



AIDB

Release 7.6.0

EnterpriseDB Corporation

Aug 27, 2026

1	Introduction	3
1.1	Where to start	3
1.2	Documentation map	4
2	AIDB 7.x release notes	5
2.1	AIDB 7.6.0 release notes	5
2.2	AIDB 7.5.0 release notes	6
2.3	AIDB 7.4.0 release notes	7
2.4	AIDB 7.3.2 release notes	8
2.5	AIDB 7.3.0 release notes	9
2.5.1	Deprecated interfaces	10
2.6	AIDB 7.2.0 release notes	11
3	AIDB overview	13
3.1	What AIDB does	13
3.2	AIDB as a standalone extension	14
3.3	Key capabilities	14
3.3.1	Known issues	14
4	Compatibility	15
4.1	AIDB	15
4.1.1	Supported platforms	15
4.1.2	Supported Postgres versions	15
4.2	PGFS	15
5	Installing and configuring AIDB	17
5.1	Next steps	17
5.1.1	Configuring AIDB	17
5.1.2	Installing AIDB	24
5.1.3	Threading limits	25
6	Integrating models	27
6.1	Local models	27
6.1.1	Default models	28
6.1.2	Model providers	28
6.1.3	Custom models	29
6.1.4	Validating a model at creation	36

6.1.5	Local model downloads	36
6.1.6	Using a registered model	37
6.1.7	Troubleshooting	38
6.2	External model connections	38
6.2.1	Validating a model at creation	38
6.2.2	OpenAI-compatible endpoints	39
6.2.3	OpenAI Responses API	41
6.2.4	Anthropic Messages API	42
6.2.5	NVIDIA NIM	43
6.2.6	Google Gemini	45
6.2.7	OpenRouter	46
6.3	Support matrix	47
6.3.1	Local models	47
6.3.2	Remote models	49
7	Configuring external storage	53
7.1	How it works	53
7.2	Step 1: Creating a storage location	53
7.2.1	S3-compatible object store	54
7.2.2	Local file system	54
7.3	Step 2: Creating a volume	54
7.3.1	Path handling	55
7.3.2	Example: PDFs in S3	55
7.3.3	Example: Images in a local directory	55
7.4	Step 3: Using the volume as a pipeline source	56
7.5	Managing volumes	56
7.5.1	Listing volumes	56
7.5.2	Deleting a volume	56
7.6	Accessing volume contents with SQL	56
7.6.1	Listing objects in a volume	57
7.6.2	Reading a single object	57
7.6.3	Writing, overwriting, and deleting objects	57
8	Agents	59
8.1	Example: a text-to-SQL agent	59
8.1.1	Setup	59
8.1.2	Conversation	61
8.1.3	Comparison	61
8.2	Documentation map	61
8.2.1	Managing agents	62
8.2.2	Invoking agents	65
8.2.3	Delegation	67
9	Tools	69
9.1	Documentation map	70
9.1.1	Native tools	70
9.1.2	Custom SQL tools	74
9.1.3	MCP tools	75
10	AI pipelines	79
10.1	Example: a knowledge base for RAG	79
10.1.1	Set up the pipeline	79
10.1.2	Run the pipeline	80
10.1.3	Retrieve: semantic search	80

11 Knowledge bases	129
11.1 Retrieval functions	129
11.1.1 Flow of retrieval functions	129
11.1.2 <code>aidb.retrieve_text()</code>	130
11.1.3 <code>aidb.retrieve_key()</code>	130
11.2 Advanced querying: Joining intermediate steps	130
11.2.1 Example syntax	130
11.3 Deduplication and intermediate results	131
12 Knowledge bases	133
12.1 Multi-pipeline knowledge bases	133
12.1.1 Attaching a pipeline to an existing KB	133
12.1.2 Compatibility rules	134
12.1.3 Deletion semantics	135
12.1.4 Inspecting attached pipelines	135
12.1.5 Vector table schema	135
12.2 Hybrid search	136
12.2.1 Prerequisites	136
12.2.2 Volume source limitation	136
12.3 Hybrid search	136
12.3.1 Next steps	136
12.4 Examples	144
12.4.1 Example 1: Bulk embedding from a Postgres table	144
12.4.2 Example 2: Bulk embedding from an S3 bucket (PGFS)	145
13 Semantic knowledge bases	147
13.1 Semantic KB vs. vector knowledge base	147
13.2 What a semantic KB indexes	147
13.3 Create and manage a semantic KB	148
13.3.1 The single-KB shortcut	148
13.4 Keeping a KB current	148
13.5 How you use a semantic KB	149
13.6 In this section	149
13.7 Related documentation	149
13.7.1 Searching a semantic knowledge base	149
13.7.2 Semantic aliases	152
13.7.3 Text-to-SQL	155
13.7.4 Example	157
14 SQL functions	161
14.1 Text chunking	162
14.1.1 Configuration options	162
14.2 Data parsing	163
14.2.1 Parse HTML	163
14.2.2 Parse PDF	164
14.3 Performing OCR	164
14.4 Text summarization	167
14.4.1 <code>aidb.summarize_text()</code>	167
14.5 Text summarization	167
14.6 Text summarization	167
14.7 Text summarization	168
14.7.1 Summarization use cases	168
14.7.2 Summarization cookbook	171
14.8 AI inference functions	183

14.8.1	aidb.encode_text()	184
14.8.2	aidb.encode_text_batch()	184
14.8.3	aidb.encode_image()	184
14.8.4	aidb.generate_text()	185
14.8.5	aidb.generate_text_batch()	185
14.8.6	aidb.inference_config()	186
14.9	Text reranking	186
14.9.1	Using reranking after knowledge base retrieval	187
15	Reference	189
15.1	Models reference	189
15.1.1	Configuration model	190
15.1.2	Catalog views	190
15.1.3	Model management functions	191
15.1.4	HCP model functions	194
15.1.5	Inference functions	196
15.1.6	Inference configuration helper	199
15.1.7	Model config helpers	202
15.2	Knowledge bases reference	211
15.2.1	Catalog views	211
15.2.2	Types	212
15.2.3	Knowledge base functions	213
15.2.4	Volume management functions	216
15.3	Operations reference	217
15.3.1	Error log functions	217
15.3.2	Error categories	221
15.4	Functions reference	221
15.4.1	aidb.chunk_text	221
15.4.2	aidb.parse_html	222
15.4.3	aidb.parse_pdf	223
15.4.4	aidb.perform_ocr	224
15.4.5	aidb.summarize_text	224
15.4.6	aidb.summarize_text_aggregate	225
15.4.7	Parameter defaults quick reference	226
15.5	Pipelines reference	227
15.6	Agents reference	227
15.6.1	Catalog views	227
15.6.2	Agent management functions	229
15.6.3	Conversation functions	231
15.6.4	Structured output helpers	233
15.6.5	Types	234
15.7	Tools reference	235
15.7.1	Catalog views	235
15.7.2	Tool management functions	236
15.7.3	Parameter helpers	239

AIDB is an EDB-maintained PostgreSQL extension that brings AI data workflows directly into your Postgres database. Define pipelines in SQL to parse, chunk, embed, and index your data — then query it with semantic search, hybrid search, and LLM-powered retrieval. Common use cases include retrieval-augmented generation (RAG), semantic search over documents and images, and automated document ingestion pipelines from S3 or local storage.

1.1 Where to start

If you're new to AIDB, follow this end-to-end workflow:

1. ***Knowledge base overview*** — Learn the core concepts: what pipelines and knowledge bases are, and how they fit together.
2. ***Installing AIDB*** — Install the `aidb` package, configure `shared_preload_libraries`, and enable the extension in your database.
3. ***Integrating models*** — Register the AI model that will power your pipeline. Every pipeline step that generates embeddings or processes text requires a model — either a default local model (no setup needed) or an external API model that you register with `aidb.create_model()`. Do this before creating pipelines.
4. ***Configuring external storage*** (optional) — If your source data lives in S3 or a local file system rather than a Postgres table, configure a volume using PGFS so pipelines can read from it.
5. ***Agents reference*** (optional) — Build an in-database agent that reasons over multiple steps and calls *Tools reference* — AIDB's own operations, your own SQL queries, or tools imported from an external MCP server.
6. ***AI pipelines*** — Define a pipeline that reads from your data source, transforms it (chunk, parse, embed), and writes the output to a knowledge base. This is where the model you registered in step 3 is put to work.
7. ***Knowledge bases reference*** — Query the vector embeddings your pipeline produced. Use `aidb.retrieve_text()` or `aidb.retrieve_key()` for semantic search, or combine vector and keyword search with hybrid search.
8. ***Semantic knowledge bases*** (optional) — Index your schema — tables, views, columns, and comments — for natural-language schema search and text-to-SQL, so questions map to the right relations and to reusable, parameterized queries.

1.2 Documentation map

Section	What it covers
<i>AIDB 7.2.0 release notes</i>	What changed in each version
<i>Compatibility</i>	Supported platforms and Postgres versions for AIDB and PGFS
<i>Installing AIDB</i>	How to install and configure the AIDB extension
<i>Integrating models</i>	Local models, external APIs (OpenAI, NIM, Gemini, OpenRouter), and supported model variants
<i>Configuring external storage</i>	Using S3-compatible object stores and local file systems as pipeline data sources
<i>Agents reference</i>	Building in-database agents, invoking them, and delegation
<i>Tools reference</i>	The unified tool catalog: native tools, custom SQL tools, and MCP tools
<i>AI pipelines</i>	Defining pipelines, pipeline steps, orchestration, and examples
<i>Knowledge bases reference</i>	Semantic search, hybrid search, and vector index options
<i>Semantic knowledge bases</i>	Schema-metadata search and text-to-SQL: semantic KBs, semantic aliases, and agent tools
<i>SQL functions</i>	Standalone SQL functions for chunking, parsing, OCR, summarization, and embedding
<i>AI pipelines</i>	Full API reference for all AIDB functions and views

AIDB 7.x release notes

Release notes for AIDB major version 7.

Version	Release date
<i>AIDB 7.6.0 release notes</i>	26 Aug 2026
<i>AIDB 7.5.0 release notes</i>	29 Jul 2026
<i>AIDB 7.4.0 release notes</i>	06 Jul 2026
<i>AIDB 7.3.2 release notes</i>	17 Apr 2026
<i>AIDB 7.3.0 release notes</i>	08 Apr 2026
<i>AIDB 7.2.0 release notes</i>	30 Mar 2026

2.1 AIDB 7.6.0 release notes

Released: 26 Aug 2026

AIDB 7.6.0 introduces **Agents and tools**, a SQL-native framework for building in-database AI agents: an agent reasons over multiple steps — calling tools, observing their results, and repeating — instead of answering in one shot. Tools come from a unified catalog (`aidb.tools`) spanning a large built-in native tool set, your own custom SQL queries, and tools imported from an external MCP server. Agents support per-call budgets (tokens, reasoning iterations, wall-clock time), delegation to other agents, and per-user conversation visibility. See *Agents reference* and *Tools reference* to get started.

AIDB 7.6.0 includes the following enhancements and bug fixes:

Type	Description
Enhancement	Agents and tools: The new SQL-native agent framework and unified tools hub described above. See <i>Agents reference</i> and <i>Tools reference</i> .

continues on next page

Table 2 – continued from previous page

Type	Description
Enhancement	Native tool-calling model adapters: The new <i>openai_responses</i> and <i>anthropic_messages</i> providers (plus <i>openai_responses_azure</i> and <i>anthropic_messages_azure/anthropic_messages_bedrock</i> hosting variants) connect to OpenAI's Responses API and Anthropic's Messages API natively, emitting real <i>tools/tool_choice</i> request fields and parsing real tool-call responses, instead of simulating tool calls through prompt injection. See External model connections .
Enhancement	Model credentials from an environment variable: <i>aidb.create_model()</i> accepts a new <i>credentials_env</i> argument, letting credentials be read from a server-side environment variable at model-use time instead of passed inline as JSON.
Enhancement	Local reranking via llama.cpp: The new <i>llamacpp_reranking</i> provider reranks text locally from a GGUF file, supporting both cross-encoder models (for example, <i>bge-reranker-v2-m3</i>) and the Qwen3-Reranker decoder-only family — no external API required. See Local models .
Enhancement	Local OCR via llama.cpp: The new <i>llamacpp_ocr</i> provider runs OCR locally from a GGUF vision model, with a pre-registered default model (<i>lightonocr-2-1b-Q8_0</i>). See Local models .
Enhancement	Renamed `decode_text` to `generate_text`: <i>aidb.decode_text()</i> and <i>aidb.decode_text_batch()</i> are renamed to <i>aidb.generate_text()</i> and <i>aidb.generate_text_batch()</i> , matching <i>encode_text()</i> 's naming on the embedding side. The old names still work but are deprecated and will be removed in a future version. See Inference functions .
Enhancement	Semantic KB search across sources: The new <i>aidb.semantic_kb_search()</i> function fuses ranked vector search results from schema metadata and curated semantic aliases into a single result set using Reciprocal Rank Fusion.
Enhancement	Simplified semantic KB calls: <i>kb_name</i> (and <i>name</i>) are now optional on <i>aidb.get_column_definitions()</i> , <i>aidb.get_metadata()</i> , <i>aidb.get_entity_definitions()</i> , <i>aidb.search_by_comment()</i> , <i>aidb.create_semantic_kb()</i> , <i>aidb.delete_semantic_kb()</i> , <i>aidb.refresh_semantic_kb()</i> , <i>aidb.update_semantic_kb_auto_processing()</i> , and <i>aidb.semantic_kb_stats()</i> — when omitted, AIDB resolves the single existing semantic KB.
Change	Semantic aliases can belong to multiple KBs: <i>aidb.create_semantic_alias()</i> and related functions drop the <i>model</i> argument; an alias is now embedded once per knowledge base that owns the schema its SQL reads, instead of being bound to exactly one KB and model.
Enhancement	Pipeline metrics show last run time: <i>aidb.get_pipeline_metrics()</i> and the pipeline metrics views now include a <i>last_run_completed</i> timestamp.
Change	Leaked credential detection: <i>aidb.create_model()</i> now rejects a <i>config</i> that embeds <i>api_key</i> or <i>basic_auth</i> directly — pass them via <i>credentials/credentials_env</i> instead. The new <i>aidb.audit_leaked_credentials()</i> function reports any pre-existing models where credentials were detected embedded in their config.
Bug fix	Credentials with invalid header characters: An <i>api_key</i> or <i>basic_auth</i> value containing characters that aren't valid in an HTTP header (for example, a stray newline) now fails with a clear configuration error instead of crashing the backend.

2.2 AIDB 7.5.0 release notes

Released: 29 Jul 2026

AIDB 7.5.0 includes the following enhancements and bug fixes:

Type	Description
Enhancement	Model validation on creation: <i>aidb.create_model()</i> now validates a model by default, running a small probe inference — downloading and loading local models, or sending a test request to external providers — so configuration and credential problems surface at creation time instead of on first use. Pass <i>validate => false</i> to skip it, or use the new <i>aidb.validate_model()</i> to check an existing model. See Integrating models .
Enhancement	Local LLM inference with llama.cpp: AIDB can now run GGUF-format models locally through the new <i>llamacpp_generate</i> and <i>llamacpp_embeddings</i> providers, alongside the existing Candle-based local models. Ships with 7 pre-registered default models (5 for embeddings, 2 for generation). See Local models .
Enhancement	Requeuing failed pipeline records: The new <i>aidb.requeue_pipeline_errors()</i> function marks record-level error log entries dirty for reprocessing on the pipeline's next run, without needing to touch source data. See Requeuing failed records .
Enhancement	Live-tunable model thread limit: The new <i>aidb.max_threads</i> parameter caps CPU threads used for local model inference (Candle and llama.cpp), and can be changed live without a restart. See Threading limits .
Enhancement	I/O thread pool control: The new <i>aidb.max_io_threads</i> parameter caps the async I/O thread pool used for remote-model HTTP calls and volume/object-store access, live-tunable without a restart. See Threading limits .
Enhancement	llama.cpp native log control: The new <i>aidb.enable_llamacpp_logs</i> parameter turns on llama.cpp's own diagnostic logging in the server log, for troubleshooting model loading and inference issues. Off by default; requires a restart to take effect. See AI pipelines .
Enhancement	Clearer error for pipeline column/step type mismatches: <i>aidb.create_pipeline()</i> now rejects a source column whose SQL type doesn't match its configured step (for example, a <i>text</i> column feeding a <i>KnowledgeBase(Image)</i> step) with a specific error at creation time, instead of failing later with an opaque internal error.
Change	Metrics collector is now opt-in: The background worker that materializes pipeline metrics into <i>aidb.pipeline_metrics_collection</i> is disabled by default. Enable it with the new <i>aidb.enable_background_metrics_collector</i> setting, instance-wide or per database. See Materialized metrics .
Change	`topk` must be a positive integer: <i>aidb.retrieve_key()</i> and <i>aidb.retrieve_text()</i> now reject <i>topk</i> values less than 1 with a clear error, instead of silently clamping to 1.
Bug fix	Error log functions and Empty-source pipelines: <i>aidb.get_error_logs()</i> , <i>aidb.get_error_log_summary()</i> , and <i>aidb.get_all_error_summaries()</i> no longer raise an error for pipelines with an Empty source (such as Semantic KB) — they return empty results instead, and no longer abort results for other pipelines in the same call.

2.3 AIDB 7.4.0 release notes

Released: 06 Jul 2026

AIDB 7.4.0 includes the following enhancements and bug fixes:

Type	Description
Enhancement	Volumes as queryable foreign tables: Each AIDB volume is now exposed as a PGFS-backed foreign table, so the objects it contains can be listed, read, written, and deleted with standard SQL — including <i>INSERT</i> , <i>UPDATE</i> , and <i>DELETE</i> on objects, and <i>SELECT body FROM my_volume WHERE key = '...'</i> to fetch a single object. See Configuring external storage .
Enhancement	Egress allowlist: Optional outbound HTTP allowlist for model calls and model downloads, configured via the <i>aidb.egress_allowlist</i> parameter (with <i>edb.egress_allowlist</i> as a shared fallback). Accepts host and CIDR entries; when unset, no check is applied.
Enhancement	More reliable HuggingFace model downloads: Downloads now resume across unstable connections, verify size and SHA-256, emit progress notices, and respond to Ctrl-C between retries. Corrupt cached snapshots self-heal on the next load.
Enhancement	Strict pipeline option validation: Unknown keys and wrong types in <i>step_N_options</i> , knowledge-base configs, and standalone helpers (<i>chunk_text</i> , <i>summarize_text</i> , <i>parse_html</i> , <i>parse_pdf</i> , <i>pdf_to_image</i> , <i>perform_ocr</i>) now raise a clear error instead of being silently dropped.
Enhancement	Clearer model error messages: HTTP errors from model providers now include the request URL. Large input values are truncated in error output. OCR responses are only parsed as JSON on HTTP 200.
Enhancement	Batch summarization: <i>SummarizeText</i> pipeline steps now process records in batches, with per-input retry on rate-limit errors.
Enhancement	Clearer permission errors on pipeline creation: Creating a pipeline against a source table without <i>SELECT</i> privilege now returns an explicit "permission denied" error instead of an opaque internal failure.
Bug fix	Empty retrieve returns empty result: <i>retrieve_text</i> on an empty knowledge base now returns no rows with a warning instead of raising an error, matching standard SQL <i>SELECT</i> behavior.
Bug fix	'retrieve_text' value column on non-text sources: Retrieval from pipelines whose original source is non-text (such as PDF or Image) now returns <i>NULL</i> in <i>value</i> instead of the raw bytes decoded as text. Match metadata is still returned.
Bug fix	Deadlock with metrics collector: The background metrics collector no longer holds locks on every user table in a single transaction, removing a deadlock against <i>delete pipeline</i> .
Bug fix	BIGINT pipeline keys: Pipelines on source tables with <i>BIGINT</i> primary keys above the <i>int4</i> range no longer fail with an "out of range for type integer" error.
Bug fix	Pipeline batch size respected across steps: Steps that explode records (such as chunking) now sub-batch their outputs before passing them to the next step, keeping memory and parallelism within the configured <i>batch_size</i> .
Bug fix	OCR config validation: OCR model configurations are now validated at model creation time.

2.4 AIDB 7.3.2 release notes

Released: 17 Apr 2026

AIDB 7.3.2 includes the following enhancements and bug fixes:

Type	Description
Enhancement	Pipeline DLQ — Phase 3: Metrics and error log API: Expanded the Dead Letter Queue with a metrics and error log API, providing programmatic access to per-pipeline error counts, error details, and historical error logs.
Enhancement	Multi-user pipeline PGD trigger wrappers: Added multi-user aware trigger wrappers for PGD pipelines, ensuring that pipeline auto-processing triggers work correctly in multi-user PGD environments.
Enhancement	Knowledge base tables in PGD replication setup: Knowledge base tables are now automatically included in PGD replication configuration during setup, ensuring knowledge base data is replicated across PGD nodes.
Enhancement	Improved PDF parsing: Updated the <i>lopdf</i> library to a newer version with improved structural PDF parsing, resulting in better text extraction from complex and malformed PDF documents.
Enhancement	HTTP 429 retry handling: AIDB now automatically delays and retries requests that receive an HTTP 429 (Too Many Requests) response from model provider APIs, improving resilience when API rate limits are hit.
Enhancement	Deduplicate retrieve results by source key: Retrieve operations now deduplicate results based on the source key, preventing duplicate records from appearing in retrieval output when the same source document is indexed multiple times or referenced across multiple pipeline runs.
Bug fix	Fix handling of wrong input type errors: Input type validation errors are now caught and surfaced correctly at the input stage rather than propagating as unexpected errors downstream.
Bug fix	Fix multi-pipeline ID bug: Corrected an issue where pipeline IDs were incorrectly assigned in multi-pipeline configurations, which could cause pipelines to be misidentified or executed incorrectly.
Bug fix	Fix auto-processing state bug: Resolved a state management issue in the pipeline auto-processing system that could cause pipelines to enter or remain in an incorrect processing state.

