GriptapeCloudAssistantDriver
from griptape.structures import Pipeline
from griptape.tasks import AssistantTask

pipeline = Pipeline(
    tasks=[
        AssistantTask(
            assistant_driver=GriptapeCloudAssistantDriver(
                assistant_id=os.environ["GT_CLOUD_ASSISTANT_ID"],
            ),
        ),
    ]
)

pipeline.run("Hello!")
```

Text to Speech Task

This Task enables Structures to synthesize speech from text using *Text to Speech Drivers* .

```
import os

from griptape.drivers.text_to_speech.elevenlabs import ElevenLabsTextToSpeechDriver
from griptape.structures import Pipeline
from griptape.tasks import TextToSpeechTask
```

(continues on next page)

(continued from previous page)

```
driver = ElevenLabsTextToSpeechDriver(
    api_key=os.environ["ELEVEN_LABS_API_KEY"],
    model="eleven_multilingual_v2",
    voice="Matilda",
)

task = TextToSpeechTask(
    text_to_speech_driver=driver,
)

Pipeline(tasks=[task]).run("Generate audio from this text: Hello, world!")
```

Audio Transcription Task

This Task enables Structures to transcribe speech from text using *Audio Transcription Drivers* .

```
from griptape.drivers.audio_transcription.openai import OpenAiAudioTranscriptionDriver
from griptape.loaders import AudioLoader
from griptape.structures import Pipeline
from griptape.tasks import AudioTranscriptionTask

driver = OpenAiAudioTranscriptionDriver(model="whisper-1")

task = AudioTranscriptionTask(
    input=lambda _: AudioLoader().load("tests/resources/sentences2.wav"),
    audio_transcription_driver=driver,
)

Pipeline(tasks=[task]).run()
```

Workflows

Overview

A *Workflow* is a non-sequential DAG that can be used for complex concurrent scenarios with tasks having multiple inputs.

You can access the final output of the Workflow by using the *Structured Output* attribute.

Context

Workflows have access to the following *RAG Context* variables in addition to the *base context* :

- `task_outputs` : dictionary containing mapping of all task IDs to their outputs.
- `parent_outputs` : dictionary containing mapping of parent task IDs to their outputs.
- `parents_output_text` : string containing the concatenated outputs of all parent tasks.
- `parents` : dictionary containing mapping of parent task IDs to their task objects.
- `children` : dictionary containing mapping of child task IDs to their task objects.

Workflow

Let's build a simple workflow. Let's say, we want to write a story in a fantasy world with some unique characters. We could setup a workflow that generates a world based on some keywords. Then we pass the world description to any number of child tasks that create characters. Finally, the last task pulls in information from all parent tasks and writes up a short story.

```
from griptape.structures import Workflow
from griptape.tasks import PromptTask
from griptape.utils import StructureVisualizer

world_task = PromptTask(
    "Create a fictional world based on the following key words {{ keywords|join(, ) }}
↪",
    context={"keywords": ["fantasy", "ocean", "tidal lock"]},
    id="world",
)

def character_task(task_id: str, character_name: str) -> PromptTask:
    return PromptTask(
        "Based on the following world description create a character named {{ name }}
↪:\n{{ parent_outputs[world] }}",
        context={"name": character_name},
        id=task_id,
        parent_ids=["world"],
    )

scotty_task = character_task("scotty", "Scotty")
annie_task = character_task("annie", "Annie")

story_task = PromptTask(
    "Based on the following description of the world and characters, write a short␣
↪story:\n{{ parent_outputs[world] }}\n{{ parent_outputs[scotty] }}\n{{ parent_
↪outputs[annie] }}",
    id="story",
    parent_ids=["world", "scotty", "annie"],
)

workflow = Workflow(tasks=[world_task, story_task, scotty_task, annie_task, story_
↪task])

print(StructureVisualizer(workflow).to_url())

workflow.run()
```

Note that we use the `StructureVisualizer` to get a visual representation of the workflow. If we visit the printed url, it should look like this:

Fig. 1: Workflow

Declarative vs Imperative Syntax

The above example showed how to create a workflow using the declarative syntax via the `parent_ids` init param, but there are a number of declarative and imperative options for you to choose between. There is no functional difference, they merely exist to allow you to structure your code as is most readable for your use case. Possibilities are illustrated below.

Declaratively specify parents (same as above example):

```
from griptape.rules import Rule
from griptape.structures import Workflow
from griptape.tasks import PromptTask

workflow = Workflow(
    tasks=[
        PromptTask("Name an animal", id="animal"),
        PromptTask("Describe {{ parent_outputs[animal] }} with an adjective", id=
↪"adjective", parent_ids=["animal"]),
        PromptTask("Name a {{ parent_outputs[adjective] }} animal", id="new-animal",␣
↪parent_ids=["adjective"]),
    ],
    rules=[Rule("output a single lowercase word")],
)

workflow.run()
```

Declaratively specify children:

```
from griptape.rules import Rule
from griptape.structures import Workflow
from griptape.tasks import PromptTask

workflow = Workflow(
    tasks=[
        PromptTask("Name an animal", id="animal", child_ids=["adjective"]),
        PromptTask(
            "Describe {{ parent_outputs[animal] }} with an adjective", id="adjective",
↪ child_ids=["new-animal"]
        ),
        PromptTask("Name a {{ parent_outputs[adjective] }} animal", id="new-animal"),
    ],
    rules=[Rule("output a single lowercase word")],
)

workflow.run()
```

Declaratively specifying a mix of parents and children:

```
from griptape.rules import Rule
from griptape.structures import Workflow
from griptape.tasks import PromptTask

workflow = Workflow(
    tasks=[
        PromptTask("Name an animal", id="animal"),
```

(continues on next page)

(continued from previous page)

```
        PromptTask(
            "Describe {{ parent_outputs[animal] }} with an adjective",
            id="adjective",
            parent_ids=["animal"],
            child_ids=["new-animal"],
        ),
        PromptTask("Name a {{ parent_outputs[adjective] }} animal", id="new-animal"),
    ],
    rules=[Rule("output a single lowercase word")],
)

workflow.run()
```

Imperatively specify parents:

```
from griptape.rules import Rule
from griptape.structures import Workflow
from griptape.tasks import PromptTask

animal_task = PromptTask("Name an animal", id="animal")
adjective_task = PromptTask("Describe {{ parent_outputs[animal] }} with an adjective",
↪ id="adjective")
new_animal_task = PromptTask("Name a {{ parent_outputs[adjective] }} animal", id="new-
↪animal")

adjective_task.add_parent(animal_task)
new_animal_task.add_parent(adjective_task)

workflow = Workflow(
    tasks=[animal_task, adjective_task, new_animal_task],
    rules=[Rule("output a single lowercase word")],
)

workflow.run()
```

Imperatively specify children:

```
from griptape.rules import Rule
from griptape.structures import Workflow
from griptape.tasks import PromptTask

animal_task = PromptTask("Name an animal", id="animal")
adjective_task = PromptTask("Describe {{ parent_outputs[animal] }} with an adjective",
↪ id="adjective")
new_animal_task = PromptTask("Name a {{ parent_outputs[adjective] }} animal", id="new-
↪animal")

animal_task.add_child(adjective_task)
adjective_task.add_child(new_animal_task)

workflow = Workflow(
    tasks=[animal_task, adjective_task, new_animal_task],
```

(continues on next page)

(continued from previous page)

```
    rules=[Rule("output a single lowercase word")],
)

workflow.run()
```

Imperatively specify a mix of parents and children:

```
from griptape.rules import Rule
from griptape.structures import Workflow
from griptape.tasks import PromptTask

workflow = Workflow(
    rules=[Rule("output a single lowercase word")],
)

animal_task = PromptTask("Name an animal", id="animal", structure=workflow)
adjective_task = PromptTask(
    "Describe {{ parent_outputs[animal] }} with an adjective", id="adjective",␣
↪structure=workflow
)
new_animal_task = PromptTask("Name a {{ parent_outputs[adjective] }} animal", id="new-
↪animal", structure=workflow)

adjective_task.add_parent(animal_task)
adjective_task.add_child(new_animal_task)

workflow.run()
```

Or even mix imperative and declarative:

```
from griptape.rules import Rule
from griptape.structures import Workflow
from griptape.tasks import PromptTask

animal_task = PromptTask("Name an animal", id="animal")
adjective_task = PromptTask(
    "Describe {{ parent_outputs[animal] }} with an adjective", id="adjective", parent_
↪ids=["animal"]
)

new_animal_task = PromptTask("Name a {{ parent_outputs[adjective] }} animal", id="new-
↪animal")
new_animal_task.add_parent(adjective_task)

workflow = Workflow(
    tasks=[animal_task, adjective_task, new_animal_task],
    rules=[Rule("output a single lowercase word")],
)

workflow.run()
```

Insert Parallel Tasks

`Workflow.insert_tasks()` provides a convenient way to insert parallel tasks between parents and children.

Imperatively insert parallel tasks between a parent and child:

```
from griptape.rules import Rule
from griptape.structures import Workflow
from griptape.tasks import PromptTask

workflow = Workflow(
    rules=[Rule("output a single lowercase word")],
)

animal_task = PromptTask("Name an animal", id="animal")
adjective_task = PromptTask("Describe {{ parent_outputs[animal] }} with an adjective",
↪ id="adjective")
color_task = PromptTask("Describe {{ parent_outputs[animal] }} with a color", id=
↪"color")
new_animal_task = PromptTask("Name an animal described as: \n{{ parents_output_text }}
↪", id="new-animal")