2.5 AIDB 7.3.0 release notes

Released: 08 Apr 2026

AIDB 7.3.0 includes the following enhancements and bug fixes:

Type	Description
Enhancement	Pipeline DLQ — Phase 2: Nonblocking errors: The Dead Letter Queue now handles nonblocking errors, meaning that transient or non-critical processing failures no longer halt pipeline execution. Failed records are queued for inspection and reprocessing.
Enhancement	Transient error classification: Model and database errors are now classified as <i>PipelineTemporary</i> where appropriate, enabling the pipeline to retry transient failures automatically rather than treating them as permanent errors.
Enhancement	DLQ user APIs: Introduced user-facing SQL functions for interacting with the Dead Letter Queue, including the ability to list, inspect, and reprocess failed pipeline records.
Enhancement	Comments in semantic knowledge base search: Semantic knowledge base search now includes support for searching over comments stored alongside document content.

2.5.1 Deprecated interfaces

AIDB 7.3.0 removes the following SQL views and functions as part of the major version interface cleanup. Replacements are noted for each.

Deprecated objects	Replacement
Views	
<i>aidb.preparers</i>	<i>aidb.pipelines</i> (filter by operation type if needed)
<i>aidb.preps</i>	<i>aidb.pipelines</i>
<i>aidb.retrievers</i>	<i>aidb.knowledge_bases</i> or <i>aidb.kbs</i>
<i>aidb.preparer_registry</i>	<i>aidb.pipeline_registry</i> (internal)
<i>aidb.preparer_registry_source_table</i>	<i>aidb.pipeline_registry</i> (internal)
<i>aidb.preparer_registry_source_volume</i>	<i>aidb.pipeline_registry</i> (internal)
Functions	
<i>aidb.create_table_knowledge_base(...)</i>	<i>aidb.create_pipeline(... step_1 => 'KnowledgeBase')</i>
<i>aidb.create_volume_knowledge_base(...)</i>	<i>aidb.create_pipeline(... step_1 => 'KnowledgeBase')</i>
<i>aidb.set_auto_knowledge_base(...)</i>	<i>aidb.update_pipeline(name, auto_processing => ...)</i>
<i>aidb.assert_knowledge_base_preconditions(...)</i>	Removed — no-op; validation is now handled internally
<i>aidb.create_retriever_for_table(...)</i>	<i>aidb.create_pipeline(... step_1 => 'KnowledgeBase')</i>
<i>aidb.create_retriever_for_volume(...)</i>	<i>aidb.create_pipeline(... step_1 => 'KnowledgeBase')</i>
<i>aidb.delete_retriever(...)</i>	<i>aidb.delete_pipeline(...)</i>
<i>aidb.set_retriever_auto_processing(...)</i>	<i>aidb.update_pipeline(name, auto_processing => ...)</i>
<i>aidb.enable_auto_embedding_for_table(...)</i>	<i>aidb.update_pipeline(name, auto_processing => 'Live')</i>
<i>aidb.disable_auto_embedding_for_table(...)</i>	<i>aidb.update_pipeline(name, auto_processing => 'Disabled')</i>
<i>aidb.create_table_preparer(...)</i>	<i>aidb.create_pipeline(...)</i>
<i>aidb.create_volume_preparer(...)</i>	<i>aidb.create_pipeline(...)</i>
<i>aidb.create_preparer_for_table(...)</i>	<i>aidb.create_pipeline(...)</i>
<i>aidb.create_preparer_for_volume(...)</i>	<i>aidb.create_pipeline(...)</i>
<i>aidb.delete_preparer(...)</i>	<i>aidb.delete_pipeline(...)</i>
<i>aidb.set_preparer_auto_processing(...)</i>	<i>aidb.update_pipeline(name, auto_processing => ...)</i>
<i>aidb.get_preparer_data_source(...)</i>	Query <i>aidb.pipelines</i> view directly
<i>aidb.assert_preparer_preconditions(...)</i>	Removed — validation is now handled internally
<i>aidb.assert_table_preparer_preconditions(...)</i>	Removed
<i>aidb.assert_volume_preparer_preconditions(...)</i>	Removed
<i>aidb.bulk_embedding(name)</i>	<i>aidb.run_pipeline(name)</i>
<i>aidb.bulk_data_preparation(name)</i>	<i>aidb.run_pipeline(name)</i>
<i>aidb.single_embedding(name, key, text)</i>	<i>aidb.run_pipeline(name)</i> or direct model calls
<i>aidb.single_embedding(name, key, bytea)</i>	<i>aidb.run_pipeline(name)</i> or direct model calls
<i>aidb.set_auto_preparer(name, mode)</i>	<i>aidb.update_pipeline(name, auto_processing => mode)</i>
<i>aidb.upsert_chunk_text(name, key, value)</i>	Use pipeline execution via <i>aidb.run_pipeline()</i>
<i>aidb.upsert_summarize_text(name, key, value)</i>	Use pipeline execution via <i>aidb.run_pipeline()</i>
<i>aidb.upsert_parse_html(name, key, value)</i>	Use pipeline execution via <i>aidb.run_pipeline()</i>
<i>aidb.upsert_parse_pdf(name, key, value)</i>	Use pipeline execution via <i>aidb.run_pipeline()</i>
<i>aidb.upsert_perform_ocr(name, key, value)</i>	Use pipeline execution via <i>aidb.run_pipeline()</i>

2.6 AIDB 7.2.0 release notes

Released: 30 Mar 2026

AIDB 7.2.0 includes the following enhancements and bug fixes:

Type	Description
Enhancement	Multi-pipeline knowledge bases: Knowledge bases can now be associated with multiple pipelines, enabling richer and more flexible data ingestion and retrieval architectures.
Enhancement	Hybrid search (BYOSQL): Introduced Tier 1 hybrid search support with a bring-your-own-SQL (BYOSQL) interface, allowing custom retrieval logic to be combined with semantic search.
Enhancement	Retrieval support for multistep pipelines: Retrieval operations are now supported within multistep pipeline configurations, enabling end-to-end RAG workflows across pipeline stages.
Enhancement	HuggingFace API support: Added support for the HuggingFace Inference API as a model provider for embeddings and other AI tasks.
Enhancement	Model options: Model configurations can now include provider-specific options, giving finer control over model behavior such as temperature, max tokens, and other parameters.
Enhancement	Pipeline Dead Letter Queue (DLQ) — Phase 1: Introduced the first phase of Dead Letter Queue support for pipelines. Processing errors are now logged for failed pipeline operations, enabling inspection and reprocessing of failed records.
Enhancement	Configurable CPU limit: Added a configurable CPU limit for background pipeline workers to prevent resource contention in shared environments.
Enhancement	HCP model sync hints: When listing models that require HCP synchronization, AIDB now displays a helpful hint guiding users on how to sync models from HCP.
Enhancement	Optionally versioned interfaces (OVI): Pipeline and knowledge base views now support optionally versioned interfaces, allowing clients to pin to a specific API version.
Enhancement	Improved deprecation UX: Deprecated functions now provide clearer messages with guidance on replacement functions and migration steps.
Enhancement	`functions` tag on `aidb.models` view: The <i>aidb.models</i> view now includes a <i>functions</i> tag, making it easier to identify models that support function-calling capabilities.
Bug fix	Fix volume listing with directories: Corrected an issue where volume listing failed when directories were present alongside files in a volume.
Change	Volume names with quoted identifiers blocked: Creating volumes with names that require quoted identifiers (for example, names containing spaces or special characters) is no longer allowed, to prevent ambiguity in SQL references.

CHAPTER 3

AIDB overview

AIDB is an EDB-maintained Postgres extension that brings AI data workflows directly into your Postgres database. It provides a declarative, SQL-native way to build pipelines that prepare, embed, and index data for AI applications — including retrieval-augmented generation (RAG), semantic search, and large language model (LLM) integrations.

Because AIDB runs inside Postgres, your data never needs to leave the database. You define your AI workflows in SQL, and AIDB handles the complexity of embedding generation, vector indexing, model integration, and real-time synchronization.

3.1 What AIDB does

AIDB has four primary building blocks: **pipelines**, **knowledge bases**, **agents**, and **tools**.

**** *Creating pipelines* **** define multi-step workflows that transform raw data into AI-ready vectors. Each step in a pipeline handles one transformation — parsing a PDF, chunking text, running OCR, summarizing content, or producing vector embeddings. Pipelines can run on demand, in batch, or automatically whenever source data changes.

**** *Knowledge bases reference* **** are the output of an embedding pipeline step. They store vector embeddings in a pgvector-indexed table and expose semantic search through the `aidb.retrieve_key()` and `aidb.retrieve_text()` functions. Multiple pipelines can feed into the same knowledge base, and a single knowledge base can serve multiple retrieval queries.

**** *Agents reference* **** let a model reason over multiple steps instead of answering in one shot: an agent calls tools, observes their results, and repeats until it has an answer. Agents can also delegate part of a request to another agent.

**** *Tools reference* **** are what an agent calls to take action — AIDB’s own built-in operations (querying a pipeline’s status, searching a knowledge base, running a SQL query), your own custom SQL queries, or operations imported from an external MCP server.

All pipeline operations are also available as standalone **SQL functions** — `aidb.chunk_text()`, `aidb.parse_pdf()`, `aidb.perform_ocr()`, `aidb.summarize_text()`, and others — so you can call them directly in queries without defining a full pipeline.

3.2 AIDB as a standalone extension

AIDB is a Postgres extension — it lives entirely inside your database and has no runtime dependency on Agent Factory or Hybrid Manager (HM). You install it like any other Postgres extension, configure it via `postgresql.conf`, and interact with it through SQL.

!!!note If you're using Agent Factory, AIDB is already included as part of that platform, but Agent Factory isn't required to use AIDB. This documentation covers standalone AIDB.

3.3 Key capabilities

Model flexibility — AIDB connects to locally running models (such as BERT and T5) as well as external models via the OpenAI-compatible API. Supported providers include NVIDIA NIM, Google Gemini, and any OpenAI-compatible endpoint. Model credentials are managed securely through the standard Postgres user mapping feature — or, to avoid storing a secret in the database at all, name an environment variable to read it from instead. See [Credentials from environment variables](#).

Data source support — Pipelines can read from standard Postgres tables or from [Configuring external storage](#) via the Postgres File System (PGFS) extension, which supports object store and local file systems. Pipeline output can be written to Postgres tables or back to a volume using PGFS.

Auto-processing — Pipelines can be configured to run automatically using Postgres trigger-based live mode or background worker-based scheduling. This keeps knowledge bases up to date as source data changes without manual intervention.

Schema flexibility — All AIDB objects (pipelines, knowledge bases, volumes) can be created in arbitrary Postgres schemas, making it straightforward to organize AI workloads alongside existing database structures.

Vector index control — Knowledge bases support configurable vector indexes, including HNSW and IVFFlat, via pgvector.

3.3.1 Known issues

- **Limited air-gapped support.** The built-in CPU model runtimes (Candle and llama.cpp) run fully offline once their model files are available locally, but any pipeline step or agent tool that calls out to a remote model provider still needs outbound network access to that provider. See [Configuring AIDB](#).

4.1 AIDB

4.1.1 Supported platforms

Platform	x86/64	ARM64	ppc64le
Ubuntu 22.04 LTS	OK	—	—
Ubuntu 24.04 LTS	OK	—	—
Debian 12 (Bookworm)	OK	OK	—
RHEL 8	OK	—	—
RHEL 9	OK	OK	OK (Postgres 18 only)

SLES, earlier Debian versions, and non-Linux platforms are not currently supported.

4.1.2 Supported Postgres versions

AIDB supports Postgres versions 14–18, including EDB Postgres Advanced Server, EDB Postgres Extended, and community PostgreSQL.

4.2 PGFS

AIDB pipelines that read from cloud storage (S3, GCS, Azure) require the PGFS extension. For platform and Postgres version compatibility, see *Compatibility*.

Installing and configuring AIDB

AIDB is designed to be easy to install and configure. Follow the steps below to get started with AIDB in your project.

- *Installing AIDB*
- *Configuring AIDB*
- *Threading limits*

5.1 Next steps

Once AIDB is installed and configured, register the models that will power your pipelines and SQL functions — see *Integrating models* .

5.1.1 Configuring AIDB

To use AIDB, add it to `shared_preload_libraries` , create the extensions in the database, manage user access, and verify the installation. This page also covers proxy settings and the AIDB configuration parameters (GUCs) you can tune afterward.

Configure `shared_preload_libraries`

1. In the `postgresql.conf` file, add `aidb` to the `shared_preload_libraries` parameter:

```
shared_preload_libraries = aidb
```

!!!note If `shared_preload_libraries` already has other extensions listed, append `aidb` using a comma separator. The order doesn't matter. .. ::

1. Restart Postgres.

Create the extensions

1. Create the AIDB extension in your database:

```
CREATE EXTENSION aidb CASCADE;
```

The `CASCADE` option automatically installs the required `vector` extension if it isn't already present.

!!!Note PGD users If you're running EDB Postgres Distributed (PGD), run the following after creating the extension to register AIDB catalog tables in the replication set:

```
SELECT aidb.bdr_setup();
```

```
CREATE EXTENSION pgfs;
```

For more information, see PGFS documentation .

Manage user access with the `aidb_users` role

When you create the AIDB extension, Postgres automatically creates a role named `aidb_users` . Grant a user membership in this role to give it access to AIDB's SQL functions. This role isn't removed if you later drop the extension.

`aidb_users` is designed to simplify access management for AIDB features:

- **Purpose:** Granting a user the `aidb_users` role gives it access to all AIDB functions, so it can use the extension's capabilities.
- **Security:** The `aidb_users` role **doesn't** grant access to AIDB's internal tables, so only the extension's public API is exposed to users who hold it.
- **Background workers:** When background workers are used, they execute as the user who defined them, or as a role specified by the user, provided that user has access to that role.

To let a user use AIDB functions, grant them the `aidb_users` role:

```
GRANT aidb_users TO your_user;
```

For example, create a role named `alice` with no privileges or role membership to start with:

```
CREATE ROLE alice
LOGIN
PASSWORD change_me;  -- replace with a real password
```

Then add `alice` to the `aidb_users` role:

```
GRANT aidb_users TO alice;
```

`alice` now inherits whatever privileges `aidb_users` has, for example `EXECUTE` on routines in the `aidb` schema.

Membership in `aidb_users` doesn't grant access to your own data. Give `alice` (not `aidb_users`) the privileges it needs on the tables AIDB will read or write. This example grants access to three arbitrary tables that would ordinarily be source tables used in a pipeline:

```
GRANT SELECT, INSERT, UPDATE, DELETE
ON public.documents,
   public.email_text,
   public.chats
TO alice;
```

If `alice` also needs to create objects in the `public` schema:

```
GRANT CREATE ON SCHEMA public TO alice;
```

!!!note Using AIDB through Pipeline Designer If you access AIDB through Hybrid Manager's Pipeline Designer, pipeline operations run under a separate role, `visual_pipeline_user`, which must also be granted membership in `aidb_users`. See [VPU and permissions](#) for that setup.

Proxy settings

If your environment routes outbound HTTP traffic through a proxy, set the `HTTP_PROXY` and `HTTPS_PROXY` environment variables in the Postgres environment file. On Ubuntu with community Postgres, this file is at `/etc/postgresql/<version>/main/environment`:

```
echo "HTTP_PROXY = http://<your-proxy-settings>/" | sudo tee -a /etc/postgresql/16/
  ↳main/environment
echo "HTTPS_PROXY = http://<your-proxy-settings>/" | sudo tee -a /etc/postgresql/16/
  ↳main/environment
```

Then restart Postgres for the changes to take effect:

```
sudo systemctl restart postgresql@16-main
```

Replace `<your-proxy-settings>` with your proxy address, and `16` with your Postgres version. Consult the documentation for your Postgres distribution for the exact location of the environment file.

!!!note AIDB has limited air-gapped support. The built-in CPU model runtimes (Candle and `llama.cpp`) run fully offline once their model files are available locally, but any pipeline step or agent tool that calls out to a remote model provider still needs outbound network access to that provider.

Configuration parameters

AIDB exposes a set of Postgres configuration parameters (GUCs) in the `aidb.*` namespace. Set them in `postgresql.conf`, per-session with `SET`, or per-connection with `-c`. Most parameters below are `USERSET` — any role can change them for its own session; parameters with a different context are called out individually.

Threading limits

Parameter	Type	Default	Range	Description
<code>aidb.max_threads</code>	integer	<code>0</code>	0–1024	Maximum number of CPU threads used for local model inference (Candle and <code>llama.cpp</code>). <code>0</code> uses the default: half of the available CPUs. <i>SUSET</i> .

continues on next page

Table 1 – continued from previous page

Parameter	Type	Default	Range	Description
<i>aidb.max_io_threads</i>	integer	0	0–1024	Maximum number of threads for AIDB's async I/O run-times (remote model HTTP calls, and volume/object-store I/O). 0 falls back to the <i>TOKIO_WORKER_THREADS</i> environment variable, or Tokio's own default (one per CPU) if that isn't set either. <i>SUSET</i> .

Both apply live: a session-local `SET` takes effect immediately in that backend only. Setting either with `ALTER SYSTEM` or in `postgresql.conf` takes effect after `pg_reload_conf()` (or `SIGHUP`), and propagates to all backends and background workers — no restart required.

aidb.max_threads

```
SET aidb.max_threads = 4;
```

aidb.max_io_threads

```
SET aidb.max_io_threads = 4;
```

See [Threading limits](#) for what these two parameters control, how they interact, and when to lower them.

Model download

The following parameters control how AIDB downloads local model files from HuggingFace (see [Local models](#) for applicable providers).

Parameter	Type	Default	Range	Description
<i>aidb.download_log_level</i>	string	<i>notice</i>	—	Controls the output channel for routine download progress messages (for example, download start/complete, byte-progress heartbeats, and adapter load lines). Valid values: <i>notice</i> , <i>log</i> , <i>off</i> . See below.
<i>aidb.download_max_attempts</i>	integer	30	1–500	Maximum total download attempts per file (initial request plus retries) before AIDB aborts. See Local model downloads for the retry behavior.

aidb.download_log_level

Controls where routine download-progress messages are sent. Warnings and errors are unaffected.

Value	Behavior
<i>notice</i>	Send messages to the client at <i>NOTICE</i> level. Useful for interactive sessions in <i>psql</i> .
<i>log</i>	Send messages to the server log only (subject to <i>log_min_messages</i>). Use this for batch jobs and tests where the client output needs to be stable across cold and warm caches.
<i>off</i>	Suppress routine progress messages entirely.

Example:

```
SET aidb.download_log_level = log;
```

aidb.download_max_attempts

Caps the total number of download attempts per file. The downloader aborts early when consecutive attempts make zero progress, so this parameter is primarily a safety ceiling against excessive retry loops.

Lower it (for example, to 3 or 5) for fail-fast behavior in CI or scripted batch jobs. Raise it only for very large files on lossy links.

Example:

```
SET aidb.download_max_attempts = 5;
```

llama.cpp native logs

Parameter	Type	Default	Range	Description
<i>aidb.enable_llamacpp_logs</i>	boolean	<i>false</i>	—	Controls whether llama.cpp's own diagnostic output (GGUF metadata, tensor shapes, and backend/model initialization) is written to the server log. <i>SUSET</i> . Read once at extension initialization — see below.

aidb.enable_llamacpp_logs

Off by default: llama.cpp's native logging is verbose — hundreds of lines per model load — so it's suppressed unless you need it. This is separate from `aidb.download_log_level` above, which controls AIDB's own download-progress messages; the two are independent.

Unlike the other parameters on this page, `aidb.enable_llamacpp_logs` is *SUSET* (a superuser, or a role granted *SET* on the parameter, can change it) and is read once when the extension initializes. Setting it with *SET*, *ALTER SYSTEM*, or editing `postgresql.conf` and reloading (`pg_reload_conf()`) doesn't take effect until Postgres is restarted:

```
aidb.enable_llamacpp_logs = true
```

```
sudo systemctl restart postgresql@16-main
```

Applies only to `llamacpp_generate` and `llamacpp_embeddings` models — it has no effect on Candle-based models (`bert_local`, `clip_local`, `t5_local`, `llama_instruct_local`). See [Local models — Troubleshooting](#) for when to use this.

Network egress

The following parameter restricts outbound HTTP traffic from AIDB — both calls to model provider APIs and model file downloads from HuggingFace. When unset, no restriction is applied and AIDB can reach any host.

Parameter	Type	Default	Description
<code>aidb.egress_allowlist</code>	string	unset	Comma-separated list of hosts and/or CIDR ranges that AIDB is allowed to reach. When set, any outbound request to a destination not on the list is rejected before the connection is made. <i>SUSET</i> .
<code>edb.egress_allowlist</code>	string	unset	Shared fallback allowlist used by all EDB extensions when the per-extension <code>aidb.egress_allowlist</code> is not set. Useful when AIDB and other EDB extensions (such as PGFS) share the same network policy. <i>SUSET</i> .