#  The following workflow runs animal_task, then (adjective_task, and color_task)

#  in parallel, then finally new_animal_task.

#
#  In other words, the output of animal_task is passed to both adjective_task and␣
↪color_task

#  and the outputs of adjective_task and color_task are then passed to new_animal_
↪task.

workflow.add_task(animal_task)
workflow.add_task(new_animal_task)
workflow.insert_tasks(animal_task, [adjective_task, color_task], new_animal_task)

workflow.run()
```

Bitshift Composition

Task relationships can also be set up with the Python bitshift operators `>>` and `<<` . The following statements are all functionally equivalent:

```
task1 >> task2
task1.add_child(task2)

task2 << task1
task2.add_parent(task1)

task3 >> [task4, task5]
task3.add_children([task4, task5])
```

When using the bitshift to compose operators, the relationship is set in the direction that the bitshift operator points. For example, `task1 >> task2` means that `task1` runs first and `task2` runs second. Multiple operators can be composed

– keep in mind the chain is executed left-to-right and the rightmost object is always returned. For example:

```
task1 >> task2 >> task3 << task4
```

is equivalent to:

```
task1.add_child(task2)
task2.add_child(task3)
task3.add_parent(task4)
```

11.7 Tools in Gen AI Builder

11.7.1 Overview

One of the most powerful features of Gen AI Builder is the ability to use tools that interact with the outside world. Tools give the LLM abilities to invoke APIs, reference data sets, and expand their capabilities beyond simple text generation.

Many of our *Prompt Drivers* leverage the native function calling built into LLMs. For models that do not support this, Gen AI Builder provides its own implementation using the ReAct technique.

You can switch between these strategies by setting `use_native_tools` to `True` (LLM-native tool calling) or `False` (Gen AI Builder tool calling) on your *Prompt Drivers* .

11.7.2 Custom Tools

See *Custom Tools* for more information on building your own tools.

11.7.3 Tool Output and Task Memory

Output artifacts from all tool activities (except `InfoArtifact` and `ErrorArtifact`) are stored in short-term `TaskMemory` . To disable this behavior, set the `off_prompt` tool parameter to `False` .

11.7.4 Using Tools in Pipelines

Gen AI Builder provides a set of official tools for accessing and processing data. You can also *build your own tools* .

Here is an example of a Pipeline using tools:

```
from griptape.structures import Pipeline
from griptape.tasks import PromptTask
from griptape.tools import FileManagerTool, PromptSummaryTool, WebScraperTool

pipeline = Pipeline()

pipeline.add_tasks(
    PromptTask(
        "Load https://www.griptape.ai, summarize it, and store it in a file called␣
↪griptape.txt",
        tools=[
            WebScraperTool(off_prompt=True),
            FileManagerTool(off_prompt=True),
            PromptSummaryTool(off_prompt=False),
        ],
    ),
```

(continues on next page)

(continued from previous page)

```
)

pipeline.run()
```

Custom Tools

Overview

Building your own tools is easy with Gen AI Builder!

Tools are nothing more than Python classes that inherit from *BaseTool* . Each method in the class is decorated with an *process_output(tool_activity, subtask, output_artifact)* decorator which informs the LLM how and when it should use that Tool Activity.

Random Tool

Here is a simple Tool for performing various operations with the `random` module.

```
import random

from schema import Literal, Optional, Schema

from griptape.artifacts import TextArtifact
from griptape.artifacts.list_artifact import ListArtifact
from griptape.structures import Agent
from griptape.tools import BaseTool
from griptape.utils.decorators import activity

class RandomTool(BaseTool):
    @activity(
        config={
            "description": "Can be used to generate random numbers",
        }
    )
    def generate_rand_num(self) -> TextArtifact:
        """Generate a random number between 0 and 1.

        Returns:
            TextArtifact: The random number as a Text Artifact.
        """
        return TextArtifact(random.random())

    @activity(
        config={
            "description": "Can be used to generate random numbers",
            "schema": Schema(
                {
                    Literal("start", description="The start of the rand range,␣
↪inclusive."): int,
                    Literal("stop", description="The start of the rand range,␣
↪exclusive."): int,
                }
            ),
```

(continues on next page)

(continued from previous page)

```
        }
    )
    def generate_rand_range(self, start: int, stop: int) -> TextArtifact:
        """Generate a random number between start and stop.

        Args:
            start (int): The starting number.
            stop (int): The ending number.

        Returns:
            TextArtifact: The random number as a Text Artifact.
        """
        return TextArtifact(random.randrange(start, stop))

    @activity(
        config={
            "description": "Can be used to select a random item from a list",
            "schema": Schema(
                {
                    "items": [str],
                }
            ),
        }
    )
    def select_rand_item(self, *, values: dict) -> TextArtifact:
        """Select a random item from a list.

        Args:
            values (dict): The values declared by the schema.

        Returns:
            TextArtifact: The selected item as a Text Artifact.
        """
        items = values["items"]

        return TextArtifact(random.choice(items))

    @activity(
        config={
            "description": "Can be used to sample a list",
            "schema": Schema(
                {
                    "items": [str],
                    Optional("k"): int,
                }
            ),
        }
    )
    def sample_list(self, *, params: dict) -> ListArtifact[TextArtifact]:
        """Shuffle a list.

        Args:
```

(continues on next page)

(continued from previous page)

```
            params (dict): A dictionary containing a key, `values`, which contains␣
↪the values declared by the schema.

        Returns:
            TextArtifact: The sampled list as a List Artifact of Text Artifacts.
        """
        values = params["values"]

        items = values["items"]
        k = values.get("k", 5)

        sampled = random.sample(items, k)

        return ListArtifact([TextArtifact(item) for item in sampled])

agent = Agent(tools=[RandomTool()])

agent.run("Generate a number between 5 and 10, then generate that many animals,␣
↪sample 3, then randomly select one.")
```

Tool Activities

Activities are actions an LLM can perform with a various Tool. They pair a natural language description with some code to execute. Some examples:

- "Can be used to create a random number" -> `generate_rand_num`
- "Can be used to select a random item from a list" -> `select_rand_item`

Technically, each Activity is a method in the tool class that's decorated with the *process_output(tool_activity, subtask, output_artifact)* decorator.

Gen AI Builder will convert the Tool and its Activities into the appropriate format for the LLM to use. You can see the schema for a particular Activity by calling *to_json_schema* .

```
from rich.pretty import pprint

from griptape.tools import CalculatorTool

tool = CalculatorTool()

pprint(tool.to_activity_json_schema(tool.calculate, "Calculate Schema"))
```

Each Activity takes a `config` keyword argument that contains the configuration for the Activity. The configuration can contain:

- `description` (str): A plain text description of the Activity. Ensure that the description is clear and concise as it will help inform the LLM of when to pick this Activity.
- `schema` (optional): An optional instance of `Schema` that defines the input values to the Activity. This field should be omitted if the Activity does not accept any input values.

Activity Methods

Activity decorated methods should return an *Artifacts* , though Gen AI Builder will automatically convert any other return type to an *InfoArtifact* .

If an Activity's config declares a `schema` , the method should declare parameters using one of the following styles:

- Standard python keyword arguments. See `generate_random_number` .
- `values` (dict): A dictionary of the input values to the Activity. See `select_random_item` .
- `params` (dict): A dictionary that will contain a single key, `values` . See `sample_list` . Other keys may be added in the future, though generally, you should prefer using one of the other styles.

Warning

Do not name any schema fields as *values* or *params* . They are reserved for the Activity method signature.

Tool Dependencies

Each Tool can also have its own dependencies. You can specify them in a `requirements.txt` file in the tool directory and Gen AI Builder will install them during Tool execution. To start, create a directory for your Tool inside your project. The directory must have the following structure:

```
griptape/tools/calculator/
├── __init__.py
├── requirements.txt # file with tool Python dependencies
└── tool.py # file with tool Python class
```

That's it! Import and use your Tool in your project as you would with any other Gen AI Builder Tool.

Check out other *Gen AI Builder Tools* to learn more about tool implementation details.

Official Tools

Overview

Gen AI Builder provides a set of official Tools for common tasks. In general, Gen AI Builder-included Tools are designed to be general-purpose and Driver-based for easy integration with different backends.

Tools

Audio Transcription

This Tool enables *Agents* to transcribe speech from text using *Audio Transcription Drivers* .

```
from griptape.drivers.audio_transcription.openai import OpenAiAudioTranscriptionDriver
from griptape.structures import Agent
from griptape.tools.audio_transcription.tool import AudioTranscriptionTool

driver = OpenAiAudioTranscriptionDriver(model="whisper-1")

tool = AudioTranscriptionTool(
    off_prompt=False,
    audio_transcription_driver=driver,
```

(continues on next page)

(continued from previous page)

```
)

Agent(tools=[tool]).run(
    "Transcribe the following audio file: /Users/andrew/code/griptape/tests/resources/
↪sentences2.wav"
)
```

Calculator

This tool enables LLMs to make simple calculations.