Entry format

Each comma-separated entry is one of:

Form	Matches
<code>host.example.com</code>	Exact hostname match (case-insensitive).
<code>.example.com</code>	Any subdomain of <code>example.com</code> (doesn't match <code>example.com</code> itself).
<code>203.0.113.10</code>	Exact IPv4 or IPv6 literal in the destination URL.
<code>203.0.113.0/24</code>	IPv4 CIDR range; matches IP literals in that range. Doesn't match hostnames.
<code>2001:db8::/32</code>	IPv6 CIDR range.

Whitespace around entries is ignored. CIDR entries are checked only against IP literals — hostnames aren't resolved for matching.

Resolution order

For each outbound request, AIDB checks `aidb.egress_allowlist` first. If it isn't set, AIDB falls back to `edb.egress_allowlist`. If neither is set, no check is applied. Redirects (for example, HuggingFace redirecting to a CDN host) are re-checked against the same allowlist; a redirect to a host not on the list is rejected.

Example

Allow only direct calls to OpenAI and HuggingFace plus its CDN:

```
aidb.egress_allowlist = api.openai.com, huggingface.co, .huggingface.co, cdn-lfs.
↳huggingface.co
```

When a request is rejected, AIDB raises an error that names the active parameter so the operator can adjust the list.

!!!note PGFS implements the same allowlist mechanism under `pgfs.egress_allowlist`, applied to traffic from PGFS to cloud object-store endpoints (S3, Google Cloud Storage, Azure Blob). `edb.egress_allowlist` is the shared fallback for both extensions: set it to apply one list to AIDB and PGFS together, or use the per-extension parameters when each needs a different policy. See [PGFS — Network egress](#).

If your AIDB pipelines read from external storage, both allowlists are relevant: AIDB's controls model-provider and HuggingFace traffic, and PGFS's controls the connection to your object store. See [Configuring external storage](#) for how AIDB and PGFS work together.

`aidb.create_model(credentials_env => ...)` and `aidb.import_mcp_tools(headers_env => ...)` read a Postgres backend process environment variable, named by the caller, instead of storing a credential in the database. The following parameter restricts which variable names may be referenced this way:

Parameter	Type	Default	Description
<code>aidb.env_var_allowed_prefix</code>	string	<code>AIDB_</code>	Required prefix for environment variable names read via <code>credentials_env/headers_env</code> . Set to an empty string to disable the restriction. Only a superuser may change it. <i>SUSET</i> .

Combined with a caller-controlled provider or MCP server URL, an unrestricted variable name would let a malicious configuration read out unrelated secrets that happen to live in the Postgres process's environment — this parameter closes that off by default while staying overridable per deployment. See [Credentials from environment variables](#) for `credentials_env`, and [MCP tools](#) for `headers_env`.

Validate the installation

Confirm that the extensions are installed by running `\dx` in psql and looking for `aidb`, `vector`, `pgfs` (if installed), in the output:

```
aidb=# \dx
__OUTPUT__
```

List of installed extensions			
Name	Version	Schema	Description
aidb	7.6.0	aidb	aidb: makes it easy to build AI applications with postgres
pgfs	3.3.0	pgfs	pgfs: enables access to filesystem-like storage locations
vector	0.8.0	public	vector data type and ivfflat and hnsw access methods

5.1.2 Installing AIDB

AIDB is a standalone PostgreSQL extension — it installs directly on your Postgres instance with no dependency on Agent Factory or Hybrid Manager. The steps below walk you through installing it as a standalone package.

!!!note If you're deploying through Agent Factory, AIDB is bundled and configured as part of that setup. Refer to the Agent Factory installation guide instead.

AIDB is supported on the same platforms as the Postgres distribution you're using. For details, see:

- [PostgreSQL product compatibility](#)
- [EDB Postgres Advanced Server product compatibility](#)
- [EDB Postgres Extended Server product compatibility](#)

Prerequisites

Before you begin the installation process:

1. Set up the repository: Setting up the repository is a one-time task. If you've already set up your repository, you don't need to perform this step. To set up the repository, go to [EDB repositories](#) and follow the instructions provided there.
2. Install your flavor of Postgres if not already installed. See:
 - [Installing PostgreSQL](#)
 -

[Installing EDB Postgres Advanced Server](#)

•
[Installing EDB Postgres Extended Server](#)

3. Install the pgvector extension:

AIDB relies on the pgvector extension for vector data management.

For installation instructions, see [Installing pgvector](#) .

Install the package

The syntax for the package install command is:

```
# For Debian/Ubuntu:

sudo apt-get install edb-<postgres><postgres_version>-aidb

# For Linux:

sudo dnf install edb-<postgres><postgres_version>-aidb
```

Where:

- <postgres> is the distribution of Postgres you're using:

Postgres distribution	Value
PostgreSQL	pg
EDB Postgres Advanced Server (EPAS)	as
EDB Postgres Extended	pgextended

- `<postgres_version>` is the version of Postgres you're using.

For example, to install AIDB for EDB Postgres Advanced Server 17 on Debian/Ubuntu:

```
sudo apt-get install edb-as17-aidb
```

For example, to install AIDB for PostgreSQL 16 on RHEL8:

```
sudo dnf install edb-pg16-aidb
```

After installing the AIDB package, see [Configuring AIDB](#) for the next steps.

5.1.3 Threading limits

AIDB runs local model inference and network/volume I/O on internal thread pools that, by default, size themselves from the number of CPUs available on the host. On a database server shared with other workloads, or one that's resource-constrained (for example, a small container), an unbounded thread pool can consume more CPU than you want to give it.

Two parameters cap these thread pools independently, depending on which kind of work you need to limit.

Local model inference threads

`aidb.max_threads` caps the CPU thread pool AIDB uses for local model inference — both runtimes: `Candle` (`bert_local`, `clip_local`, `t5_local`, `llama_instruct_local`) and `llama.cpp` (`llamacpp_generate`, `llamacpp_embeddings`). See [Local models](#) for the full list of local providers.

More threads generally means faster inference, at the cost of higher CPU usage. On a host running many concurrent pipelines, or one with a small CPU allocation, an unbounded pool can starve other backends of CPU time.

The default, 0, uses half of the available CPUs. Set a specific number to cap it:

```
SET aidb.max_threads = 4;
```

Async I/O threads

`aidb.max_io_threads` caps AIDB's async I/O thread pools — used for calling remote model providers over HTTP (OpenAI-compatible endpoints, NVIDIA NIM, Gemini, and OpenRouter) and for volume/object-store I/O against PGFS-backed volumes.

This is a separate resource from local model inference: it governs network and file I/O concurrency, not CPU-bound computation. A pipeline with many concurrent external-API calls, or a volume scan touching many objects at once, can open more concurrent I/O than you want on a host with limited network bandwidth or file descriptors.

The default, 0, falls back to the `TOKIO_WORKER_THREADS` environment variable if set, or otherwise to one thread per CPU (the underlying Tokio runtime's own default). Set a specific number to cap it:

```
SET aidb.max_io_threads = 4;
```

How to apply a change

Both parameters accept the same three ways to set them:

- **Per session**, with `SET` — applies immediately, to that backend only. Useful for testing a limit before rolling it out.
- **Per connection**, with `-c aidb.max_threads=4` on the connection string or in `PGOPTIONS`.
- **Instance-wide**, with `ALTER SYSTEM SET aidb.max_threads = 4;` followed by `SELECT pg_reload_conf();`, or by editing `postgresql.conf` directly and reloading (`SIGHUP`). This propagates to every backend and background worker without a restart.

For example, to cap both instance-wide on a shared host:

```
ALTER SYSTEM SET aidb.max_threads = 4;
ALTER SYSTEM SET aidb.max_io_threads = 4;
SELECT pg_reload_conf();
```

See *Threading limits* for the parameter reference (type, default, range).

Integrating models

Models are the AI engines that power AIDB’s core operations — embedding generation, text chunking, summarization, OCR, and completions. Every pipeline step and SQL function that performs an AI operation requires a model to be registered and reachable before it can run.

When to add models: Register your models after installing and configuring AIDB, and before creating pipelines or knowledge bases that reference them. If you’re using one of AIDB’s default local models (such as BERT or CLIP), no registration is needed — they’re available immediately. For external APIs (OpenAI, NVIDIA NIM, Gemini) or custom HuggingFace variants, use `aidb.create_model()` to register the model first.

Page	What it covers
<i>Local models</i>	Default local models (BERT, CLIP, T5, Llama), model providers, registering custom local models, and llama.cpp GGUF models
<i>External model connections</i>	OpenAI-compatible APIs, the native OpenAI Responses and Anthropic Messages APIs, NVIDIA NIM, Google Gemini, and OpenRouter
<i>Support matrix</i>	Pre-vetted local model variants and supported NVIDIA NIM models

6.1 Local models

AIDB supports two kinds of local models — both run directly on your Postgres host without requiring an external API:

- **Default models** — AIDB bundles a default variant of each supported model type and registers them automatically. No configuration is needed to start using them, though the model’s files still download from HuggingFace the first time it’s used — see [Local model downloads](#).
- **Custom models** — If you need a different model variant, a fine-tuned model, or a specific local path, you can register a custom version using `aidb.create_model()`.

Local models run on one of two inference runtimes, Candle or llama.cpp. Day-to-day usage — `aidb.create_model()`, `aidb.encode_text()`, `aidb.generate_text()`, and so on — is identical either way; each model’s provider below just notes which runtime it uses.

6.1.1 Default models

The following models are available in every AIDB installation without any configuration. See [Model providers](#) for what the provider column means:

Model name	Provider	Use
<i>bert</i>	<i>bert_local</i> (Candle)	Text embeddings
<i>clip</i>	<i>clip_local</i> (Candle)	Multimodal embeddings (text and image)
<i>t5</i>	<i>t5_local</i> (Candle)	Text-to-text generation
<i>llama</i>	<i>llama_instruct_local</i> (Candle)	Text-to-text generation
<i>bge-small-en-v1.5-f16</i>	<i>llamacpp_embeddings</i> (llama.cpp)	Text embeddings
<i>nomie-embed-text-v1.5-Q</i>	<i>llamacpp_embeddings</i> (llama.cpp)	Text embeddings
<i>bge-m3-f16</i>	<i>llamacpp_embeddings</i> (llama.cpp)	Text embeddings
<i>qwen3-embedding-0.6b-ζ</i>	<i>llamacpp_embeddings</i> (llama.cpp)	Text embeddings
<i>qwen3-embedding-4b-Q8</i>	<i>llamacpp_embeddings</i> (llama.cpp)	Text embeddings
<i>qwen3.5-0.8b-Q8_0</i>	<i>llamacpp_generate</i> (llama.cpp)	Text-to-text generation
<i>llama-3.2-1b-instruct-Q8</i>	<i>llamacpp_generate</i> (llama.cpp)	Text-to-text generation
<i>lightonocr-2-1b-Q8_0</i>	<i>llamacpp_ocr</i> (llama.cpp)	OCR (text extraction from images)
<i>dummy</i>	<i>dummy</i>	Testing and development (returns zeros)

Use them directly by name in any AIDB SQL function or pipeline step configuration:

```

-- Text embedding with the default BERT model
SELECT aidb.encode_text(bert, The quick brown fox);

-- Image embedding with the default CLIP model
SELECT aidb.encode_image(clip, pg_read_binary_file(/tmp/photo.jpg)::BYTEA);

-- Text generation with the default T5 model
SELECT aidb.generate_text(t5, Translate to French: Hello, world.);

-- Text embedding with a default llama.cpp embedding model
SELECT aidb.encode_text(bge-small-en-v1.5-f16, The quick brown fox);

-- Text generation with a default llama.cpp model
SELECT aidb.generate_text(qwen3.5-0.8b-Q8_0, Translate to French: Hello, world.);

```

6.1.2 Model providers

A **provider** tells `aidb.create_model()` which inference engine to use for a model. It isn't a SQL enum — it's the name of a Postgres foreign server registered for AIDB's model registry, matched against an adapter at model-creation time. List the providers registered in your installation, with their descriptions:

```
SELECT server_name, server_description FROM aidb.model_providers ORDER BY server_name;
```

AIDB's local providers, the runtime behind each, and what they're for:

Provider	Runtime	Description
<i>bert_local</i>	Candle	Simple language model, ideal for encoding text.
<i>clip_local</i>	Candle	Vision/text model, ideal for encoding text and images.
<i>t5_local</i>	Candle	Ideal for translation, summarization, and question answering.
<i>llama_instruct_local</i>	Candle	Instruction variants of Meta's family of large language models.
<i>llamacpp_generate</i>	llama.cpp	Local LLM inference via llama.cpp (GGUF format).
<i>llamacpp_embeddings</i>	llama.cpp	Local text embeddings via llama.cpp (GGUF format).
<i>llamacpp_reranking</i>	llama.cpp	Local reranking via llama.cpp (GGUF format).
<i>llamacpp_ocr</i>	llama.cpp	Local OCR (text extraction from images) via llama.cpp (GGUF format).
<i>dummy</i>	—	Provides fake data, ideal for testing.

`aidb.model_providers` also lists external API providers (OpenAI, NVIDIA NIM, Gemini, OpenRouter, and so on) — see [External model connections](#) for those. See [aidb.model_providers](#) for the full column reference.

Pick the provider that matches your model file format and task, then configure it with the matching config helper under [Custom models](#) — or use one of the ready-to-use [Default models](#) above.

6.1.3 Custom models

Use `aidb.create_model()` to register a custom variant when you need a different model version, a fine-tuned model, or a specific local path:

```
SELECT aidb.create_model(
  name      => my_model,
  provider  => <provider>,
  config    => <config_helper>(...)
);
```

Parameter	Type	Description
<i>name</i>	TEXT	A unique name for this model. Used to reference it in pipelines and SQL functions.
<i>provider</i>	TEXT	The model provider. Determines which inference engine AIDB uses.
<i>config</i>	JSONB	Provider-specific configuration, built with a config helper function.

BERT, CLIP, T5, and Llama below run on Candle, each with its own config helper. llama.cpp (its own subsection, last) runs a separate runtime and doesn't have a config helper yet.

BERT

Use `aidb.bert_config()` to specify the HuggingFace model identifier, an optional revision, and an optional local cache directory:

```
SELECT aidb.create_model(
  my_bert,
  bert_local,
  config => aidb.bert_config(sentence-transformers/all-MiniLM-L6-v2)
);
```

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	Model name or path (HuggingFace identifier).
<i>revision</i>	TEXT	NULL	Model revision or branch.
<i>cache_dir</i>	TEXT	NULL	Local directory for caching model files.

CLIP

Use `aidb.clip_config()` to configure a CLIP-based multimodal model for joint text and image embeddings:

```
SELECT aidb.create_model(
    my_clip,
    clip_local,
    config => aidb.clip_config(openai/clip-vit-base-patch32)
);
```

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	Model name or path (HuggingFace identifier).
<i>revision</i>	TEXT	NULL	Model revision or branch.
<i>cache_dir</i>	TEXT	NULL	Local directory for caching model files.
<i>image_size</i>	INTEGER	NULL	Input image size in pixels.

T5

Use `aidb.t5_config()` to configure a T5 text-to-text model. T5 models support tasks like translation, summarization, and question answering depending on the variant:

```
SELECT aidb.create_model(
    my_t5,
    t5_local,
    config => aidb.t5_config(google/flan-t5-base)
);
```

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	Model name or path.
<i>revision</i>	TEXT	NULL	Model revision or branch.
<i>temperature</i>	DOUBLE PRECISION	NULL	Sampling temperature.
<i>seed</i>	BIGINT	NULL	Random seed for reproducible outputs.
<i>max_tokens</i>	INTEGER	NULL	Maximum tokens to generate.
<i>repeat_penalty</i>	REAL	NULL	Penalty applied to repeated tokens.
<i>repeat_last_n</i>	INTEGER	NULL	Number of trailing tokens considered for the repeat penalty.

continues on next page

Table 7 – continued from previous page

Parameter	Type	Default	Description
<i>model_path</i>	TEXT	NULL	Explicit local path to model weights (overrides <i>model</i>).
<i>cache_dir</i>	TEXT	NULL	Local directory for caching model files.
<i>top_p</i>	DOUBLE PRECISION	NULL	Nucleus sampling threshold.

Llama

Use `aidb.llama_config()` to configure a Llama-family language model for text generation. Llama models support temperature, top-p sampling, context length, and other generation parameters:

```
SELECT aidb.create_model(
  my_llama,
  llama_instruct_local,
  config => aidb.llama_config(
    meta-llama/Llama-3.2-3B-Instruct,
    temperature => 0.5
  )
);
```

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	Model name or path.
<i>revision</i>	TEXT	NULL	Model revision or branch.
<i>cache_dir</i>	TEXT	NULL	Local directory for caching model files.
<i>system_prompt</i>	TEXT	NULL	Default system prompt.
<i>use_flash_attention</i>	BOOLEAN	NULL	Enable flash attention.
<i>model_path</i>	TEXT	NULL	Explicit local path to model weights (overrides <i>model</i>).
<i>seed</i>	BIGINT	NULL	Random seed for reproducible outputs.
<i>temperature</i>	DOUBLE PRECISION	NULL	Sampling temperature.
<i>top_p</i>	DOUBLE PRECISION	NULL	Nucleus sampling threshold.
<i>sample_len</i>	INTEGER	NULL	Maximum tokens to generate.
<i>use_kv_cache</i>	BOOLEAN	NULL	Enable KV cache.
<i>repeat_penalty</i>	REAL	NULL	Penalty applied to repeated tokens.
<i>repeat_last_n</i>	INTEGER	NULL	Number of trailing tokens considered for the repeat penalty.

!!!note Local model files are downloaded from HuggingFace and cached on first use. The Postgres process must have network access and write access to the cache directory. For air-gapped environments, pre-download the model files and use `model_path` or `cache_dir` to point to the local copy.

llama.cpp

llama.cpp runs GGUF-format model files — a common distribution format for quantized, open-weight models — through four providers: `llamacpp_generate` for text generation, `llamacpp_embeddings` for text embeddings, `llamacpp_reranking` for reranking, and `llamacpp_ocr` for OCR (text extraction from images). Point any of them at a GGUF file, downloaded from HuggingFace or already on disk. See [Default models](#) above for AIDB’s pre-registered GGUF models; use the recipes below to register a different one.

llamacpp_generate

```
SELECT aidb.create_model(
  name => my_llamacpp_chat,
  provider => llamacpp_generate,
  config => jsonb_build_object(
    local_path, /var/lib/aidb/models/qwen2.5-3b-instruct-q4_k_m.gguf,
    n_ctx, 4096,
    temperature, 0.2
  )
);
```

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required unless <i>local_path</i> is set	HuggingFace repo id, for example, <i>Qwen/Qwen2-0.5B-Instruct-GGUF</i> .
<i>model_file</i>	TEXT	Required unless <i>local_path</i> is set	Filename of the <i>.gguf</i> inside the repo. For a split GGUF, name any one shard — every shard downloads and llama.cpp reassembles them.
<i>revision</i>	TEXT	<i>main</i>	HuggingFace revision.
<i>local_path</i>	TEXT	NULL	Absolute path to a local <i>.gguf</i> file. Overrides <i>model/model_file/revision</i> .
<i>cache_dir</i>	TEXT	NULL	Local directory for caching downloaded model files.
<i>n_ctx</i>	INTEGER	4096	Context window, in tokens. Capped at the value the GGUF was trained with.
<i>chat_template</i>	TEXT	NULL	Override the chat template baked into the GGUF.
<i>temperature</i>	DOUBLE PRECISION	0.7	Sampling temperature.
<i>top_p</i>	DOUBLE PRECISION	0.9	Nucleus sampling threshold.
<i>max_tokens</i>	INTEGER	512	Maximum tokens to generate.
<i>seed</i>	BIGINT	NULL	Random seed for reproducible outputs.
<i>system_prompt</i>	TEXT	NULL	System prompt prepended to every request.
<i>repeat_penalty</i>	REAL	1.1	Penalty applied to repeated tokens.
<i>repeat_last_n</i>	INTEGER	64	Number of trailing tokens considered for the repeat penalty.

continues on next page

Table 9 – continued from previous page

Parameter	Type	Default	Description
<i>thinking</i>	BOOLEAN	NULL (keeps tags)	Controls <i><think></i> tag handling in the response: <i>false</i> strips them, <i>true</i> /unset keeps them.

`llamacpp_generate` also supports `tools`, `tool_choice`, and `response_format` for tool calling and structured output, as do the native `openai_responses` and `anthropic_messages` providers (see [External model connections](#)) — see [Tool calling and structured output](#).

llamacpp_embeddings

```
SELECT aidb.create_model(
  name => my_llamacpp_embed,
  provider => llamacpp_embeddings,
  config => jsonb_build_object(
    model, nomic-ai/nomic-embed-text-v1.5-GGUF,
    model_file, nomic-embed-text-v1.5.Q4_K_M.gguf,
    revision, 0188c9bf409793f810680a5a431e7b899c46104c,
    n_ctx, 2048,
    normalize, true
  )
);
```

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required unless <i>local_path</i> is set	HuggingFace repo id, for example, <i>nomic-ai/nomic-embed-text-v1.5-GGUF</i> .
<i>model_file</i>	TEXT	Required unless <i>local_path</i> is set	Filename of the <i>.gguf</i> inside the repo. For a split GGUF, name any one shard — every shard downloads and llama.cpp reassembles them.
<i>revision</i>	TEXT	<i>main</i>	HuggingFace revision.
<i>local_path</i>	TEXT	NULL	Absolute path to a local <i>.gguf</i> file. Overrides <i>model/model_file/revision</i> .
<i>cache_dir</i>	TEXT	NULL	Local directory for caching downloaded model files.
<i>n_ctx</i>	INTEGER	2048	Context window, in tokens. Capped at the value the GGUF was trained with.
<i>pooling</i>	TEXT	<i>unspecified</i>	Pooling strategy: <i>unspecified</i> (use the GGUF's own metadata), <i>none</i> , <i>mean</i> , <i>cls</i> , or <i>last</i> .
<i>normalize</i>	BOOLEAN	<i>true</i>	L2-normalize output embedding vectors.

llamacpp_reranking

Serves two different reranker architectures behind one provider. The scoring mechanism auto-detects from the GGUF's metadata, or you can set it explicitly:

```
- - Cross-encoder (auto-detected from the GGUFs architecture)
SELECT aidb.create_model(
  name => my_llamacpp_reranker,
  provider => llamacpp_reranking,
  config => jsonb_build_object(
    model, gpustack/bge-reranker-v2-m3-GGUF,
    model_file, bge-reranker-v2-m3-Q4_K_M.gguf,
    revision, 3093af03b1a635e67b084b1d8c03c5f5e020fd05,
    n_ctx, 2048
  )
);

- - Decoder-only (Qwen3-Reranker family), with a custom task instruction
SELECT aidb.create_model(
  name => my_llamacpp_reranker_decoder,
  provider => llamacpp_reranking,
  config => jsonb_build_object(
    model, ggml-org/Qwen3-Reranker-0.6B-Q8_0-GGUF,
    model_file, qwen3-reranker-0.6b-q8_0.gguf,
    revision, a02f48bb4f057028298c21fa033da2b30d7742d5,
    n_ctx, 2048,
    instruction, Retrieve legal passages relevant to the query
  )
);
```

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required unless <i>local_path</i> is set	HuggingFace repo id, for example, <i>gpustack/bge-reranker-v2-m3-GGUF</i> .
<i>model_file</i>	TEXT	Required unless <i>local_path</i> is set	Filename of the <i>.gguf</i> inside the repo. For a split GGUF, name any one shard — every shard downloads and llama.cpp reassembles them.
<i>revision</i>	TEXT	<i>main</i>	HuggingFace revision.
<i>local_path</i>	TEXT	NULL	Absolute path to a local <i>.gguf</i> file. Overrides <i>model/model_file/revision</i> .
<i>cache_dir</i>	TEXT	NULL	Local directory for caching downloaded model files.
<i>n_ctx</i>	INTEGER	2048	Context window, in tokens. The query/document pair is rejected at call time if it doesn't fit, rather than silently truncated.
<i>reranker_type</i>	TEXT	NULL (auto-detected)	Force the scoring mechanism: <i>cross_encoder</i> or <i>decoder_only</i> . Only needed to correct a misdetection.

continues on next page

Table 11 – continued from previous page

Parameter	Type	Default	Description
<i>instruction</i>	TEXT	Built-in generic retrieval instruction	Task instruction for decoder-only rerankers (Qwen3-Reranker family). Ignored by cross-encoders.

`llamacpp_reranking` auto-detects the scoring mechanism from the GGUF's `general.architecture` metadata: BERT/XLM-RoBERTa-family architectures (`bge-reranker-v2-m3`, `jina-reranker`, and similar) score as a cross-encoder; the `qwen3` architecture scores as decoder-only, using the Qwen3-Reranker family's yes/no verdict-token prompt. Any other architecture is rejected at model-load time — set `reranker_type` explicitly only if you're sure a model matches one of these two scoring paths despite being misdetected.

!!!note Unlike `nim_reranking` and other remote reranking providers, `llamacpp_reranking`'s `logit_score` is a raw, unbounded model logit rather than a normalized 0–1 relevance score. Compare scores within one `aidb.rerank_text()` call, not across providers or calls.

!!!note `llama.cpp` doesn't have a config helper function yet — build `config` with `jsonb_build_object()` as shown above. It's also CPU-only today; there's no GPU offload option. See [aidb.max_threads](#) to control how many threads it (and the Candle runtime) can use.

llamacpp_ocr

Extracts text from images using a local vision-language GGUF model plus its multimodal projector (mmproj) file:

```
SELECT aidb.create_model(
  name => my_llamacpp_ocr,
  provider => llamacpp_ocr,
  config => jsonb_build_object(
    model, ggml-org/LightOnOCR-2-1B-GGUF,
    model_file, LightOnOCR-2-1B-Q8_0.gguf,
    mmproj_file, mmproj-LightOnOCR-2-1B-Q8_0.gguf,
    revision, 7d4daf70c2856a155b1413def00f4ae59843a9cf,
    n_ctx, 8192
  )
);
```

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required unless <i>local_path</i> is set	HuggingFace repo id, for example, <code>ggml-org/LightOnOCR-2-1B-GGUF</code> .
<i>model_file</i>	TEXT	Required unless <i>local_path</i> is set	Filename of the base model <code>.gguf</code> inside the repo.
<i>mmproj_file</i>	TEXT	Required unless <i>mmproj_local_path</i> is set	Filename of the multimodal projector (mmproj) <code>.gguf</code> , in the same HuggingFace repo/revision as the base model.
<i>mmproj_local_path</i>	TEXT	NULL	Absolute path to a local mmproj <code>.gguf</code> file. Overrides the HuggingFace-based <i>mmproj_file</i> download.
<i>revision</i>	TEXT	<i>main</i>	HuggingFace revision.

continues on next page

Table 12 – continued from previous page

Parameter	Type	Default	Description
<i>local_path</i>	TEXT	NULL	Absolute path to a local base model <i>.gguf</i> file. Overrides <i>model/model_file/revision</i> .
<i>cache_dir</i>	TEXT	NULL	Local directory for caching downloaded model files.
<i>n_ctx</i>	INTEGER	8192	Context window, in tokens.
<i>prompt</i>	TEXT	Built-in transcription instruction	OCR instruction sent with each image. Prompt influence is model-dependent.
<i>temperature</i>	DOUBLE PRECISION	0.0	Sampling temperature. Defaults to <i>0.0</i> for faithful transcription rather than creative sampling.

6.1.4 Validating a model at creation

By default, `aidb.create_model()` validates the model as part of registration: it downloads and loads the model files and runs a small probe inference to confirm the model is functional before the registration commits. For local models this moves the (potentially large) download and load to creation time instead of first use.

If validation fails — for example, the model identifier doesn’t exist on HuggingFace, a `model_path` is wrong, or the weights can’t be loaded — the error is reported immediately and no model is registered.

```
-- Validated by default: downloads, loads, and probes the model now.
SELECT aidb.create_model(my_bert, bert_local,
    config => aidb.bert_config(sentence-transformers/all-MiniLM-L6-v2));
```

Pass `validate => false` to register the model without downloading or loading it — useful to defer the download to first use, or to register models in an environment that doesn’t have network access yet:

```
SELECT aidb.create_model(my_bert, bert_local,
    config      => aidb.bert_config(sentence-transformers/all-MiniLM-L6-v2),
    validate    => false);
```

You can validate a model later — or recheck an existing one — with `aidb.validate_model()`.

6.1.5 Local model downloads

When a local model is used for the first time (or any time its cache is missing or corrupt), AIDB downloads the model files from HuggingFace into the configured `cache_dir`. The downloader emits progress messages and automatically retries transient network failures.

Progress messages

For each file, AIDB emits a `downloading` line when the request starts and a `complete` line when it finishes. For multi-gigabyte files, a heartbeat line reports cumulative bytes and percent complete approximately every 30 seconds, so the session doesn’t appear frozen during large downloads.

```
SELECT aidb.generate_text(t5, test);
__OUTPUT__
NOTICE: [t5_local] loading t5-small (rev refs/pr/15) -> /var/.../snapshots/refs/pr/15
NOTICE: [hf] downloading https://huggingface.co/t5-small/resolve/refs%2Fpr%2F15/
```

(continues on next page)

(continued from previous page)

```

↪config.json (expected: 1.2 KB)
NOTICE: [hf] complete: https://huggingface.co/t5-small/resolve/refs%2Fpr%2F15/config.
↪json (1.2 KB) in 0.9s
NOTICE: [hf] downloading https://huggingface.co/t5-small/resolve/refs%2Fpr%2F15/
↪model.safetensors (size unknown)
NOTICE: [hf] downloading t5-small/model.safetensors: 115.4 MB / 230.8 MB (50.0%)
NOTICE: [hf] downloading t5-small/model.safetensors: 216.4 MB / 230.8 MB (93.7%)
NOTICE: [hf] complete: https://huggingface.co/t5-small/resolve/refs%2Fpr%2F15/model.
↪safetensors (230.8 MB) in 83.5s

```

By default, these lines are sent to the client at `NOTICE` level. Use the `aidb.download_log_level` parameter to route them to the server log instead, or to suppress them entirely.

Retries and resume

A CDN that drops the connection partway through a multi-gigabyte download won't fail the request outright:

- Each retry sends a `Range` header from wherever the previous attempt stopped, so progress accumulates across attempts instead of restarting from zero.
- AIDB distinguishes “still receiving new bytes” from “zero progress” and only counts stalled attempts against `aidb.download_max_attempts`, with exponential backoff between attempts.
- A `Ctrl-C` from `psql` lands between retries instead of waiting for the current request to time out.

The total number of attempts per file is capped by `aidb.download_max_attempts` (default 30).

Verification and self-heal

AIDB verifies downloaded files using the size and (when available) the SHA-256 hash advertised by HuggingFace, and writes a small sidecar file next to each completed download recording those values. On every cache hit AIDB rechecks the file size against the sidecar, so a file truncated after the original download is caught before it's used.

If a cached file is detected as corrupt at model-load time, AIDB purges the affected snapshot and redownloads it once automatically.

6.1.6 Using a registered model

Once registered, a model is referenced by its name in any AIDB function. Custom and default models are interchangeable:

```

-- Use a custom BERT model for embedding
SELECT aidb.encode_text(my_bert, Hello world);

-- Use a custom BERT model in a pipeline step
SELECT aidb.create_pipeline(
  name => my_pipeline,
  source => my_source_table,
  source_key_column => id,
  source_data_column => content,
  step_1 => KnowledgeBase,
  step_1_options => aidb.knowledge_base_config(model => my_bert, data_format =>
↪Text)
);

```

See the [Models reference](#) for full parameter details on all model config helper functions. Some functions (`generate_text()`, `summarize_text()`) also accept a per-call configuration override — see [Configuration model](#) for creation-time vs. call-time settings.

For a list of HuggingFace model variants that have been tested and verified to work with AIDB’s local providers, see [Support matrix](#).

6.1.7 Troubleshooting

llama.cpp model loading and inference issues

If a `llamacpp_generate`, `llamacpp_embeddings`, or `llamacpp_reranking` model fails to load, times out, or otherwise behaves unexpectedly, AIDB’s own error messages may not have enough detail to diagnose it. Enable llama.cpp’s native diagnostic logging to see its own output — GGUF metadata, tensor shapes, and backend/model initialization — in the server log:

```
aidb.enable_llamacpp_logs = true
```

This parameter is read once at extension initialization, so restart Postgres for the change to take effect — `SET` and `pg_reload_conf()` alone won’t apply it. See [aidb.enable_llamacpp_logs](#) for the full parameter reference.

It’s off by default because llama.cpp’s native output is verbose (hundreds of lines per model load), so turn it on only while investigating an issue, then turn it back off. It applies only to llama.cpp models; Candle-based models (`bert_local`, `clip_local`, `t5_local`, `llama_instruct_local`) are unaffected.

6.2 External model connections

External models are API-based models hosted outside your Postgres instance. AIDB connects to them by registering a model with `aidb.create_model()`, supplying the provider name, a configuration object, and credentials such as an API key, if required.

!!!note The examples on this page pass credentials inline as `credentials => '{"api_key": "..."}':JSONB` for brevity. `config` itself can never contain an `api_key` or `basic_auth` key at any nesting depth — `aidb.create_model()` rejects it — so always pass credentials through the separate `credentials` argument. To avoid storing the secret in the database at all, pass `credentials_env` instead, naming an environment variable to read it from at connection time. See [Credentials from environment variables](#).

Once registered, an external model is used exactly like a default local model — by name in any AIDB SQL function or pipeline step.

6.2.1 Validating a model at creation

By default, `aidb.create_model()` validates an external model as part of registration by sending a small probe request to the provider. This confirms the endpoint is reachable and the configuration and credentials are correct before the registration commits, surfacing problems like an invalid API key or wrong URL immediately instead of on first use. If the probe fails, the error is reported and no model is registered.

```
- - Validated by default: sends a test request to the provider now.
SELECT aidb.create_model(
  my_openai_embedder,
  openai_embeddings,
  config      => aidb.embeddings_config(model => text-embedding-3-small),
  credentials => '{"api_key": "sk-..."}':JSONB
);
```

Pass `validate => false` to skip the probe — for example, when the endpoint or credentials aren't available yet, or to avoid the request during scripted setup:

```
SELECT aidb.create_model(
  my_openai_embedder,
  openai_embeddings,
  config      => aidb.embeddings_config(model => text-embedding-3-small),
  credentials => {"api_key": "sk-..."}::JSONB,
  validate    => false
);
```

!!!note Validation sends a real request to the provider, so it requires network access from the Postgres host at creation time and may incur a small usage cost.

You can validate a model later — or re-check an existing one — with `aidb.validate_model()`.

6.2.2 OpenAI-compatible endpoints

AIDB supports any embedding or chat/completions service that implements the OpenAI API. This includes OpenAI itself, self-hosted servers like [Ollama](#), and any other compliant endpoint.

Embeddings

Use the `openai_embeddings` provider with `aidb.embeddings_config()`. Override the `url` parameter to redirect requests to any OpenAI-compatible server:

```
-- OpenAI
SELECT aidb.create_model(
  my_openai_embedder,
  openai_embeddings,
  config      => aidb.embeddings_config(
    model => text-embedding-3-small,
    url   => https://api.openai.com/v1
  ),
  credentials => {"api_key": "sk-..."}::JSONB
);

-- Local Ollama server
SELECT aidb.create_model(
  my_ollama_embedder,
  openai_embeddings,
  config => aidb.embeddings_config(
    model => llama3.2,
    url   => http://llama.local:11434/v1/embeddings
  )
);
```

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	Model identifier as expected by the API.
<i>api_key</i>	TEXT	NULL	API key for authentication.
<i>url</i>	TEXT	NULL	API endpoint URL. Defaults to OpenAI's endpoint.

continues on next page

Table 13 – continued from previous page

Parameter	Type	Default	Description
<i>basic_auth</i>	TEXT	NULL	Basic auth credentials (<i>user:password</i>).
<i>max_concurrent_requests</i>	INTEGER	NULL	Maximum concurrent requests to the endpoint.
<i>max_batch_size</i>	INTEGER	NULL	Maximum number of inputs per batch request.
<i>input_type</i>	TEXT	NULL	Input type hint for encoding (provider-specific).
<i>input_type_query</i>	TEXT	NULL	Input type hint for query encoding (provider-specific).
<i>is_hcp_model</i>	BOOLEAN	NULL	Set to <i>true</i> if referencing a model running on HCP.

Chat completions

Use the `openai_completions` provider with `aidb.completions_config()`. The `url` parameter works the same way — omit it for OpenAI, or override it for any other compliant server:

```
SELECT aidb.create_model(
  my_openai_llm,
  openai_completions,
  config => aidb.completions_config(
    model => gpt-4o,
    temperature => 0.2
  ),
  credentials => {"api_key": "sk-..."::JSONB
);
```

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	Model identifier.
<i>api_key</i>	TEXT	NULL	API key for authentication.
<i>url</i>	TEXT	NULL	API endpoint URL. Defaults to OpenAI's endpoint.
<i>basic_auth</i>	TEXT	NULL	Basic auth credentials (<i>user:password</i>).
<i>temperature</i>	DOUBLE PRECISION	NULL	Sampling temperature.
<i>top_p</i>	DOUBLE PRECISION	NULL	Nucleus sampling threshold.
<i>seed</i>	BIGINT	NULL	Random seed for reproducible outputs.
<i>system_prompt</i>	TEXT	NULL	Default system prompt prepended to every request.
<i>max_tokens</i>	JSONB	NULL	Max tokens config (from <code>aidb.max_tokens_config()</code>).
<i>thinking</i>	BOOLEAN	NULL	Enable extended reasoning (supported models only).
<i>max_concurrent_requests</i>	INTEGER	NULL	Maximum concurrent requests to the endpoint.

continues on next page

Table 14 – continued from previous page

Parameter	Type	Default	Description
<i>extra_args</i>	JSONB	NULL	Additional provider-specific arguments.
<i>is_hcp_model</i>	BOOLEAN	NULL	Set to <i>true</i> if referencing a model running on HCP.

!!!note When using pgvector indexing, be aware that [pgvector limits indexed vectors to 2000 dimensions](#). If the model you choose produces more dimensions, use `aidb.vector_index_disabled_config()` in your pipeline step and manage the index manually.

6.2.3 OpenAI Responses API

AIDB has two providers for OpenAI models — both work with the same model identifiers (`gpt-4o`, `gpt-5.x`, and so on), so pick between them based on what you’re doing, not which models you need:

Provider	API	Best for	Tool calling
<i>openai_completions</i>	Chat Completions (<i>/v1/chat/completions</i>)	Simple, one-off text generation — <i>aidb.generate_text()</i> , summarization, pipeline steps.	Rejects <i>tools/tool_choice/response_format</i> in <i>aidb.generate_text()</i> 's <i>inference_config</i> .
<i>openai_responses</i>	Responses API (<i>/v1/responses</i>)	Agentic workloads — multi-step reasoning, reliable tool calling, structured output.	Native — emits the API's real <i>tools/tool_choice</i> request fields and parses real tool-call responses back.

Both providers can also back an [Managing agents](#) agent — AIDB always gives the agent tool-calling ability, but *how* differs by provider. Backed by `openai_responses`, tool calls go over the wire as genuine Responses API tool calls. Backed by `openai_completions` (or almost any other provider), AIDB describes the available tools in the prompt text and parses a tool call back out of the model’s plain-text reply — a simulation, not the model’s own tool-calling behavior. `openai_responses` is the more reliable choice for agents; use `openai_completions` when you just need text out, not tool calling.

```
SELECT aidb.create_model(
  my_gpt,
  openai_responses,
  config      => aidb.openai_responses_config(model => gpt-5.1),
  credentials => {"api_key": "sk-..."::JSONB
);
```

Azure AI Foundry

Use `openai_responses_azure` for models hosted on Azure AI Foundry’s unified Responses API surface. It’s the same wire shape as direct OpenAI, but `url` is required since there’s no universal default — point it at your resource’s endpoint:

```
SELECT aidb.create_model(
  my_gpt_azure,
```

(continues on next page)

(continued from previous page)

```

openai_responses_azure,
config      => aidb.openai_responses_config(
    model => gpt-5.1,
    url   => https://<resource>.openai.azure.com/openai/v1/responses
),
credentials => [{"api_key": "..."}]:JSONB
);

```

See `aidb.openai_responses_config` for the full parameter list, and *Tool calling and structured output* for using `tools`, `tool_choice`, and `response_format` with either provider.

6.2.4 Anthropic Messages API

The `anthropic_messages` provider connects to Anthropic’s native Messages API (`/v1/messages`), with the same native tool-calling support as `openai_responses`: `real tools/tool_choice` request fields, real tool-call responses, and structured output via `response_format`.

```

SELECT aidb.create_model(
    my_claude,
    anthropic_messages,
    config      => aidb.anthropic_messages_config(model => claude-opus-4-6),
    credentials => [{"api_key": "sk-ant-..."}]:JSONB
);

```

Azure AI Foundry and AWS Bedrock

Two more variants share the same config helper and wire body, differing only in auth and endpoint:

- `anthropic_messages_azure` — Azure AI Foundry’s native Anthropic Messages API surface. `url` is required (your resource’s Messages endpoint).
- `anthropic_messages_bedrock` — AWS Bedrock’s InvokeModel API, authenticated with a Bedrock API key (bearer token, not AWS SigV4). `url` is required and should be the regional `bedrock-runtime` base endpoint — AIDB appends the model ID to the path automatically.

```

-- Azure AI Foundry
SELECT aidb.create_model(
    my_claude_azure,
    anthropic_messages_azure,
    config      => aidb.anthropic_messages_config(
        model => claude-opus-4-6,
        url   => https://<resource>.services.ai.azure.com/anthropic/v1/messages
    ),
    credentials => [{"api_key": "..."}]:JSONB
);

-- AWS Bedrock
SELECT aidb.create_model(
    my_claude_bedrock,
    anthropic_messages_bedrock,
    config      => aidb.anthropic_messages_config(
        model => anthropic.claude-opus-4-6-v1:0,
        url   => https://bedrock-runtime.us-east-1.amazonaws.com

```

(continues on next page)

(continued from previous page)

```

),
credentials => {"api_key": "..."}::JSONB
);

```

!!!note Anthropic's Messages API has no native structured-output field. When you use `response_format`, AIDB translates it into a forced call to a synthetic tool internally — functionally equivalent, but it means `response_format` can't be combined with `tools/tool_choice` in the same call.

`anthropic_messages` is AIDB's only provider built specifically for Claude — there's no older `anthropic_completions` equivalent to `openai_completions`. If you'd rather reach Claude (or OpenAI, or other vendors) through a single unified gateway instead of a per-vendor provider, see [OpenRouter](#) below; the tradeoff is the same as `openai_completions` above — OpenRouter models get simulated (prompt-injected), not native, tool calling when backing an agent.

See [aidb.anthropic_messages_config](#) for the full parameter list.

6.2.5 NVIDIA NIM

AIDB supports NVIDIA NIM microservices hosted on [build.nvidia.com](#) as well as NIM instances running in your own environment.