```
from griptape.structures import Agent
from griptape.tools import CalculatorTool

#  Create an agent with the CalculatorTool tool

agent = Agent(tools=[CalculatorTool()])

#  Run the agent with a task to perform the arithmetic calculation of \(10^5\)

agent.run("What is 10 raised to the power of 5?")
```

Computer

This tool enables LLMs to execute Python code and run shell commands inside a Docker container. You have to have the Docker daemon running in order for this tool to work.

You can specify a local working directory and environment variables during tool initialization:

```
from griptape.structures import Agent
from griptape.tools import ComputerTool

#  Initialize the ComputerTool tool

computer = ComputerTool()

#  Create an agent with the ComputerTool tool

agent = Agent(tools=[computer])

agent.run("Make 2 files and then list the files in the current directory")
```

Date Time

This tool enables LLMs to get current date and time.

```
from griptape.structures import Agent
from griptape.tools import DateTimeTool

agent = Agent(tools=[DateTimeTool()])

agent.run("What day is 5 days past christmas in 2026?")
```

Email

The *EmailTool* enables LLMs to send emails.

```
import os

from griptape.tools import EmailTool

email_tool = EmailTool(
    smtp_host=os.environ.get("SMTP_HOST"),
    smtp_port=int(os.environ.get("SMTP_PORT", "465")),
    smtp_password=os.environ.get("SMTP_PASSWORD"),
    smtp_user=os.environ.get("FROM_EMAIL"),
    smtp_use_ssl=bool(os.environ.get("SMTP_USE_SSL")),
)
```

For debugging purposes, you can run a local SMTP server that the LLM can send emails to:

```
python -m smtpd -c DebuggingServer -n localhost:1025
```

Extraction

The *ExractionTool* enables LLMs to extract structured text from unstructured data.

```
import schema

from griptape.engines import JsonExtractionEngine
from griptape.structures import Agent
from griptape.tools import ExtractionTool, WebScraperTool

agent = Agent(
    input="Load {{ args[0] }} and extract key info",
    tools=[
        WebScraperTool(off_prompt=True),
        ExtractionTool(
            off_prompt=False,
            extraction_engine=JsonExtractionEngine(
                template_schema=schema.Schema(
                    {
                        "company_name": str,
                        "industry": str,
                        schema.Literal(
                            "product_features",
                            description="List of key product features.",
                        ): list[str],
                    }
                ).json_schema("Company Info"),
            ),
        ),
    ],
)
agent.run("https://griptape.ai")
```

File Manager

This tool enables LLMs to save and load files.

```
from pathlib import Path

from griptape.structures import Agent
from griptape.tools import FileManagerTool

#  Initialize the FileManagerTool tool with the current directory as its base

file_manager_tool = FileManagerTool()

#  Add the tool to the Agent

agent = Agent(tools=[file_manager_tool])

#  Directly create a file named sample1.txt with some content

filename = "sample1.txt"
content = "This is the content of sample1.txt"

Path(filename).write_text(content)

#  Now, read content from the file sample1.txt using the agents command

agent.run("Can you get me the sample1.txt file?")
```

`

Gen AI Builder Tool

The *GriptapeCloudToolTool* integrates with Gen AI Builder's hosted Tools.

Note: This tool requires a Tool hosted in Gen AI Builder and an API Key for access.

```
import os

from griptape.structures import Agent
from griptape.tools.griptape_cloud_tool.tool import GriptapeCloudToolTool

agent = Agent(
    tools=[
        GriptapeCloudToolTool(  # Tool is configured as a random number generator
            tool_id=os.environ["GT_CLOUD_TOOL_ID"],
        )
    ]
)
agent.run("Generate a number between 1 and 10")
```

Image Query

This tool allows Agents to execute natural language queries on the contents of images using multimodal models.

```
from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.structures import Agent
from griptape.tools import ImageQueryTool

driver = OpenAiChatPromptDriver(model="gpt-4.1")

#  Create an Image Query Tool configured to use the engine.

tool = ImageQueryTool(
    prompt_driver=driver,
)

#  Create an agent and provide the tool to it.

Agent(tools=[tool]).run("Describe the weather in the image tests/resources/mountain.
↪png in one word.")
```

Inpainting Image Generation

This tool allows LLMs to generate images using inpainting, where an input image is altered within the area specified by a mask image according to a prompt. The input and mask images can be provided either by their file path or by their *Task Memory and Off Prompt* references.

```
from griptape.drivers.image_generation.amazon_bedrock import␣
↪AmazonBedrockImageGenerationDriver
from griptape.drivers.image_generation_model.bedrock_stable_diffusion import (
    BedrockStableDiffusionImageGenerationModelDriver,
)
from griptape.structures import Agent
from griptape.tools import InpaintingImageGenerationTool

#  Create a driver configured to use Stable Diffusion via Bedrock.

driver = AmazonBedrockImageGenerationDriver(
    image_generation_model_driver=BedrockStableDiffusionImageGenerationModelDriver(),
    model="stability.stable-diffusion-xl-v1",
)

#  Create a tool configured to use the engine.

tool = InpaintingImageGenerationTool(
    image_generation_driver=driver,
)

#  Create an agent and provide the tool to it.

Agent(tools=[tool]).run(
    "Generate an image of a castle built into the side of a mountain by inpainting␣
↪the "
```

(continues on next page)

(continued from previous page)

```
    "image at tests/resources/mountain.png using the mask at tests/resources/mountain-
↪mask.png."
)
```

Outpainting Image Generation

This tool allows LLMs to generate images using outpainting, where an input image is altered outside of the area specified by a mask image according to a prompt. The input and mask images can be provided either by their file path or by their *Task Memory and Off Prompt* references.