Text generation (NIM completions)

Use the `nim_completions` provider with `aidb.completions_config()` — the same config helper `openai_completions` uses, since NIM's completions endpoint is OpenAI-compatible. See [Chat completions](#) above for the full parameter list.

```

SELECT aidb.create_model(
  my_nim_llm,
  nim_completions,
  config      => aidb.completions_config(
    model => meta/llama-3.3-70b-instruct
  ),
  credentials => {"api_key": "nvapi-..."}::JSONB
);

```

To use a NIM instance running in your own environment, pass `url` to `aidb.completions_config()` pointing at your NIM endpoint.

After registering, run the model with `aidb.generate_text()`:

```

SELECT aidb.generate_text(my_nim_llm, Tell me a short, one sentence story);

__OUTPUT__

                                generate_text
-----
As the clock struck midnight, a single tear fell from the porcelain dolls eye.
(1 row)

```

Multimodal embeddings (NIM CLIP)

Use the `nim_clip` provider with `aidb.nim_clip_config()` for joint text and image embeddings:

```

SELECT aidb.create_model(
  my_nim_clip,
  nim_clip,
  config      => aidb.nim_clip_config(
    model => nvidia/nvclip
  ),
  credentials => [{"api_key": "nvapi-..."}>::JSONB
);

```

`aidb.nim_clip_config()` parameters:

Parameter	Type	Default	Description
<i>api_key</i>	TEXT	NULL	API key for NIM authentication.
<i>model</i>	TEXT	NULL	NIM CLIP model identifier.
<i>url</i>	TEXT	NULL	NIM endpoint URL override.
<i>basic_auth</i>	TEXT	NULL	Basic auth credentials.
<i>is_hcp_model</i>	BOOLEAN	NULL	Set to <i>true</i> if the model is running on HCP.

OCR (NIM OCR)

Use the `nim_paddle_ocr` provider with `aidb.nim_ocr_config()` to extract text from images:

```

SELECT aidb.create_model(
  my_nim_ocr,
  nim_paddle_ocr,
  config      => aidb.nim_ocr_config(),
  credentials => [{"api_key": "nvapi-..."}>::JSONB
);

```

`aidb.nim_ocr_config()` parameters:

Parameter	Type	Default	Description
<i>api_key</i>	TEXT	NULL	API key for NIM authentication.
<i>model</i>	TEXT	NULL	NIM OCR model identifier.
<i>url</i>	TEXT	NULL	NIM endpoint URL override.
<i>basic_auth</i>	TEXT	NULL	Basic auth credentials.
<i>is_hcp_model</i>	BOOLEAN	NULL	Set to <i>true</i> if the model is running on HCP.

Reranking (NIM reranking)

Use the `nim_reranking` provider with `aidb.nim_reranking_config()` to re-score search results by relevance. See [Text reranking](#) for usage with `aidb.rerank_text()` :

```

SELECT aidb.create_model(
  my_reranker,
  nim_reranking,
  config      => aidb.nim_reranking_config(
    model => nvidia/nv-rerankqa-mistral-4b-v3
  ),

```

(continues on next page)

(continued from previous page)

```
credentials => {"api_key": "nvapi-..."}::JSONB
);
```

`aidb.nim_reranking_config()` parameters:

Parameter	Type	Default	Description
<i>api_key</i>	TEXT	NULL	API key for NIM authentication.
<i>model</i>	TEXT	NULL	NIM reranking model identifier.
<i>url</i>	TEXT	NULL	NIM endpoint URL override.
<i>basic_auth</i>	TEXT	NULL	Basic auth credentials.
<i>is_hcp_model</i>	BOOLEAN	NULL	Set to <i>true</i> if the model is running on HCP.

To get a NIM API key, create an account at build.nvidia.com, select a model, and generate a key from the model's page.

6.2.6 Google Gemini

Use the `gemini` provider with `aidb.gemini_config()`. `api_key` is `aidb.gemini_config()`'s first parameter and has no default, but pass it as `NULL` and supply the real key through `create_model()`'s `credentials` argument instead — `config` can never contain an `api_key` field:

```
SELECT aidb.create_model(
  my_gemini,
  gemini,
  config      => aidb.gemini_config(
    api_key => NULL,
    model  => gemini-2.0-flash
  ),
  credentials => {"api_key": "AIza..."}::JSONB
);
```

`aidb.gemini_config()` parameters:

Parameter	Type	Default	Description
<i>api_key</i>	TEXT	Required	Has no default, but pass <i>NULL</i> and supply the real key via <i>create_model()</i> 's <i>credentials</i> argument instead — <i>config</i> can never contain <i>api_key</i> .
<i>model</i>	TEXT	NULL	Gemini model identifier (for example, <i>gemini-2.0-flash</i>).
<i>url</i>	TEXT	NULL	API endpoint URL override.
<i>max_concurrent_requests</i>	INTEGER	NULL	Maximum concurrent requests to the API.
<i>thinking_budget</i>	INTEGER	NULL	Extended thinking token budget (Gemini 2.x models only).

6.2.7 OpenRouter

OpenRouter provides a unified API gateway to 200+ models from multiple providers. AIDB supports both chat completions and embeddings via OpenRouter.

Chat completions

```
SELECT aidb.create_model(
  my_or_chat,
  openrouter_chat,
  config      => aidb.openrouter_chat_config(anthropic/claude-3-5-haiku),
  credentials => {"api_key": "sk-or-..."::JSONB
);
```

`aidb.openrouter_chat_config()` parameters:

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	OpenRouter model identifier.
<i>api_key</i>	TEXT	NULL	OpenRouter API key.
<i>url</i>	TEXT	NULL	API endpoint URL override.
<i>max_concurrent_requests</i>	INTEGER	NULL	Maximum concurrent requests.
<i>max_tokens</i>	JSONB	NULL	Max tokens config (from <code>aidb.max_tokens_config()</code>).

Embeddings

```
SELECT aidb.create_model(
  my_or_embedder,
  openrouter_embeddings,
  config      => aidb.openrouter_embeddings_config(mistral/mistral-embed),
  credentials => {"api_key": "sk-or-..."::JSONB
);
```

`aidb.openrouter_embeddings_config()` parameters:

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	OpenRouter embeddings model identifier.
<i>api_key</i>	TEXT	NULL	OpenRouter API key.
<i>url</i>	TEXT	NULL	API endpoint URL override.
<i>max_concurrent_requests</i>	INTEGER	NULL	Maximum concurrent requests.
<i>max_batch_size</i>	INTEGER	NULL	Maximum inputs per batch request.

See the [Models reference](#) for full parameter details on all config helper functions.

6.3 Support matrix

This page lists specific model variants that EDB has tested and confirmed to work with AIDB, split into *Local models* (run on your Postgres host) and *Remote models* (called over the network — NVIDIA NIM microservices, plus OpenAI and Anthropic model families for AIDB’s native agentic adapters). Each table follows the same shape: a model identifier, whether it’s a default, and a description of what the model is good for — use the description to pick the right variant for your use case.

6.3.1 Local models

Default models

AIDB registers these models automatically when the extension is created — no configuration needed to start using them. See *Default models* for how to use them, and *Model providers* for what each provider means.

Model name	Provider	Description
<i>bert</i>	<i>bert_local</i> (Candle)	General-purpose English text embeddings. Good default for most text search and RAG use cases.
<i>clip</i>	<i>clip_local</i> (Candle)	Joint text/image embeddings for multimodal search.
<i>t5</i>	<i>t5_local</i> (Candle)	Lightweight text-to-text generation — translation, summarization, question answering.
<i>llama</i>	<i>llama_instruct_local</i> (Candle)	Instruction-following local chat/completion model.
<i>bge-small-en-v1.5-f16</i>	<i>llamacpp_embeddings</i> (llama.cpp)	Compact, fast English text embeddings.
<i>nomic-embed-text-v1.5-Q</i>	<i>llamacpp_embeddings</i> (llama.cpp)	General-purpose English embeddings with a larger context window (2048 tokens) than <i>bge-small-en-v1.5-f16</i> .
<i>bge-m3-f16</i>	<i>llamacpp_embeddings</i> (llama.cpp)	Multilingual embeddings with an 8192-token context window — good for long or non-English documents.
<i>qwen3-embedding-0.6b-ζ</i>	<i>llamacpp_embeddings</i> (llama.cpp)	Larger embedding model with a 16384-token context window.
<i>qwen3-embedding-4b-Q8</i>	<i>llamacpp_embeddings</i> (llama.cpp)	Largest pre-registered embedding model (20480-token context) — higher quality, more compute.
<i>qwen3.5-0.8b-Q8_0</i>	<i>llamacpp_generate</i> (llama.cpp)	Small, fast chat/completion model with a very large context window (262144 tokens).
<i>llama-3.2-1b-instruct-Q8</i>	<i>llamacpp_generate</i> (llama.cpp)	Compact Meta Llama 3.2 instruction model for lightweight local chat/completion.
<i>lightonocr-2-1b-Q8_0</i>	<i>llamacpp_ocr</i> (llama.cpp)	Compact vision-language model for OCR — extracts text from images.
<i>dummy</i>	<i>dummy</i>	Returns fake data — for testing pipelines without running real inference.

Other tested variants (Candle)

The following HuggingFace model variants have been tested and verified to work with AIDB’s Candle-based local providers, beyond the defaults above. Use any of these identifiers with the corresponding config helper when calling `aidb.create_model()` — see *Custom models*.

BERT (`bert_local`)

Model identifier	Default	Description
<i>sentence-transformers/all</i>	OK	General-purpose English sentence embeddings; fast and small. This is what the default <i>bert</i> model uses.
<i>sentence-transformers/pa</i>	—	Multilingual embeddings — use when your text isn't (only) English.
<i>sentence-transformers/mi</i>	—	Tuned for question-answering and retrieval, rather than general sentence similarity.
<i>sentence-transformers/pa</i>	—	Smaller and faster than the default, at some quality cost.

```
SELECT aidb.create_model(my_bert, bert_local,
    config => aidb.bert_config(sentence-transformers/paraphrase-multilingual-mpnet-
    ↪base-v2));
```

All BERT variants support text embedding (`aidb.encode_text()`).

CLIP (`clip_local`)

Model identifier	Default	Description
<i>openai/clip-vit-base-patch32</i>	OK	The only tested CLIP variant. Joint text/image embeddings. This is what the default <i>clip</i> model uses.

CLIP supports both text embedding (`aidb.encode_text()`) and image embedding (`aidb.encode_image()`).

T5 (`t5_local`)

Model identifier	Default	Description
<i>t5-small</i>	OK	Fastest, smallest T5 variant. This is what the default <i>t5</i> model uses.
<i>t5-base</i>	—	Larger than the default — better quality, slower.
<i>t5-large</i>	—	Largest T5 variant — highest quality, most resource use.

T5 variants support both text encoding (`aidb.encode_text()`) and text generation (`aidb.generate_text()`).

Llama (`llama_instruct_local`)

Model identifier	Default	Description
<i>TinyLlama/TinyLlama-1.1B-Chat</i>	OK	Small, fast instruction-following chat model. This is what the default <i>llama</i> model uses.
<i>HuggingFaceTB/SmolLM2-135M</i>	—	Extremely small and fast — lower quality, lowest resource use of these variants.
<i>HuggingFaceTB/SmolLM2-360M</i>	—	Small — a step up in quality from the 135M variant.

continues on next page

Table 26 – continued from previous page

Model identifier	Default	Description
<i>Hugging-FaceTB/SmolLM2-1.7B-I</i>	—	Largest of these variants — better quality than the default, at higher resource use.

All Llama variants support text generation (`aidb.generate_text()`).

!!!note These are the pre-vetted variants confirmed to work with AIDB. You aren't limited to these — any compatible HuggingFace model can be used with the appropriate config helper, but only the variants listed here have been tested by EDB.

Other tested variants (llama.cpp)

Reranking (`llamacpp_reranking`)

Model identifier	Default	Architecture	Description
<i>gpustack/bge-reranker-v2-m3-GC</i>	—	Cross-encoder	Multilingual cross-encoder reranker. Scores each query/document pair directly via a classification head.
<i>ggml-org/Qwen3-Reranker-t</i>	—	Decoder-only	Qwen3-Reranker family. Scores each pair by reading the model's yes/no verdict-token logits — see llamacpp_reranking .

Neither variant is pre-registered as a default model. Use either identifier with `jsonb_build_object()` when calling `aidb.create_model()` — see [llamacpp_reranking](#).

6.3.2 Remote models

Remote models are called over the network rather than run on your Postgres host: NVIDIA NIM microservices and chat/agent models from OpenAI and Anthropic — reachable through several providers, including the native `openai_responses` and `anthropic_messages` adapters.

NVIDIA NIM

AIDB supports NVIDIA NIM microservices via several providers — see [NVIDIA NIM](#) for how to register one. Models not pre-deployed by Hybrid Manager's Model Serving can be self-hosted using KServe and registered with AIDB using `aidb.create_model()`.

For the full list of supported NIM models per provider (completions, embeddings, CLIP, OCR, and reranking), including which are pre-deployed by EDB Hybrid Manager's Model Serving, see [GPU recommendations for default NIM models](#).

OpenAI and Anthropic models

Each model identifier below can be registered under more than one AIDB provider — which one to pick depends on how you're using the model, not which model it is:

- ```openai_responses``` (+ `openai_responses_azure`) and ```anthropic_messages``` (+ `anthropic_messages_azure` / `anthropic_messages_bedrock`) — AIDB's native, agent-ready providers. Real tools /tool_choice/response_format support, both for direct calls and for agents created with [Managing agents](#).

- `openai_completions` — OpenAI’s older Chat Completions API. Works with the same OpenAI models, and is a perfectly good choice for plain text generation (`aidb.generate_text()`, summarization, pipeline steps), but rejects `tools` / `tool_choice` / `response_format` directly, and only *simulates* tool calling (via prompt injection) when backing an agent.
- `openrouter_chat` — a single gateway to OpenAI, Anthropic, and other vendors’ models (including everything below) via OpenRouter’s own model slugs, for example `openai/gpt-4o` or `anthropic/claude-opus-4-6`. Same simulated, not native, tool calling as `openai_completions`.

See [OpenAI Responses API](#) and [Anthropic Messages API](#) for the full comparison and registration examples. The tables below aren’t an exhaustive catalog of every model each vendor offers; they call out commonly used families as a starting point, covering both current flagship/efficient tiers and a longer-established variant for reference — check the vendor’s own model documentation for the full, current lineup and exact capabilities.

OpenAI

Model identifier	Native provider(s)	Also usable via	Cloud host support (native)			Description
<i>gpt-5.6-sol-terra / -luna</i>	<i>openai_responses, openai_responses_azure</i>	<i>openai_completions, openrouter_chat</i>	Direct, Foundry	Azure	AI	The GPT-5.6 family, offered in three tiers trading off capability, latency, and cost. Check OpenAI's docs for how the tiers compare.
<i>gpt-5.4-mini</i>	<i>openai_responses, openai_responses_azure</i>	<i>openai_completions, openrouter_chat</i>	Direct, Foundry	Azure	AI	Smaller, cost-efficient tier of the GPT-5.4 generation — solid tool-calling accuracy at a fraction of flagship cost.
<i>gpt-4o</i>	<i>openai_responses, openai_responses_azure</i>	<i>openai_completions, openrouter_chat</i>	Direct, Foundry	Azure	AI	Long-established multimodal model. A stable, well-known anchor when you don't need the newest GPT-5.x tier.

Anthropic

Model identifier	Native provider(s)	Also usable via	Cloud host support (native)	Description
<i>claude-opus-4-6</i>	<i>anthropic_messages, anthropic_messages_azure, anthropic_messages_bedrock</i>	<i>open-router_chat</i>	Direct, Azure AI Foundry, AWS Bedrock	Flagship Claude model for the most demanding reasoning and agentic tool-calling tasks.

continues on next page

Table 29 – continued from previous page

Model identifier	Native provider(s)	Also usable via	Cloud host support (native)	Description
<i>claude-sonnet-5</i>	<i>anthropic_messages, anthropic_messages_azure, anthropic_messages_bedrock</i>	<i>openrouter_chat</i>	Direct, Azure AI Foundry, AWS Bedrock	Balanced mid-tier model — strong general-purpose choice for agents and everyday chat/completion.
<i>claude-haiku-4-5</i>	<i>anthropic_messages, anthropic_messages_azure, anthropic_messages_bedrock</i>	<i>openrouter_chat</i>	Direct, Azure AI Foundry, AWS Bedrock	Fastest, most cost-efficient current Claude tier — good for high-volume or latency-sensitive tool calls.
<i>claude-3-5-haiku</i>	<i>anthropic_messages, anthropic_messages_azure, anthropic_messages_bedrock</i>	<i>openrouter_chat</i>	Direct, Azure AI Foundry, AWS Bedrock	Long-established, widely used variant. A stable, well-known anchor when you don't need the newest Claude tier.

Anthropic has no older completions-style provider the way OpenAI has `openai_completions` — `anthropic_messages` is Claude's only direct AIDB provider, alongside the `openrouter_chat` gateway option.

!!!note A model's availability on Azure AI Foundry or AWS Bedrock can lag behind its release on the vendor's own direct API, especially for very recently released tiers. If `openai_responses_azure` or `anthropic_messages_bedrock` rejects a model you expect to work, confirm current availability with the respective cloud provider.

Configuring external storage

AIDB pipelines can read from two types of data source:

- **Postgres tables** — reference a table directly by name using the `source` parameter in `aidb.create_pipeline()`. See *Creating pipelines*.
- **External storage volumes** — connect S3-compatible object stores, Google Cloud Storage, Azure, or local file systems via PGFS. The rest of this page covers how to set this up.

External storage is accessed through the Postgres File System (PGFS) extension, which maps external storage into Postgres as storage locations. AIDB then wraps each storage location in a *volume* that pipelines reference by name. A volume is a *foreign table*, so you can list, read, write, and delete objects with standard SQL.

7.1 How it works

Connecting external storage to an AIDB pipeline involves two objects:

- **PGFS storage location** — Defines the external storage provider: its URI, credentials, and connection options. Created with `pgfs.create_storage_location()`.
- **AIDB volume** — A foreign table that exposes the contents of a PGFS storage location as rows. Stores the data format (`Text`, `Image`, `Pdf`) and an optional sub-path. Created with `aidb.create_volume()`.

Once a volume exists, reference it as the `source` in `aidb.create_pipeline()` exactly as you would a Postgres table, or query it directly with SQL.

!!!note The PGFS extension must be installed before creating storage locations. See *Configuring AIDB*.

7.2 Step 1: Creating a storage location

Use `pgfs.create_storage_location()` to define the connection to external storage. The `url` parameter identifies the storage backend and path; the `options` JSONB object carries provider-specific settings such as region; the `credentials` JSONB object carries authentication secrets such as access keys.

!!!note Explicit credentials in the `credentials` parameter aren't the only way to authenticate. If you omit them, S3, Azure, and GCS storage locations fall back to the respective cloud provider's own native credential resolution

(IAM roles, managed/workload identity, instance metadata, Application Default Credentials) — useful when Postgres is already running with a cloud identity attached, so there's no secret to store at all. See the PGFS documentation for details.

7.2.1 S3-compatible object store

```
-- Private S3 bucket with credentials
SELECT pgfs.create_storage_location(
    name      => my_s3_location,
    url       => s3://my-bucket/my-folder,
    options   => {"region": "us-east-1"},
    credentials => {"access_key_id": "<key>", "secret_access_key": "<secret>"}
    -- or omit the credentials argument to use the cloud providers native credential_
    ↳ chain (IAM role, managed identity, etc.) instead
);

-- Public S3 bucket (no credentials required)
SELECT pgfs.create_storage_location(
    name      => my_public_bucket,
    url       => s3://aidb-rag-app,
    options   => {"region": "eu-central-1", "skip_signature": "true"}
);
```

7.2.2 Local file system

For local file system access, declare the allowed base paths in `postgresql.conf` before creating the storage location. PGFS restricts access to these paths for security.

```
# postgresql.conf

pgfs.allowed_local_fs_paths = /tmp/pgfs
```

After restarting Postgres, create the storage location using a `file://` URI:

```
SELECT pgfs.create_storage_location(
    name => local_tmp_pgfs,
    url  => file:///tmp/pgfs/
);
```

For full details on storage location options for S3, GCS, and Azure, see the PGFS documentation .

!!!Note Restricting egress to your object store PGFS supports an optional outbound allowlist (`pgfs.egress_allowlist`, with `edb.egress_allowlist` as a shared fallback) that restricts which hosts PGFS is allowed to reach when accessing storage locations. This is independent of AIDB's allowlist for model-provider traffic; both can be configured together. See PGFS — Network egress and *AIDB — Network egress* .

7.3 Step 2: Creating a volume

Use `aidb.create_volume()` to create a foreign table that exposes a PGFS storage location to AIDB. The volume name is the foreign-table name that you reference in pipelines and SQL queries.

```
SELECT aidb.create_volume(
    name      => my_volume,
```

(continues on next page)

(continued from previous page)

```

server_name => my_s3_location,
path        => /,
data_format => Text
);

```

Parameters:

Parameter	Description
<i>name</i>	Unique name for the volume. Must be a valid unquoted PostgreSQL identifier (lowercase, starts with a letter or underscore, contains only letters, digits, and underscores — no dashes or spaces). Used as the foreign-table name.
<i>server_name</i>	Name of the PGFS storage location to attach.
<i>path</i>	Optional sub-path within the storage location. See Path handling below.
<i>data_format</i>	The type of data in this volume. One of <i>Text</i> , <i>Image</i> , or <i>Pdf</i> . Pipelines use this to choose the correct parsing step.

!!!note `data_format` is metadata — it tells AIDB how to treat the objects, but doesn't filter them. Ensure the volume only contains objects of the declared format.

7.3.1 Path handling

The `path` argument is canonicalized at creation time: any leading and trailing slashes are stripped, then exactly one trailing slash is appended. This means the following all store as `foo/`:

- `foo`
- `/foo`
- `foo/`
- `/foo/`

Passing `NULL` or `/` stores as `/`, which targets the storage location's root. The canonical value is what you see in `aidb.volumes` and what's used to compute object keys.

7.3.2 Example: PDFs in S3

```

SELECT pgfs.create_storage_location(
  pdf_bucket,
  s3://my-docs-bucket,
  options      => {"region": "us-east-1"},
  credentials => {"access_key_id": "<key>", "secret_access_key": "<secret>"}
  -- or omit the credentials argument to use the cloud providers native credential_
  ↪chain instead
);

SELECT aidb.create_volume(pdf_volume, pdf_bucket, /, Pdf);

```

7.3.3 Example: Images in a local directory

```

SELECT pgfs.create_storage_location(local_tmp_pgfs, file:///tmp/pgfs/);

SELECT aidb.create_volume(ocr_input_volume, local_tmp_pgfs, ocr_input/, Image);

```

7.4 Step 3: Using the volume as a pipeline source

Set the `source` parameter in `aidb.create_pipeline()` to the volume name. Volume sources work the same as table sources from the pipeline's perspective:

```
SELECT aidb.create_pipeline(
  name      => my_pdf_pipeline,
  source    => pdf_volume,
  step_1     => ParsePdf,
  step_2     => ChunkText,
  step_3     => KnowledgeBase,
  step_3_options => aidb.knowledge_base_config(bert_local, Text)
);
```

See [Create a pipeline](#) for a full walkthrough.

7.5 Managing volumes

7.5.1 Listing volumes

List all volumes via the `aidb.volumes` view:

```
SELECT * FROM aidb.volumes;
__OUTPUT__
 schema | volume      | storage_location | path
-----+-----+-----+-----
 public | pdf_volume  | pdf_bucket      | /
 public | text_bucket | text_bucket     | aidb-rag-app/
(2 rows)
```

The view returns four columns: `schema`, `volume`, `storage_location`, and `path` (canonicalized).

7.5.2 Deleting a volume

```
SELECT aidb.delete_volume(my_volume);
```

!!!note Deleting a PGFS storage location also deletes all volumes built on top of it.

7.6 Accessing volume contents with SQL

A volume is a foreign table backed by PGFS, so the storage location's objects are exposed as rows. The columns are:

Column	Type	Description
<i>key</i>	TEXT	Object key (file path) relative to the volume's <i>path</i> .
<i>size</i>	BIGINT	Object size in bytes.
<i>last_modified</i>	TIMESTAMPTZ	Last-modified timestamp reported by the storage backend.
<i>e_tag</i>	TEXT	Entity tag (ETag) reported by the storage backend, when available.

continues on next page

Table 2 – continued from previous page

Column	Type	Description
<i>body</i>	BYTEA	Object contents. Only populated when the query includes <i>WHERE key = '<exact-key>'</i> (see note below).

!!!note The *body* column is only fetched when you select it **and** filter on a single key with *WHERE key = '...'*. Selecting *body* without a key filter returns *NULL* — this is a safeguard against accidentally downloading every object in the volume.

7.6.1 Listing objects in a volume

```
SELECT key, size, last_modified
FROM my_volume
ORDER BY key;
```

```
__OUTPUT__
      key              | size | last_modified
-----+-----+-----
Test 1.html           | 1093 | 2026-05-14 15:49:32+00
Test 2.html           | 1563 | 2026-05-14 15:50:02+00
Test 3.html           | 1395 | 2026-05-14 15:49:32+00
(3 rows)
```

LIMIT is pushed down to the storage backend, so it's safe to peek at large buckets:

```
SELECT key, size FROM my_volume LIMIT 5;
```

7.6.2 Reading a single object

Filter on the exact key to fetch the object body:

```
-- Read a binary file as BYTEA
SELECT body FROM my_volume WHERE key = report.pdf;

-- Read a text file as text
SELECT convert_from(body, utf8)
FROM my_volume
WHERE key = notes.txt;
```

7.6.3 Writing, overwriting, and deleting objects

Volumes support standard INSERT, UPDATE, and DELETE. *body* is BYTEA; cast text to bytea as needed.

```
-- Upload a new object
INSERT INTO my_volume (key, body)
VALUES (hello.txt, Hello, world::bytea);

-- Overwrite an existing object
UPDATE my_volume
SET body = Hello again::bytea
WHERE key = hello.txt;
```

(continues on next page)

(continued from previous page)

```
-- Delete an object
DELETE FROM my_volume WHERE key = hello.txt;
```

!!!note Object-store writes aren't transactional. A successful INSERT , UPDATE , or DELETE isn't rolled back if the surrounding PostgreSQL transaction aborts.

Renaming a key (UPDATE my_volume SET key = ...) isn't supported.

With AIDB, you can configure and run agents within your database. They have secure, local access to your data and depending on where your model runs, data never has to leave Postgres.

An **agent** is a named, reusable configuration — instructions, a model, and a set of tools it’s allowed to call — that AIDB runs through a ReAct-style reasoning loop: the model thinks, optionally calls one or more tools, observes the results, and repeats until it has an answer.

!!!note This is a distinct in-database capability different from [Agent Factory in Hybrid Manager \(HM\)](#) . AIDB’s agents run entirely as SQL function calls inside your Postgres database.

Agents are ordinary Postgres objects, created and queried with SQL. The example below builds a small text-to-SQL agent end to end: it doesn’t know the database schema in advance, discovers it with the *catalog discovery tools* , and answers questions by running real queries against your data.

8.1 Example: a text-to-SQL agent

8.1.1 Setup

Create some sample data, register a model, and create the agent:

```
CREATE TABLE public.foods (  
  id SERIAL PRIMARY KEY,  
  name TEXT NOT NULL,  
  category TEXT NOT NULL,  
  calories INTEGER NOT NULL,  
  price_usd NUMERIC(6,2) NOT NULL,  
  is_vegetarian BOOLEAN NOT NULL DEFAULT false  
);  
  
INSERT INTO public.foods (name, category, calories, price_usd, is_vegetarian) VALUES  
  (Apple, Fruit, 95, 0.50, true),  
  (Banana, Fruit, 105, 0.30, true),  
  (Grilled Chicken Breast, Meat, 165, 3.50, false),
```

(continues on next page)

(continued from previous page)

```

(Salmon Fillet,      Fish,      208, 6.00, false),
(Broccoli,          Vegetable, 55, 1.20, true),
(Cheddar Cheese,    Dairy,     402, 2.80, true),
(Brown Rice,        Grain,     216, 0.80, true),
(Bacon,             Meat,      541, 4.20, false),
(Almonds,           Nuts,      579, 5.50, true),
(Dark Chocolate,    Snack,     546, 3.00, true);

SELECT aidb.create_model(
  gpt_5_6_terra_azure,
  openai_responses,
  config => aidb.openai_responses_config(
    model => gpt-5.6-terra,
    url   => https://my-ai-service.example.com/openai/v1/responses
  ),
  credentials => jsonb_build_object(
    api_key, <your-api-key>
  )
);

SELECT aidb.create_agent(
  name          => text2sql_demo,
  instructions => You are a helpful, read-only database assistant that answers_
↳questions by      || querying the database. You do not know this databases schema_
↳in                || advance -- before answering, use the catalog_list_objects_
↳tool to           || discover which tables exist, then use catalog_get_object_
↳details to        || inspect the columns of any table you intend to query. Never_
↳guess a           || table or column name -- always verify it via these catalog_
↳tools first.      || Once you know the relevant table and columns, use the run_sql_
↳query             || tool to fetch the actual data needed to answer the question --
↳never             || state a specific result as fact unless it came from a_
↳successful        || run_sql_query call. You only have read-only access and cannot_
↳modify            || any data. Once a query has given you enough information to_
↳answer,           || respond immediately -- do not repeat a tool call you have_
↳already made      || with the exact same arguments.,
  model          => gpt_5_6_terra_azure,
  tools          => ARRAY[run_sql_query, catalog_list_objects, catalog_get_object_
↳details]
);

```

8.1.2 Conversation

Ask the agent a question about the data. Behind the scenes it lists the available tables, inspects the `foods` columns, runs a query, and answers from the result:

```
SELECT * FROM aidb.agent_converse(text2sql_demo, Which vegetarian food item has the
↳fewest calories?);
__OUTPUT__
      conversation_id      message
-----
↳
+-----+
Broccoli has the fewest calories among vegetarian items, with **55 calories**. |
↳08571a03-0758-4cda-8595-607e2a69c8ce |
(1 row)
```

To watch the reasoning loop's individual tool calls and results, re-run with `debug => true` — see [Invoking agents](#).

8.1.3 Comparison

The same question, asked to the same model directly — without the agent's reasoning loop and tools — can only be answered from the model's training data. It has no idea your `foods` table exists:

```
SELECT aidb.generate_text(gpt_5_6_terra_azure, Which vegetarian food item has the
↳fewest calories?);
__OUTPUT__
      generate_text
-----
↳
+-----+
Among common solid vegetarian foods, **raw watercress** is one of the lowest-calorie
↳options, at about **11 calories per 100 g**.+
+
[...]
```

The agent grounded its answer in your actual data; the bare model call could only guess.

An agent's `tools` array draws from AIDB's unified tool catalog — a built-in AIDB operation, a custom parameterized SQL query you register, or an operation imported from an external MCP server. See [Tools and delegates](#) for the full catalog.

8.2 Documentation map

Page	What it covers
Managing agents	<code>aidb.create_agent()</code> , updating and deleting agents, roles, structured output
Invoking agents	<code>aidb.agent_converse()</code> , conversations and sessions, read-only mode, debug mode, budgets and limits
Delegation	Presenting delegates to the model, continuing delegate conversations, what propagates across a handoff, delegation depth

For full parameter tables and return types, see [Agents reference](#).

8.2.1 Managing agents

An agent is a row in `aidb.agents`, created with `aidb.create_agent()`. Nothing runs until you call *Invoking agents* — creating an agent just stores its configuration.

```
SELECT aidb.create_agent(
  name                => db_helper,
  instructions         => You are a helpful assistant that answers questions about_
  ↪this database. Be concise.,
  model               => my_gpt,
  tools               => ARRAY[run_sql_query], -- optional; tool names from aidb.
  ↪tools. Default: NULL (no tools)
  delegates           => NULL,                -- optional; names of other_
  ↪agents to delegate to. Default: NULL (no delegation)
  role                => NULL,                -- optional; Postgres role tool_
  ↪calls run as. Default: NULL (run as the calling role)
  output_type         => NULL,                -- optional; structured output_
  ↪schema from aidb.output_type(). Default: NULL (plain-text answer)
  input_token_budget  => NULL,                -- optional; input token cap per_
  ↪agent_converse call. Default: NULL (no limit)
  output_token_budget => NULL,                -- optional; output token cap_
  ↪per agent_converse call. Default: NULL (no limit)
  max_iterations      => NULL,                -- optional; reasoning-round cap_
  ↪per call. Default: NULL (a hard ceiling of 25 always applies)
  timeout             => NULL,                -- optional; wall-clock seconds_
  ↪cap per call. Default: NULL (300 seconds applied at runtime)
  budget_strategy     => attempt_complete,    -- optional; behavior when a_
  ↪budget is exceeded. Default: attempt_complete
  preset              => NULL                -- optional; reserved for future_
  ↪use, has no effect today. Default: NULL
);
```

Returns no rows on success. On failure (for example, the name is already taken), it returns a single row with an error column set — it never raises, so a batch of `create_agent` calls can't abort partway through:

```
SELECT * FROM aidb.create_agent(db_helper, Be helpful., my_gpt);

__OUTPUT__

      error
-----
agent db_helper already exists
(1 row)
```

Choosing a model

`model` must name an already-registered *Choosing a model* that's capable of generating text and following instructions. Embedding, reranking, OCR, and multimodal-embedding providers (`bert_local`, `openai_embeddings`, `nim_reranking`, `clip_local`, and so on) can't back an agent at all, and `t5_local` is rejected outright since it has no instruction-following or system-prompt support. Every other text-generation provider works, split into two tiers by tool-calling quality:

Preferred — these send real, provider-native tool-call requests:

- `openai_responses` (+ `openai_responses_azure`)
- `anthropic_messages` (+ `anthropic_messages_azure` / `anthropic_messages_bedrock`)

- `llamacpp_generate`

Supported — these work, but AIDB simulates tool calling by describing the available tools in the prompt text and parsing a tool call back out of the model’s plain-text reply:

- `openai_completions`
- `completions` (any other OpenAI-Chat-Completions-compatible endpoint)
- `nim_completions`
- `openrouter_chat`
- `gemini`
- `llama_instruct_local`

See [Tool calling and structured output](#) for more on the preferred/supported distinction. The model doesn’t need to exist yet when you call `create_agent` — only `agent_converse` requires it to be resolvable.

Tools and delegates

`tools` is an array of names from the [Tools reference](#) (`aidb.tools`) the agent is allowed to call; `delegates` is an array of names of other agents it may [Delegation](#). Both are validated against `aidb.tools` / `aidb.agents` at `create_agent` / `update_agent` time — an unknown name is rejected immediately rather than surfacing later inside `agent_converse`. Omit either for an agent with no tools or no delegates.

```
SELECT aidb.create_agent (
  name          => help_desk,
  instructions   => Help users with general questions, delegating data questions to a
  ↳specialist.,
  model         => my_gpt,
  tools         => ARRAY[run_sql_query],
  delegates     => ARRAY[sql_specialist]
);
```

Permissions

By default, an agent’s tool calls run as whichever Postgres role calls `agent_converse`. Pass `role` to pin an agent to a specific role instead — for example, to give it narrower table permissions than the calling application user has. The caller must already be a member of that role (`pg_has_role(..., 'MEMBER')`); `create_agent` / `update_agent` reject an agent configured with a role you don’t belong to.

```
SELECT aidb.create_agent (
  name          => readonly_reporter,
  instructions   => Answer questions about sales data. Never modify data.,
  model         => my_gpt,
  tools         => ARRAY[run_sql_query],
  role         => reporting_role
);
```

Structured output

By default, `agent_converse`’s answer is plain text. To get back structured JSON instead, build a schema with `aidb.output_type()` and `aidb.output_field()`:

```
SELECT aidb.create_agent (
  name          => sentiment_tagger,
  instructions   => Classify the sentiment of the given text.,
  model         => my_gpt,
  output_type   => aidb.output_type(
    aidb.output_field(sentiment, TEXT, One of: positive, negative, neutral),
    aidb.output_field(confidence, FLOAT)
  )
);
```

`aidb.output_field()`’s `description` argument is optional. A call-time `output_type` argument to `agent_converse` overrides the agent’s configured schema for that one call only.

Budgets, iterations, and timeouts

`input_token_budget`, `output_token_budget`, `max_iterations`, and `timeout` (seconds) cap how much a single `agent_converse` call can cost and how long it can run; `budget_strategy` controls what happens when a budget is exceeded. All are optional — see [Budgets and limits](#) for the full behavior, since it’s a property of how a conversation runs, not of the agent’s static configuration.

```
SELECT aidb.create_agent (
  name          => cost_capped_helper,
  instructions   => Answer questions about the orders table.,
  model         => my_gpt,
  input_token_budget => 20000,    -- INTEGER; input tokens allowed per agent_
↪converse call
  output_token_budget => 2000,    -- INTEGER; output tokens allowed per agent_
↪converse call
  max_iterations  => 10,         -- INTEGER; reasoning rounds allowed per call (a_
↪hard ceiling of 25 always applies)
  timeout        => 60,         -- INTEGER; wall-clock seconds allowed per call
  budget_strategy => error      -- TEXT; one of ignore, error, summarize, attempt_
↪complete
);
```

Updating and deleting agents

`aidb.update_agent()` takes the same fields as `create_agent`, all optional — only the fields you pass change; everything else keeps its current value:

```
SELECT aidb.update_agent(db_helper, model => my_new_gpt, max_iterations => 15);
```

There’s no dedicated “list agents” function — `aidb.agents` is a normal table, so `SELECT * FROM aidb.agents` lists them. See [Agents reference](#) for its full column list.

`aidb.delete_agent()` removes an agent by name:

```
SELECT aidb.delete_agent(db_helper);
```

If the agent has any conversation history, this fails by default (its internal task records are referenced by foreign key) — pass `force => true` to delete that internal history alongside the agent. The conversation *transcript* itself is untouched either way and remains readable with [Retrieving conversations](#) even after the agent that produced it has been deleted.

```
SELECT aidb.delete_agent(db_helper, force => true);
```

!!!note `create_agent` / `update_agent` also accept a `preset` argument (for example, `'text_to_sql'`). It's stored on the agent but doesn't currently change its behavior — reserved for future use.

8.2.2 Invoking agents

`aidb.agent_converse()` sends a prompt to an agent and runs its reasoning loop — think, optionally call tools, observe, repeat — until it produces an answer or hits a limit:

```
SELECT * FROM aidb.agent_converse(
  agent_name      => db_helper,
  prompt          => How many rows are in the orders table?,
  conversation_id => NULL, -- optional; continue an existing conversation by id.
  ↳Default: NULL (starts a new conversation)
  read_only       => NULL, -- optional; force read-only mode on/off. Default: NULL
  ↳(auto: on for a read replica, off otherwise)
  output_type     => NULL, -- optional; override the agents structured output
  ↳schema for this call. Default: NULL (use the agents own configuration)
  debug          => NULL -- optional; also emit every action as a NOTICE.
  ↳Default: NULL (false)
);
```

message	conversation_id	error
The orders table has 4,213 rows. (1 row)	8f14e45f-ceea-467e-9575-a3d1a2c1a123	

Like *Managing agents*, this never raises for an agent-logic failure (an unknown agent, a mid-loop reasoning failure, a budget that was exceeded) — `error` is populated on the returned row instead, so the call's transaction (and everything already logged for that attempt) still commits.

Pass `output_type` to override the agent's configured output schema for just this call — see *Structured output* for how to build one.

Conversations

Every `agent_converse` call belongs to a conversation, identified by `conversation_id`. Omit it on the first call and AIDB generates one; pass that same id back in on later calls to continue the conversation, with the agent's own prior history as context:

```
SELECT * FROM aidb.agent_converse(db_helper, And how many are still pending?,
  conversation_id => 8f14e45f-ceea-467e-9575-a3d1a2c1a123);
```

If you'd rather have the id in hand before sending a first prompt (for example, to log it alongside a UI session), mint one upfront with `aidb.start_agent_session()`:

```
SELECT * FROM aidb.start_agent_session(db_helper);
SELECT * FROM aidb.agent_converse(db_helper, How many rows are in the orders table?,
  conversation_id => <the id start_agent_session returned>);
```

`start_agent_session` only validates the agent name and hands back a fresh id. A conversation doesn't actually exist until the first `agent_converse` call that uses it.

Retrieving conversations

`aidb.get_conversation()` returns a conversation’s user prompts and final answers, in order. It does not return tool calls or intermediate reasoning steps, which are activity details rather than “the conversation”:

```
SELECT * FROM aidb.get_conversation(8f14e45f-ceea-467e-9575-a3d1a2c1a123);
```

__OUTPUT__

message_id	task_id	action_type	role	sender_id	contents
...	...	user_prompt	user	postgres	How many rows are in the orders table?
...	...	answer	agent	db_helper	The orders table has 4,213 rows.

(2 rows)

`aidb.get_message()` retrieves one message’s plain-text contents by its `message_id`. Both are backed by the `aidb.conversation_log` view, which you can also query directly; `aidb.conversations` aggregates it into one row per conversation (message count, status, last message). See [Agents reference](#) for their full column lists.

Read-only mode

Pass `read_only => true` to run an agent without persisting anything — no conversation history is written, and `conversation_id` always comes back `NULL`, so a read-only run can’t be resumed later. It’s automatic (no need to pass it explicitly) when the database is a read replica.

A read-only run enforces two things at once: every tool call executes under `SET LOCAL transaction_read_only = on` for that statement, and any tool that isn’t provably read-only is excluded from what the agent can even see or call in the first place. MCP tools are always excluded in read-only mode, regardless of how they’re classified — an external server’s own read/write behavior isn’t something AIDB can verify. If a [Delegation](#) happens during a read-only run, the delegate’s own sub-call is also forced read-only.

Debug mode

Pass `debug => true` to also emit every action — model requests/responses, tool calls and their results, reasoning steps — as a Postgres `NOTICE` as it happens, in addition to (not instead of) the normal history written to `aidb_internal.action_log`. This is useful for watching an agent work interactively (psql prints `NOTICE` to the terminal by default) or for debugging why it took the path it did.