```
from griptape.drivers.image_generation.amazon_bedrock import␣
↪AmazonBedrockImageGenerationDriver
from griptape.drivers.image_generation_model.bedrock_stable_diffusion import (
    BedrockStableDiffusionImageGenerationModelDriver,
)
from griptape.structures import Agent
from griptape.tools import OutpaintingImageGenerationTool

#  Create a driver configured to use Stable Diffusion via Bedrock.

driver = AmazonBedrockImageGenerationDriver(
    image_generation_model_driver=BedrockStableDiffusionImageGenerationModelDriver(),
    model="stability.stable-diffusion-xl-v1",
)

#  Create a tool configured to use the engine.

tool = OutpaintingImageGenerationTool(
    image_generation_driver=driver,
)

#  Create an agent and provide the tool to it.

Agent(tools=[tool]).run(
    "Generate an image of a mountain shrouded by clouds by outpainting the "
    "image at tests/resources/mountain.png using the mask at tests/resources/mountain-
↪mask.png."
)
```

Prompt Image Generation

This tool allows LLMs to generate images from a text prompt.

```
from griptape.drivers.image_generation.amazon_bedrock import␣
↪AmazonBedrockImageGenerationDriver
from griptape.drivers.image_generation_model.bedrock_stable_diffusion import (
    BedrockStableDiffusionImageGenerationModelDriver,
)
from griptape.structures import Agent
from griptape.tools import PromptImageGenerationTool
```

(continues on next page)

(continued from previous page)

```
#  Create a driver configured to use Stable Diffusion via Bedrock.

driver = AmazonBedrockImageGenerationDriver(
    image_generation_model_driver=BedrockStableDiffusionImageGenerationModelDriver(),
    model="stability.stable-diffusion-xl-v1",
)

#  Create a tool configured to use the engine.

tool = PromptImageGenerationTool(
    image_generation_driver=driver,
)

#  Create an agent and provide the tool to it.

Agent(tools=[tool]).run("Generate an image of a mountain on a summer day.")
```

Prompt Summary

The *PromptSummaryTool* enables LLMs summarize text data.

```
from griptape.structures import Agent
from griptape.tools import PromptSummaryTool, WebScraperTool

agent = Agent(tools=[WebScraperTool(off_prompt=True), PromptSummaryTool()])

agent.run(
    "How can I build Neovim from source for MacOS according to this https://github.
↪com/neovim/neovim/blob/master/BUILD.md"
)
```

`

Rag

The *RagTool* enables LLMs to query modular RAG engines.

Here is an example of how it can be used with a local vector store driver:

```
from griptape.artifacts import TextArtifact
from griptape.drivers.embedding.openai import OpenAiEmbeddingDriver
from griptape.drivers.prompt.openai import OpenAiChatPromptDriver
from griptape.drivers.vector.local import LocalVectorStoreDriver
from griptape.engines.rag import RagEngine
from griptape.engines.rag.modules import PromptResponseRagModule,␣
↪VectorStoreRetrievalRagModule
from griptape.engines.rag.stages import ResponseRagStage, RetrievalRagStage
from griptape.structures import Agent
from griptape.tools import RagTool

vector_store_driver = LocalVectorStoreDriver(embedding_driver=OpenAiEmbeddingDriver())
```

(continues on next page)

(continued from previous page)

```
artifact = TextArtifact(
    "Griptape builds AI-powered applications that connect securely to your enterprise␣
↪data and APIs."
    "Griptape Agents provide incredible power and flexibility when working with large␣
↪language models."
)

vector_store_driver.upsert(artifact, namespace="griptape")

rag_tool = RagTool(
    description="Contains information about Griptape",
    off_prompt=False,
    rag_engine=RagEngine(
        retrieval_stage=RetrievalRagStage(
            retrieval_modules=[
                VectorStoreRetrievalRagModule(
                    vector_store_driver=vector_store_driver, query_params={"namespace
↪": "griptape", "top_n": 20}
                )
            ]
        ),
        response_stage=ResponseRagStage(
            response_modules=[PromptResponseRagModule(prompt_
↪driver=OpenAiChatPromptDriver(model="gpt-4.1"))]
        ),
    ),
)

agent = Agent(tools=[rag_tool])

agent.run("what is Griptape?")
```

Query

The *ImageQueryTool* enables Agents to query unstructured data for specific information.