```
SELECT * FROM aidb.agent_converse(db_helper, How many rows are in the orders table?,
debug => true);
```

NOTICE: [tool_call] run_sql_query({"query": "SELECT count(*) FROM orders"})

NOTICE: [tool_response] run_sql_query -> [{"count": 4213}]

NOTICE: [answer] The orders table has 4,213 rows.

Each `NOTICE` includes that action’s request/response payload — prompts, tool arguments, and results — truncated but **not redacted**. Avoid `debug => true` in contexts where those details (which may include data returned by your own tools) shouldn’t be visible to whoever can see the client output or server log. It applies to any delegate sub-calls the run makes, too.

Budgets and limits

Each `agent_converse` call is capped by four budgets, all optional and set on `aidb.create_agent` :

Parameter	What it caps
<code>input_token_budget</code>	Input tokens for the call.
<code>output_token_budget</code>	Output tokens for the call.
<code>max_iterations</code>	Reasoning rounds — one model call plus any tools it calls in that round, not individual tool calls. A hard ceiling of 25 always applies, regardless of configuration, as a backstop against a runaway agent.
<code>timeout</code>	Wall-clock seconds for the call. Defaults to 300 seconds if unset.

What happens when a budget is exceeded depends on `budget_strategy` :

Strategy	Behavior
<code>attempt_complete</code> (default)	Grants 3 extra reasoning rounds once a budget is first exceeded, telling the model its time is running out so it can wrap up — then halts with an error if it still hasn't finished.
<code>error</code>	Halts immediately with an error.
<code>summarize</code>	Stops immediately and asks the model for a brief status report of what it had done so far, returning that as the answer instead of an error.
<code>ignore</code>	Logs a one-time warning (as a Postgres <i>WARNING</i>) and keeps going with no limit.

Context length

Long conversations are bounded automatically: once a task's assembled prompt would exceed an internally tracked size estimate, AIDB asks the model to summarize the older part of the history and continues with that summary in place of the original turns. This is basic, automatic bounding to keep long-running conversations from blowing past a model's context window — there's nothing to configure. A `compaction` entry appears in the activity log when this happens (see [Debug mode](#), or query `aidb_internal.action_log` directly for the full activity history behind a conversation — every model request/response and tool call/response, not just the user-facing messages `aidb.get_conversation()` returns).

8.2.3 Delegation

An agent can hand off part of a request to another agent instead of (or alongside) calling tools directly. List the other agent's name in `delegates` when you [Managing agents](#) it:

```
SELECT aidb.create_agent (
  name          => sql_specialist,
  instructions => You answer questions about the orders and customers tables.
  ↳precisely.,
  model         => my_gpt,
  tools         => ARRAY[run_sql_query]
);

SELECT aidb.create_agent (
  name          => help_desk,
  instructions => You help users with general questions, delegating data questions.
  ↳to the SQL specialist.,
  model         => my_gpt,
  delegates     => ARRAY[sql_specialist]
```

(continues on next page)

(continued from previous page)

```
);
SELECT * FROM aidb.agent_converse(help_desk, How many orders came in last week?);
```

Presenting delegates to the model

Each configured delegate becomes an ordinary function-call tool, named after the delegate agent, with two arguments: `prompt` (required — the request to hand off) and `conversation_id` (optional — see [Continuing a delegate conversation](#) below). The delegate’s own `instructions` become that tool’s description, so the delegating model sees them as “what this delegate is for” — write an agent’s instructions with that dual purpose in mind if you expect it to be delegated to.

A delegate name must not collide with one of the agent’s own `tools`, and it’s validated to be a currently-registered agent at `create_agent` / `update_agent` time, the same as `tools` is.

Continuing a delegate conversation

Calling a delegate returns an object with `answer` and `conversation_id`. The model can pass that `conversation_id` back on a later delegate call in the same or a future `agent_converse` run to continue that specific delegate conversation — with its own prior history — instead of starting fresh each time. Omitting it (the common case) starts a new conversation with the delegate on every call.

Propagating behavior across a handoff

- **Read-only mode propagates.** If the delegating run is *Read-only mode*, the delegate’s own sub-call is forced read-only too — a read-only agent can’t use delegation to reach a write through another agent. A read-only delegate call always returns `conversation_id: null` and can’t be continued.
- **Debug mode propagates.** If the top-level call passed `debug => true`, the delegate’s own actions are emitted as NOTICE messages too.
- **Role doesn’t propagate.** A delegate runs under its *own* configured `role` (or the current user, if it has none) — not the delegating agent’s role.
- **Budgets don’t propagate.** Each delegate call gets its own fresh token/iteration budget and timeout, from its own configuration — a parent’s budget doesn’t constrain how much a delegate can do.

Delegation depth

Delegation chains are capped at 10 levels deep (a chain of agents, each delegating to the next). This exists mainly to fail fast on a cycle — one agent delegating back to itself, or two agents delegating back and forth — rather than recursing until Postgres’s stack depth check aborts the backend. Hitting the limit stops the task immediately with an error; it isn’t retried the way an ordinary tool failure might be.

A **tool** is anything an agent can invoke by name. Every tool AIDB knows about — built-in, custom, or imported from an MCP server — is listed in one place, `aidb.tools`:

```
SELECT name, tool_type, read_only FROM aidb.tools;
```

```
__OUTPUT__
      name          | tool_type | read_only
-----+-----+-----
 chunk_text         | native_tool | t
 create_agent       | native_tool | f
 ...
```

Tools can be invoked in three different ways:

1. natively by the internal agents in AIDB.
2. by external agents via MCP.
3. through SQL for testing and development or other custom use cases.

```
SELECT aidb.run_tool(catalog_list_objects, {"schema": "public", "object_type": "table"
→ "});
```

```
__OUTPUT__
      run_tool
-----
 {"comment": null, "truncated": false, "object_name": "my_table", "object_type":
→ "table", "schema_name": "public"}, ...
```

AIDB has three kinds of tools:

- **``native\_tool``** — AIDB’s own built-in catalog. Always available; can’t be added, removed, or reconfigured. See [Native tools](#).
- **``sql\_tool``** — a parameterized SQL query you register yourself. See [Custom SQL tools](#).

- ```mcp_tool``` — a tool imported from an external MCP server. See [MCP tools](#).

Give an agent access to a tool by listing its name in `aidb.create_agent`'s `tools` array.

9.1 Documentation map

Page	What it covers
Native tools	AIDB's built-in tool catalog: ad hoc SQL, catalog discovery, semantic knowledge bases, models, volumes, embedding and document processing, agent management, pipelines
Custom SQL tools	<code>aidb.create_sql_tool()</code> , naming, parameter placeholders, read-only enforcement, removing tools
MCP tools	<code>aidb.import_mcp_tools()</code> , authentication, filtering, refreshing, removing servers, exposing AIDB's tools over MCP

For full parameter tables and return types, see [Tools reference](#).

9.1.1 Native tools

AIDB ships with a large built-in catalog of native tools — mostly thin wrappers around existing AIDB functionality, so an agent can operate on pipelines, knowledge bases, models, and volumes the same way you would from SQL directly. This is one of three tool types AIDB supports — see [Tools reference](#) for the full catalog, including [Custom SQL tools](#) and [MCP tools](#). `sleep` is always available to every agent regardless of its configured `tools` list — unless the backing model caps how many tools a request can carry and the agent's other configured tools already fill that cap — since it backs the reasoning loop's own ability to pause and resume. Every other native tool must be explicitly listed in `tools` to be usable.

A tool's exact parameters are always visible with `SELECT params FROM aidb.tools WHERE name = '<name>'`; the tables below give the one-line description and read-only status for each.

Ad hoc SQL

Tool	Read-only	Description
<code>run_sql_query</code>	Yes	Run a read-only SQL query against the database and return its rows as JSON.

Catalog discovery

Introspection of the database catalog: these tools let an agent discover what schemas, tables, views, and functions exist, without having to construct `information_schema/pg_catalog` queries itself.

They're related to the [Semantic knowledge bases](#) below, but the two are complementary rather than interchangeable:

- **Catalog discovery tools** do live, exact lookups against the Postgres catalog. They need no setup, always reflect the current state of the database, and match by exact name or SQL `LIKE` pattern.
- **Semantic knowledge base tools** run semantic (vector) search over a crawled snapshot of the schema. They require a pre-built, opt-in knowledge base (and a re-crawl to pick up structural changes), but can answer natural-language questions like “which table holds customer contact info?” that exact catalog lookups can't.

A typical agent uses both: semantic search to find the right table from a vague description, then catalog discovery to get its exact, current columns and foreign keys.

All four tools are privilege-aware — they only return objects the calling user can actually see. Listings hide Postgres and AIDB internal schemas as well as installed extensions' schemas and objects by default (`catalog_get_object_details` deliberately doesn't hide extension objects: explicitly naming one is asking for it). Listings are capped at 200 rows; every row carries a `truncated` flag that's `true` when more rows exist, signaling the caller to narrow the search with `schema` or `name_pattern`. Optional text parameters treat an empty string the same as an omitted argument.

Tool	Read-only	Description
<i>catalog_list_schemas</i>	Yes	List schemas in the current database. Hides Postgres/AIDB internal schemas and installed extensions' schemas unless <i>include_system</i> is set.
<i>catalog_list_objects</i>	Yes	List tables, views, materialized views, foreign tables, and/or functions. Narrow with <i>schema</i> , <i>object_type</i> , and/or <i>name_pattern</i> .
<i>catalog_get_object_details</i>	Yes	Get full detail on one object by name: columns, primary/unique/foreign keys for relations; arguments and return type for functions.
<i>catalog_list_relations</i>	Yes	List foreign-key relationships — every FK in a schema, or only edges touching one specific table, in either direction.

Semantic knowledge bases

Schema-search knowledge bases to allow semantic search based discovery of tables, columns, and other DB objects. Distinct from the knowledge bases for user data that pipelines can create.

Tool	Read-only	Description
<i>create_semantic_kb</i>	No	Create a semantic knowledge base over one or more schemas for natural-language schema search.
<i>delete_semantic_kb</i>	No	Delete a semantic knowledge base by name.
<i>list_semantic_kbs</i>	Yes	List all registered semantic knowledge bases.
<i>get_semantic_kb</i>	Yes	Get a registered semantic knowledge base's configuration by name.
<i>refresh_semantic_kb</i>	No	Re-crawl a semantic knowledge base's schemas to pick up structural changes.
<i>update_semantic_kb_auto_processing</i>	No	Change a semantic knowledge base's auto-processing mode.
<i>get_column_definitions</i>	Yes	Search a semantic knowledge base for matching column/table definitions.
<i>get_metadata</i>	Yes	Search a semantic knowledge base's full schema metadata (columns, tables, comments).
<i>get_entity_definitions</i>	Yes	Search a semantic knowledge base for matching table/view definitions.
<i>semantic_kb_search</i>	Yes	Run a composite semantic search over a knowledge base, returning one ranked list of matching tables, views, columns, and aliases with a source-type discriminator.
<i>search_by_comment</i>	Yes	Search a semantic knowledge base's schema/table/column comments by similarity.
<i>semantic_kb_stats</i>	Yes	Get row/entity counts for a semantic knowledge base.

Models

See [Integrating models](#) .

Tool	Read-only	Description
<i>list_models</i>	Yes	List all registered models.
<i>get_model</i>	Yes	Get a registered model's configuration by name.
<i>create_model</i>	No	Register a new model with a provider and configuration.
<i>delete_model</i>	No	Delete a registered model by name.
<i>sync_hcp_models</i>	No	Sync models available from the configured Hybrid Manager (HM) into the model registry.
<i>list_hcp_models</i>	Yes	List models already synced from HM.
<i>get_hcp_models</i>	Yes	List models available from HM, synced or not.
<i>create_hcp_model</i>	No	Register a model from HM under a local name.

Volumes

See [Configuring external storage](#) .

Tool	Read-only	Description
<i>list_volumes</i>	Yes	List all registered volumes.
<i>create_volume</i>	No	Register a new volume backed by an object store.
<i>delete_volume</i>	No	Delete a volume by name.
<i>list_volume_content</i>	Yes	List the files stored in a volume.
<i>read_volume_file</i>	Yes	Read a file's contents from a volume.
<i>write_volume_data</i>	No	Write data to a file in a volume.
<i>delete_volume_file</i>	No	Delete a file from a volume.

Embedding and document processing

See [SQL functions](#) .

Tool	Read-only	Description
<i>encode_text</i>	Yes	Compute a text embedding vector for document/search-index storage.
<i>encode_text_query</i>	Yes	Compute a text embedding vector tuned for querying (vs. indexing).
<i>encode_image</i>	Yes	Compute an image embedding vector.
<i>generate_text</i>	Yes	Generate text from a language model.
<i>get_model_description</i>	Yes	Get a human-readable description of a model.
<i>list_model_providers</i>	Yes	List the foreign-server-backed model providers available for registering models against.
<i>rerank_text</i>	Yes	Rerank a list of candidate texts against a query using a reranker model.
<i>get_adapter_embedding_dimensions</i>	Yes	Get a model's output embedding dimensionality for text or image input.

continues on next page

Table 7 – continued from previous page

Tool	Read-only	Description
<i>chunk_text</i>	Yes	Split text into chunks suitable for embedding.
<i>parse_html</i>	Yes	Parse HTML into plaintext or markdown.
<i>parse_pdf</i>	Yes	Extract text from a PDF, one row per page.
<i>pdf_to_image</i>	Yes	Render a PDF's pages to images, one row per page.
<i>perform_ocr</i>	Yes	Run OCR over an image or scanned document.
<i>summarize_text</i>	Yes	Summarize text using a language model.

Agent management

See [Managing agents](#) . Letting an agent create or reconfigure *other* agents (including itself) is a powerful, self-modifying capability — grant these tools deliberately.

Tool	Read-only	Description
<i>list_agents</i>	Yes	List all registered agents.
<i>create_agent</i>	No	Create a new agent. Returns a single row with an <i>error</i> column (<i>NULL</i> on success).
<i>update_agent</i>	No	Update an existing agent's configuration. Only fields explicitly passed are changed.
<i>delete_agent</i>	No	Delete an agent by name. Returns a single row with an <i>error</i> column (<i>NULL</i> on success).
<i>sleep</i>	Yes	Pause reasoning for a number of seconds, then automatically resume. Always available to every agent, regardless of its configured tool list, unless the model's own tool-count cap is already filled by the agent's other configured tools.

Pipelines

See [AI pipelines](#) .

Tool	Read-only	Description
<i>search_pipelines</i>	Yes	List all registered data pipelines.
<i>get_pipeline</i>	Yes	Get a registered data pipeline's configuration by name.
<i>delete_pipeline</i>	No	Delete a registered data pipeline by name.
<i>run_pipeline</i>	No	Manually run a pipeline once, processing any pending source rows.
<i>get_pipeline_metrics</i>	Yes	Get a pipeline's processing status and row counts.
<i>get_error_logs</i>	Yes	List a pipeline's record-level processing errors, optionally filtered.
<i>clear_error_logs</i>	No	Delete specific pipeline error log entries by id.
<i>requeue_pipeline_errors</i>	No	Mark specific pipeline errors' source rows dirty so the next run retries them.
<i>get_error_log_summary</i>	Yes	Get error counts for a pipeline, grouped by step, operation, and category.
<i>get_all_error_summaries</i>	Yes	Get error counts across every pipeline, grouped by step, operation, and category.

!!!note Pipeline authoring (`aidb.create_pipeline` / `aidb.update_pipeline`) has no native tool. Those func-

tions take up to 30 parameters across step/step-options pairs — impractical for a single generic tool call — so multi-step pipeline authoring isn't currently something an agent can do on its own.

9.1.2 Custom SQL tools

A SQL tool is a stored, parameterized query that an agent can invoke by name. Use it to give an agent access to your own tables and business logic, without writing any Rust or exposing a native AIDB function:

```
SELECT aidb.create_sql_tool(
  name          => orders_by_customer,
  description    => Look up recent orders for a customer by id.,
  sql_statement  => SELECT id, status, total FROM orders WHERE customer_id =_
  ↳ ${customer_id} ORDER BY created_at DESC LIMIT ${limit},
  params        => aidb.tool_params(
    aidb.tool_param(customer_id, INT, The customers id.),
    aidb.tool_param(limit, INT, Maximum number of orders to return.)
  ),
  read_only      => true,  -- optional; enforced, not just descriptive (see_
  ↳ below). Default: true
  return_type_hint => NULL  -- optional; documents the returned row shape, same_
  ↳ aidb.tool_params() shape as `params`. Default: NULL
);
```

Parameter	Type	Default	Description
<i>name</i>	TEXT	Required	Unique name for the tool.
<i>description</i>	TEXT	Required	Shown to the model as part of the tool's definition.
<i>sql_statement</i>	TEXT	Required	The query to run, with <i>\${name}</i> placeholders for each declared parameter.
<i>params</i>	JSONB	Required	The tool's parameters, built with <i>aidb.tool_params()</i> .
<i>read_only</i>	BOOLEAN	<i>true</i>	If <i>true</i> , reject the statement at registration time unless it's provably read-only.
<i>return_type_hint</i>	JSONB	<i>NULL</i>	Documents the query's return shape, in the same <i>aidb.tool_params()</i> shape. Not currently surfaced to the model.

aidb.tool_param(name, type, description) builds one parameter; *aidb.tool_params(...)* combines several; *aidb.tool_params()* with no arguments declares a tool that takes no parameters.

Register the tool on an agent the same way as any other:

```
SELECT aidb.create_agent(sales_helper, Answer questions about customer orders., my_
  ↳ gpt,
  tools => ARRAY[orders_by_customer]);
```

Naming

A tool's name must be unique across every tool type — it can't collide with a built-in *Native tools* or a tool already imported from an *MCP tools*. `aidb.create_sql_tool()` rejects the call up front if it does.

Parameter placeholders

Reference each declared parameter in `sql_statement` as `${name}` — AIDB substitutes each with the value the model supplied for that call, bound as a real query parameter (not string-interpolated). A parameter used more than once in the statement only needs to be declared once. `params` is required; pass `aidb.tool_params()` with no arguments for a tool that takes no parameters.

Read-only tools

`read_only` defaults to `true`. Pass `read_only => false` for a tool that needs to write — and that's enforced, not just descriptive, in both directions: left at the default `true`, registration fails immediately if `sql_statement` matches a recognized write pattern (`INSERT`, `UPDATE`, `DELETE`, and so on), rather than letting a mislabeled tool through. A tool left at the default `read_only => true` is also usable by an agent running in *Read-only mode*; a tool explicitly marked `read_only => false` is excluded from a read-only run, the same as an unclassified native or MCP tool.

`sql_statement` must be a single statement, and can't be a utility statement like `EXPLAIN` or `COPY`. Trailing whitespace and a trailing `;` are stripped automatically when you register it.

!!!note `create_sql_tool` also accepts a `return_type_hint` argument, in the same `aidb.tool_params()` shape as `params`, describing the shape of the rows the query returns. It's stored alongside the tool but isn't currently surfaced to the model or to `aidb.tools`. For now, its only use is as documentation for anyone inspecting `aidb.sql_tool_registry` directly.

Removing a tool

```
SELECT aidb.delete_tool(orders_by_customer);
```

`aidb.delete_tool()` also removes an *MCP tools* if the name you pass belongs to one instead of a SQL tool, but built-in native tools can't be deleted.

9.1.3 MCP tools

MCP (Model Context Protocol) is a standard way for a server to advertise a set of callable tools. `aidb.import_mcp_tools()` registers an external MCP server and imports the tools it advertises, so your agents can call them alongside AIDB's own native and SQL tools:

```
SELECT aidb.import_mcp_tools(
  name      => weather,
  url       => https://weather.example.com/mcp,
  transport => streamable_http, -- optional; streamable_http or sse (see below)...
  ↳Default: streamable_http
  headers   => NULL,           -- optional; request headers to send, for...
  ↳example, an auth token. Mutually exclusive with headers_env. Default: NULL
  tool_filter => NULL,         -- optional; import only these tool names...
  ↳Default: NULL (import everything the server advertises)
  headers_env => NULL          -- optional; env var holding the headers JSON...
  ↳instead of storing it. Mutually exclusive with headers. Default: NULL
);
```

```
SELECT aidb.create_agent(trip_planner, Help plan trips, checking weather when useful.,
```

(continues on next page)

(continued from previous page)

```
→ my_gpt,
   tools => ARRAY[get_forecast]; -- a tool the weather server advertised
```

Parameter	Type	Default	Description
<i>name</i>	TEXT	Required	Unique name for the server registration.
<i>url</i>	TEXT	Required	The server's MCP endpoint URL.
<i>transport</i>	TEXT	<i>'streamable_http'</i>	<i>'streamable_http'</i> or <i>'sse'</i> — only <i>'streamable_http'</i> is fully supported today.
<i>headers</i>	JSONB	<i>NULL</i>	Request headers to send, for example, <i>{ "Authorization": "Bearer ..." }</i> . Mutually exclusive with <i>headers_env</i> .
<i>tool_filter</i>	TEXT	<i>NULL</i>	Import only these tool names. Omit to import everything the server advertises.
<i>headers_env</i>	TEXT	<i>NULL</i>	Name of an environment variable holding the headers JSON, read fresh on each use instead of storing it. Mutually exclusive with <i>headers</i> .

`import_mcp_tools` validates the server by calling its `tools/list` endpoint before registering anything — if it's unreachable or advertises nothing matching `tool_filter`, nothing is stored. The tools it returns are cached immediately, so they show up in `aidb.tools` right away rather than after a separate refresh.

Only the `streamable_http` transport (the default) actually works end-to-end today; `sse` is accepted as a value but isn't fully implemented yet.

Reaching an MCP server is an outbound network call — subject to the same `aidb.egress_allowlist` restriction as model provider calls and HuggingFace downloads.

Authentication

Pass `headers` for a server that needs specific request headers (most commonly an `Authorization` bearer token):

```
SELECT aidb.import_mcp_tools(
  name    => internal_crm,
  url     => https://crm.internal.example.com/mcp,
  headers => {"Authorization": "Bearer sk-..."::JSONB
);
```

`headers` is stored in `aidb.mcp_registry`, but `aidb_users` has no `SELECT` access to that column — only AIDB's own internal calls (to validate, refresh, or invoke the server's tools) can read it back.