```
from griptape.structures import Agent
from griptape.tools import QueryTool, WebScraperTool

agent = Agent(tools=[WebScraperTool(off_prompt=True), QueryTool()])

agent.run("Tell me about the architecture as described here: https://neovim.io/doc/
↪user/vim_diff.html")
```

Rest Api

This tool enables LLMs to call REST APIs.

The *RestApiTool* tool uses the following parameters:

The following example is built using https://jsonplaceholder.typicode.com/guide/.

My First Post, A Journey.,

This tool enables LLMs to execute SQL statements via SQLAlchemy . Depending on your underlying SQL engine, configure your `engine_url` and give the LLM a hint about what engine you are using via `engine_name` , so that it can create engine-specific statements.

```
import os

import boto3

from griptape.drivers.sql.amazon_redshift import AmazonRedshiftSqlDriver
from griptape.loaders import SqlLoader
from griptape.structures import Agent
from griptape.tools import SqlTool

session = boto3.Session()

sql_loader = SqlLoader(
    sql_driver=AmazonRedshiftSqlDriver(
        database=os.environ["REDSHIFT_DATABASE"],
        session=session,
        cluster_identifier=os.environ["REDSHIFT_CLUSTER_IDENTIFIER"],
    )
)

sql_tool = SqlTool(
    sql_loader=sql_loader,
    table_name="people",
    table_description="contains information about tech industry professionals",
    engine_name="redshift",
)

agent = Agent(tools=[sql_tool])
agent.run("SELECT * FROM people;")
```

The *StructureRunTool* Tool provides a way to run Structures via a Tool. It requires you to provide a *Structure Run Drivers* to run the Structure in the desired environment.

```
import os

from griptape.drivers.structure_run.griptape_cloud import␣
↪GriptapeCloudStructureRunDriver
from griptape.structures import Agent
from griptape.tools import StructureRunTool

base_url = os.environ["GT_CLOUD_BASE_URL"]
api_key = os.environ["GT_CLOUD_API_KEY"]
structure_id = os.environ["GT_CLOUD_STRUCTURE_ID"]

structure_run_tool = StructureRunTool(
    description="RAG Expert Agent - Structure to invoke with natural language queries␣
↪about the topic of Retrieval Augmented Generation",
    structure_run_driver=GriptapeCloudStructureRunDriver(
        base_url=base_url,
```

(continues on next page)

(continued from previous page)

```
        api_key=api_key,
        structure_id=structure_id,
    ),
)

#  Set up an agent using the StructureRunTool tool

agent = Agent(tools=[structure_run_tool])

#  Task: Ask the Griptape Cloud Hosted Structure about modular RAG

agent.run("what is modular RAG?")
```

This Tool enables LLMs to synthesize speech from text using *Text to Speech Drivers* .

```
import os

from griptape.drivers.text_to_speech.elevenlabs import ElevenLabsTextToSpeechDriver
from griptape.structures import Agent
from griptape.tools.text_to_speech.tool import TextToSpeechTool

driver = ElevenLabsTextToSpeechDriver(
    api_key=os.environ["ELEVEN_LABS_API_KEY"],
    model="eleven_multilingual_v2",
    voice="Matilda",
)

tool = TextToSpeechTool(
    text_to_speech_driver=driver,
)

Agent(tools=[tool]).run("Generate audio from this text: Hello, world!")
```

This Tool allows LLMs to generate variations of an input image from a text prompt. The input image can be provided either by its file path or by its *Task Memory and Off Prompt* reference.

```
from griptape.drivers.image_generation.amazon_bedrock import␣
↪AmazonBedrockImageGenerationDriver
from griptape.drivers.image_generation_model.bedrock_stable_diffusion import (
    BedrockStableDiffusionImageGenerationModelDriver,
)
from griptape.structures import Agent
from griptape.tools import VariationImageGenerationTool

#  Create a driver configured to use Stable Diffusion via Bedrock.

driver = AmazonBedrockImageGenerationDriver(
    image_generation_model_driver=BedrockStableDiffusionImageGenerationModelDriver(
        style_preset="pixel-art",
    ),
    model="stability.stable-diffusion-xl-v1",
)
```

(continues on next page)

(continued from previous page)

```
#  Create a tool configured to use the engine.

tool = VariationImageGenerationTool(
    image_generation_driver=driver,
)

#  Create an agent and provide the tool to it.

Agent(tools=[tool]).run(
    "Generate a variation of the image located at tests/resources/mountain.png␣
↪depicting a mountain on a winter day"
)
```

```
from griptape.drivers.image_generation.amazon_bedrock import␣
↪AmazonBedrockImageGenerationDriver
from griptape.drivers.image_generation_model.bedrock_stable_diffusion import (
    BedrockStableDiffusionImageGenerationModelDriver,
)
from griptape.structures import Agent
from griptape.tools import FileManagerTool, PromptImageGenerationTool,␣
↪VariationImageGenerationTool

#  Create a driver configured to use Stable Diffusion via Bedrock.

driver = AmazonBedrockImageGenerationDriver(
    image_generation_model_driver=BedrockStableDiffusionImageGenerationModelDriver(
        style_preset="pixel-art",
    ),
    model="stability.stable-diffusion-xl-v1",
)

#  Create a prompt image generation client configured to use the driver.

prompt_tool = PromptImageGenerationTool(image_generation_driver=driver, off_
↪prompt=True)

#  Create a variation image generation client configured to use the driver.

variation_tool = VariationImageGenerationTool(
    image_generation_driver=driver,
    off_prompt=True,
)

file_tool = FileManagerTool()

#  Create an agent and provide the tools to it.

agent = Agent(tools=[prompt_tool, variation_tool, file_tool])

#  Run the agent using a prompt motivating it to generate an image, then
```

(continues on next page)

(continued from previous page)

```
#  create a variation of the image present in task memory.

agent.run(
    "Generate an image of a mountain on a summer day. Then, generate a "
    "variation of this image depicting the same mountain scene on a winter day."
    "Save them images to disk and return the file paths. "
)
```

The *VectorStoreTool* enables LLMs to query vector stores.

Here is an example of how it can be used with a local vector store driver:

```
from griptape.chunkers import TextChunker
from griptape.drivers.embedding.openai import OpenAiEmbeddingDriver
from griptape.drivers.vector.local import LocalVectorStoreDriver
from griptape.loaders import WebLoader
from griptape.structures import Agent
from griptape.tools import VectorStoreTool

NAMESPACE = "griptape"

vector_store_driver = LocalVectorStoreDriver(
    embedding_driver=OpenAiEmbeddingDriver(),
)

artifacts = WebLoader().load("https://www.griptape.ai")
chunks = TextChunker().chunk(artifacts)

vector_store_driver.upsert_collection({NAMESPACE: chunks})
vector_db = VectorStoreTool(
    description="This DB has information about the Griptape Python framework",
    vector_store_driver=vector_store_driver,
    query_params={"namespace": NAMESPACE},
)

agent = Agent(tools=[vector_db])

agent.run("what is Griptape?")
```

This tool enables LLMs to scrape web pages for full text, summaries, authors, titles, and keywords. It can also execute search queries to answer specific questions about the page. This tool uses OpenAI APIs for some of its activities, so in order to use it provide a valid API key in `openai_api_key` .

```
from griptape.structures import Agent
from griptape.tools import PromptSummaryTool, WebScraperTool

agent = Agent(tools=[WebScraperTool(off_prompt=True), PromptSummaryTool(off_
↪prompt=False)])

agent.run("Based on https://www.griptape.ai/, tell me what griptape is")
```

This tool enables LLMs to search the web.

```
import os

from griptape.drivers.web_search.google import GoogleWebSearchDriver
from griptape.structures import Agent
from griptape.tools import WebSearchTool

#  Initialize the WebSearchTool tool with necessary parameters

web_search_tool = WebSearchTool(
    web_search_driver=GoogleWebSearchDriver(
        api_key=os.environ["GOOGLE_API_KEY"],
        search_id=os.environ["GOOGLE_API_SEARCH_ID"],
        results_count=5,
        language="en",
        country="us",
    ),
)

#  Set up an agent using the WebSearchTool tool

agent = Agent(tools=[web_search_tool])

#  Task: Search the web for a specific query

agent.run("Tell me how photosynthesis works")
```

Extra schema properties can be added to the Tool to allow for more customization if the Driver supports it. In this example, we add a `sort` property to the `search` Activity which will be added as a *Google custom search query parameter* .

```
import os

import schema

from griptape.drivers.web_search.google import GoogleWebSearchDriver
from griptape.structures import Agent
from griptape.tools import WebSearchTool

agent = Agent(
    tools=[
        WebSearchTool(
            web_search_driver=GoogleWebSearchDriver(
                api_key=os.environ["GOOGLE_API_KEY"],
                search_id=os.environ["GOOGLE_API_SEARCH_ID"],
            ),
            extra_schema_properties={
                "search": {
                    schema.Literal(
                        "sort",
                        description="Date range to search within. Format:␣
↪date:r:YYYYMMDD:YYYYMMDD",
                    ): str
                }
```

(continues on next page)

(continued from previous page)

```
            },
        )
    ],
)

agent.run("Search for articles about the history of the internet from 1990 to 2000")
```