!!!note Prefer not to store the secret in the table at all? Pass `headers_env` instead: name an environment variable (read fresh from the Postgres process's own environment on every call) holding a JSON object of headers, in the same shape `headers` would take. `headers` and `headers_env` are mutually exclusive. The variable's name must start with the prefix configured by `aidb.env_var_allowed_prefix` (`AIDB_` by default) — the same mechanism and GUC used by `credentials_env` for model credentials.

Filtering which tools are imported

By default, every tool the server advertises is imported. Pass `tool_filter` to import only specific ones by name:

```
SELECT aidb.import_mcp_tools(
  name      => weather,
  url       => https://weather.example.com/mcp,
  tool_filter => ARRAY[get_forecast, get_alerts]
);
```

Keeping the tool list current

AIDB doesn't re-fetch a server's tool list on every query — `aidb.tools` only shows entries cached within the last 60 minutes, so a `SELECT` never has a network side effect. Call `aidb.refresh_mcp_tools()` to re-fetch and update the cache on demand (for example, after the server adds a new tool):

```
SELECT aidb.refresh_mcp_tools(weather);
```

It returns the number of tools now cached (after `tool_filter` applies). Changing a server's registration — its `url`, `transport`, `headers`, or `tool_filter`, via re-running `import_mcp_tools` -equivalent settings — also invalidates its cache automatically.

Removing a server

```
SELECT aidb.delete_tool(weather);
```

Deleting an MCP server removes every tool it advertised along with it — there's no way to remove just one of its tools individually; use `tool_filter` to curate which of its tools are exposed instead.

Exposing AIDB's own tools over MCP

`aidb.get_mcp_tools()` returns every tool currently in `aidb.tools` (native, SQL, and imported MCP tools alike), reshaped as MCP `tools/list` line items — useful if some other system needs AIDB's tool catalog in MCP's own descriptor format:

```
SELECT * FROM aidb.get_mcp_tools();
```

This only converts AIDB's tool catalog into MCP's descriptor shape — AIDB doesn't run an MCP server endpoint of its own that an external MCP client could connect to and actually invoke these tools over the network.

Pipelines can ingest, prepare and vectorize data for semantic retrieval and RAG applications. They integrate *pgvector iterative index scans* search capabilities and can keep the knowledge base synchronized with your source data automatically.

10.1 Example: a knowledge base for RAG

This walkthrough builds a knowledge base over internal support articles and shows each retrieval stage independently. For the complete version, see *End-to-end example: Knowledge base for RAG*, which adds model registration, a reranking demo where the cross-encoder corrects a wrong vector-search result, and a single statement that retrieves, reranks, and generates an answer with verified output.

10.1.1 Set up the pipeline

The source is a `support_articles` table holding 16 short articles about database operations — take the `CREATE TABLE` and `INSERT` statements from the full example’s *source data step*. The pipeline embeds each article body with `bge-small-en-v1.5-f16`, one of the built-in llama.cpp embedding models — pre-registered, runs locally, and its model file downloads automatically on first use:

```
SELECT aidb.create_pipeline(  
  name           => support_kb,  
  source         => support_articles,  
  source_key_column => id,  
  source_data_column => body,  
  step_1         => KnowledgeBase,  
  step_1_options  => aidb.knowledge_base_config(  
    model      => bge-small-en-v1.5-f16,  
    data_format => Text  
  ),  
  auto_processing => Disabled  
);
```

10.1.2 Run the pipeline

With `auto_processing => 'Disabled'` , nothing runs until you say so. Process all existing rows once:

```
SELECT aidb.run_pipeline(support_kb);
```

10.1.3 Retrieve: semantic search

Every article body is now embedded and indexed. Paste raw error text into a semantic search — the top match is the resolution, found by meaning rather than keywords:

```
```sql SELECT key, value, distance::numeric(5,3) AS distance FROM aidb.
retrieve_text('support_kb', 'application logs show: FATAL: sorry,
too many clients already', 3); __OUTPUT__ key | value | distance
-----+-----
AI pipelines`
```

<i>Pipeline steps</i>	Available step types: ChunkText, ParseHtml, ParsePdf, PerformOcr, SummarizeText, KnowledgeBase.
<i>Orchestration</i>	Auto-processing modes, background workers, observability, and error handling.
<i>AI pipelines</i>	Full API reference for pipeline types, views, CRUD functions, and config helpers.
<i>Examples</i>	End-to-end worked examples.

Creating pipelines

A pipeline defines how data moves from a source, through one or more transformation steps, to an AI-ready destination. Creating a pipeline is the first step in making your data searchable — without one, there is no process to generate embeddings or keep your knowledge base in sync with your source data.

Use `aidb.create_pipeline()` to define a pipeline. Its parameters fall into four groups:

Parameter group	Parameters	More information
Source	<i>source, source_key_column, source_data_column</i>	This page
Steps	<i>step_1 ... step_10, step_N_options</i>	<a href="#">Pipeline steps</a>
Orchestration	<i>auto_processing, background_sync_interval, batch_size</i>	<a href="#">Orchestration</a>
Destination	<i>destination</i>	<i>destination_key_column</i> and <i>destination_data_column</i> are not inputs — they're reported back in the function's return value, derived automatically from the destination type and steps

Table sources

To read from a Postgres table, set the `source` parameter to the table name, `source_key_column` to the unique key column, and `source_data_column` to the column containing the data to process:

```
SELECT aidb.create_pipeline(
 name => my_pipeline,
```

(continues on next page)

(continued from previous page)

```

source => my_table,
source_key_column => id,
source_data_column => content,
step_1 => KnowledgeBase,
step_1_options => aidb.knowledge_base_config(my_model, Text)
);

```

## Volume sources (PGFS)

To process files stored in external cloud storage (S3, GCS, or Azure), pipelines use Postgres File System (PGFS). PGFS mounts external object storage as a volume that the pipeline can scan for files. For more information on volumes, see [Configuring external storage](#).

### Step 1: Create a storage location

Define the external storage connection using `pgfs.create_storage_location`:

```

SELECT pgfs.create_storage_location(
 name => my_s3_location,
 url => s3://my-bucket/my-folder,
 options => {"region": "us-east-1"}
);

```

### Step 2: Create a volume

Create an [Configuring external storage](#) that references the storage location. The pipeline will scan this volume for new or changed files:

```

SELECT aidb.create_volume(
 name => my_volume,
 server_name => my_s3_location,
 path => /,
 data_format => Text
);

```

### Step 3: Reference the volume as the pipeline source

Set the `source` parameter to the volume name:

```

SELECT aidb.create_pipeline(
 name => my_pdf_pipeline,
 source => my_volume,
 step_1 => ParsePdf,
 step_2 => KnowledgeBase,
 step_2_options => aidb.knowledge_base_config(my_model, Text)
);

```

For full PGFS reference, see PGFS documentation.

For a complete walkthrough combining a table source, pipeline steps, and retrieval, see [Examples](#).

## Pipeline steps

Each step in an AI pipeline is defined by its operation type and an optional configuration object. You pass the operation name as a string to the `step_N` parameter of `aidb.create_pipeline()`, and the configuration as JSONB to the corresponding `step_N_options` parameter.

```
SELECT aidb.create_pipeline(
 name => my_pipeline,
 source => my_source_table,
 source_key_column => id,
 source_data_column => content,
 step_1 => ChunkText,
 step_1_options => aidb.chunk_text_config(desired_length => 120, max_length =>
↪150),
 step_2 => KnowledgeBase,
 step_2_options => aidb.knowledge_base_config(model => bert, data_format =>
↪Text)
);
```

Use the helper functions described below to build the `step_N_options` value for each operation type. All helper functions return a JSONB configuration object.

## Intermediate storage

Any step can persist its own output to a dedicated table, in addition to passing it on to the next step (or, for the last step, to the pipeline's final destination). This is useful for debugging a multi-step pipeline, or for reusing an intermediate result (such as chunked text) outside the pipeline. It's off by default.

Set it with an `intermediate_destination` key in that step's `step_N_options`. No step config helper (`aidb.chunk_text_config()`, `aidb.pdf_parse_config()`, and so on) has a parameter for it yet, so add it as plain JSON alongside the helper's own output, merged with the `||` operator:

```
step_1_options => aidb.chunk_text_config(desired_length => 100) || jsonb_build_object(
 intermediate_destination, jsonb_build_object(enabled, true)
)
```

Or build the whole options object directly with `jsonb_build_object()`:

```
step_1_options => jsonb_build_object(
 desired_length, 100,
 intermediate_destination, jsonb_build_object(enabled, true)
)
```

Key	Type	Default	Description
<i>enabled</i>	BOOLEAN	Required	Turn on intermediate storage for this step.
<i>destination</i>	TEXT	NULL	Table to store the step's output in. Auto-generated as <code>pipeline_&lt;name&gt;_step_&lt;N&gt;</code> when omitted.

The simplest form just turns it on and lets AIDB name the table:

```
{ "enabled": true }
```

Or name the destination table explicitly:

```
{ "enabled": true, "destination": "my_custom_chunk_storage" }
```

The destination table has the same shape as any other pipeline destination table: `source_id`, `part_ids`, and a value column holding the step's output.

Setting `intermediate_destination` on a pipeline's **last** step still creates the table and writes to it, but `aidb.retrieve_text()`'s `intermediate_steps` column never surfaces it — that lookup only considers steps before the last one, since there's no “next step” for the last one's output to be intermediate before. Query the destination table directly instead if you need to read it back. This matters most for `KnowledgeBase`, which must always be the last step in a pipeline — so there's rarely a reason to enable intermediate storage on it.

!!!note `intermediate_destination` must be an object with an `enabled` key (and optionally `destination`). Any other JSON type raises an error at pipeline creation:

```
ERROR: Invalid options for step 1: intermediate_destination must be an object with
↳ keys enabled and optionally destination; got a string
```

## ChunkText

Divides text into smaller segments to fit within LLM context windows.

**Helper function:** `aidb.chunk_text_config()`

Parameter	Type	Default	Description
<i>desired_length</i>	<i>INTEGER</i>	Required	Target chunk size. The unit depends on <i>strategy</i> .
<i>max_length</i>	<i>INTEGER</i>	<i>NULL</i>	Maximum chunk size. If omitted, <i>desired_length</i> is a strict upper limit.
<i>overlap_length</i>	<i>INTEGER</i>	<i>NULL</i>	Amount of content to overlap between consecutive chunks. Defaults to 0 (no overlap).
<i>strategy</i>	<i>TEXT</i>	<i>NULL</i>	'chars' (default) for character-based or 'words' for word-based chunking.

**Basic chunking example:**

```
SELECT aidb.create_pipeline(
 name => chunk_pipeline,
 source => source_table,
 source_key_column => id,
 source_data_column => content,
 step_1 => ChunkText,
 step_1_options => aidb.chunk_text_config(
 desired_length => 100,
 max_length => 150,
 overlap_length => 20,
 strategy => words
)
);
```

## Result

`ChunkText` transforms the shape of the data by introducing a `part_id` column. Each source row may produce multiple output rows, one per chunk.

## ParseHtml

Extracts readable text from HTML strings, stripping tags while preserving logical structure.

**Helper function:** `aidb.html_parse_config()`

Parameter	Type	Default	Description
<i>method</i>	<i>TEXT</i>	<i>NULL</i>	'StructuredPlaintext' (default) for plain text extraction, or 'Structured-Markdown' to retain hierarchy.

```
SELECT aidb.create_pipeline(
 name => html_pipeline,
 source => web_data_table,
 source_key_column => id,
 source_data_column => html_content,
 step_1 => ParseHtml,
 step_1_options => aidb.html_parse_config(method => StructuredMarkdown),
 step_2 => ChunkText,
 step_2_options => aidb.chunk_text_config(desired_length => 120)
);
```

## Result

`ParseHtml` doesn't unnest results — each source row produces exactly one output row containing the extracted text, with the original row correspondence preserved. Pair it with `ChunkText` (as in the example above) to split the extracted text into smaller segments.

## ParsePdf

Extracts text from binary PDF data, with options to handle non-compliant or complex files.

**Helper function:** `aidb.pdf_parse_config()`

Parameter	Type	Default	Description
<i>method</i>	<i>TEXT</i>	<i>NULL</i>	'Structured' (default) — uses the PDF specification to identify text blocks.
<i>allow_partial_parsing</i>	<i>BOOLEAN</i>	<i>NULL</i>	If <i>true</i> (default), continues parsing when errors are encountered on individual pages.

The resulting `part_id` column maps to the page index from which each text block was extracted.

```
SELECT aidb.create_pipeline(
 name => pdf_pipeline,
 source => pdf_files_table,
```

(continues on next page)

(continued from previous page)

```

source_key_column => id,
source_data_column => pdf_data,
step_1 => ParsePdf,
step_1_options => aidb.pdf_parse_config(
 method => Structured,
 allow_partial_parsing => true
),
step_2 => KnowledgeBase,
step_2_options => aidb.knowledge_base_config(model => bert, data_format =>
↳ Text)
);

```

## Result

ParsePdf unnests results — a multi-page PDF produces one row per page, each with a `part_id` corresponding to the page index.

## PdfToImage

Converts each page of a PDF document into an image. Use this step when source PDFs are scanned documents, image-heavy layouts, or any PDF where the text layer is absent or unreliable. The rendered page images can then pass directly to a `PerformOcr` step for text extraction.

For digitally produced PDFs that contain a native text layer, use `ParsePdf` instead — it's faster and doesn't require an OCR model.

**Options format:** Pass options as a JSON literal.

Parameter	Type	Default	Description
<i>dpi</i>	<i>INTEGER</i>	<i>300</i>	Resolution in dots per inch used to render each page. Higher values produce sharper images at the cost of larger file size.
<i>first_page</i>	<i>INTEGER</i>	<i>NULL</i>	First page to render (1-based). Renders from the beginning when omitted.
<i>last_page</i>	<i>INTEGER</i>	<i>NULL</i>	Last page to render, inclusive (1-based). Renders to the end when omitted.
<i>max_pages</i>	<i>INTEGER</i>	<i>NULL</i>	Maximum number of pages to render. Acts as a safety cap regardless of <i>first_page</i> and <i>last_page</i> .
<i>render_annotations</i>	<i>BOOLEAN</i>	<i>true</i>	Whether to include PDF annotations (such as comments and form fields) in the rendered output.
<i>format</i>	<i>JSONB</i>	<i>{"type": "png"}</i>	Output image format. Accepted values: <i>{"type": "png"}</i> , <i>{"type": "jpeg"}</i> . Use <i>{"type": "png"}</i> when passing output to <i>PerformOcr</i> .

Output images are in PNG or JPEG format depending on the `format` option. Use `"png"` (the default) when passing output to the `PerformOcr` step — NVIDIA NIM PaddleOCR requires PNG or JPEG input.

Before using this step in a pipeline that includes `PerformOcr`, register an OCR-capable model:

```
SELECT aidb.create_model(
 my_paddle_ocr_model,
 nim_paddle_ocr,
 credentials => {"api_key": "<NVIDIA_NIM_API_KEY>"}::JSONB -- or credentials_env_
 ↳to read this from an environment variable instead
);
```

Then create the pipeline:

```
SELECT aidb.create_pipeline(
 name => pdf_ocr_pipeline,
 source => pdf_files_table,
 source_key_column => id,
 source_data_column => pdf_data,
 step_1 => PdfToImage,
 step_1_options => {"dpi": 150}::json,
 step_2 => PerformOcr,
 step_2_options => aidb.ocr_config(model => my_paddle_ocr_model),
 step_3 => KnowledgeBase,
 step_3_options => aidb.knowledge_base_config(model => bert, data_format =>
 ↳Text)
);
```

## Result

`PdfToImage` unnests results — each page of the source PDF produces one output row containing the rendered page as a PNG or JPEG image, depending on the `format` option. The `part_id` column maps to the page index (zero-based). This output shape is directly compatible with the `PerformOcr` step, which expects image bytes as its input.

## PerformOcr

Extracts text from images using an OCR-capable AI model, such as NVIDIA NIM PaddleOCR.

**Helper function:** `aidb.ocr_config()`

Parameter	Type	Default	Description
<i>model</i>	<i>TEXT</i>	Required	Name of the registered OCR model to use.

Before using this step, register an OCR-capable model:

```
SELECT aidb.create_model(
 my_paddle_ocr_model,
 nim_paddle_ocr,
 credentials => {"api_key": "<NVIDIA_NIM_API_KEY>"}::JSONB -- or credentials_env_
 ↳to read this from an environment variable instead
);
```

Then reference it in your pipeline:

```

SELECT aidb.create_pipeline(
 name => ocr_pipeline,
 source => images_table,
 source_key_column => id,
 source_data_column => image_data,
 step_1 => PerformOcr,
 step_1_options => aidb.ocr_config(model => my_paddle_ocr_model),
 step_2 => KnowledgeBase,
 step_2_options => aidb.knowledge_base_config(model => bert, data_format =>_
↪Text)
);

```

## Result

`PerformOcr` unnests results — a single image may produce multiple rows, one per detected text block. The NVIDIA NIM provider currently supports only `png` and `jpeg` formats.

## SummarizeText

Generates concise summaries of long text passages using an AI language model.

**Helper function:** `aidb.summarize_text_config()`

Parameter	Type	Default	Description
<i>model</i>	<i>TEXT</i>	Required	Name of the registered model to use for summarization.
<i>chunk_config</i>	<i>JSONB</i>	<i>NULL</i>	Optional chunking configuration (from <code>aidb.chunk_text_config()</code> ) applied before summarization.
<i>prompt</i>	<i>TEXT</i>	<i>NULL</i>	Custom prompt to guide the summarization. Uses a standard prompt if omitted.
<i>strategy</i>	<i>TEXT</i>	<i>NULL</i>	'append' (default) concatenates per-chunk summaries, or 'reduce' iteratively compresses them.
<i>reduction_factor</i>	<i>INTEGER</i>	<i>NULL</i>	Used with 'reduce' strategy. Controls how aggressively text is reduced per iteration (default is 3).
<i>inference_config</i>	<i>JSONB</i>	<i>NULL</i>	Optional runtime inference settings (from <code>aidb.inference_config()</code> ).

```

SELECT aidb.create_pipeline(
 name => summary_pipeline,
 source => articles_table,
 source_key_column => id,
 source_data_column => body,
 step_1 => SummarizeText,
 step_1_options => aidb.summarize_text_config(
 model => my_t5_model,
 chunk_config => aidb.chunk_text_config(100, 120, 10, words),
 prompt => Summarize the key points concisely,

```

(continues on next page)

(continued from previous page)

```

 strategy => reduce,
 reduction_factor => 3
),
 step_2 => KnowledgeBase,
 step_2_options => aidb.knowledge_base_config(model => bert, data_format =>
↪ Text)
);

```

## KnowledgeBase

Converts processed text or image data into vector embeddings and stores them in a searchable knowledge base. This step must always be the **last step** in a pipeline, as its output is a **VECTOR** type that can't be used as input by any subsequent step. For querying the knowledge base with semantic or hybrid search, see [Knowledge bases reference](#).

**Helper function:** `aidb.knowledge_base_config()`

Parameter	Type	Default	Description
<i>model</i>	<i>TEXT</i>	Required	Name of the embedding model.
<i>data_format</i>	<i>aidb.Pipeline-DataFormat</i>	Required	'Text' or 'Image'.
<i>distance_operator</i>	<i>aidb.DistanceOperator</i>	<i>NULL</i>	Similarity metric: <i>L2</i> (default), <i>InnerProduct</i> , <i>Cosine</i> , <i>L1</i> , <i>Hamming</i> , or <i>Jaccard</i> .
<i>vector_index</i>	<i>JSONB</i>	<i>NULL</i>	Vector index config, built with a vector index helper such as <i>aidb.vector_index_hnsw_config()</i> .

```

SELECT aidb.create_pipeline(
 name => kb_pipeline,
 source => source_table,
 source_key_column => id,
 source_data_column => content,
 step_1 => KnowledgeBase,
 step_1_options => aidb.knowledge_base_config(
 model => bert,
 data_format => Text,
 distance_operator => Cosine,
 vector_index => aidb.vector_index_hnsw_config(m => 16, ef_construction =>
↪ 64)
)
);

```

Use `aidb.knowledge_base_config_from_kb(data_format)` to attach a pipeline to an existing knowledge base. This lets multiple pipelines share one vector table, and inherits the model, distance operator, and vector index from the knowledge base. See [Multi-pipeline knowledge bases](#).

## Destination table

The KnowledgeBase step creates a destination table named by the `destination` parameter. If `destination` isn't set, the table defaults to `pipeline_<pipeline_name>`. When multiple pipelines share a knowledge base, they all write to the same table.

Column	Type	Description
<i>id</i>	BIGSERIAL	Primary key.
<i>pipeline_id</i>	INT	Reference to the originating pipeline.
<i>source_id</i>	TEXT	ID of the original source record.
<i>part_ids</i>	BIGINT[]	Tracks segments if the data was chunked or parsed.
<i>value</i>	VECTOR	The pgvector embedding.

For knowledge base views, retrieval functions, and lifecycle behavior, see [Knowledge bases reference](#) and [Knowledge bases reference](#).

To see pipeline steps used together in a complete end-to-end workflow, see [Examples](#).

## Orchestration

Orchestration manages the execution of the pipeline, determining when and how data is processed. Here are the core components of orchestration:

- **\*\* [Enable auto-processing](#) \*\***: Pipelines can run in the background or stay in sync with your data. You can configure batch sizes and triggers, and choose between synchronous or asynchronous processing modes.
- **Change detection**: It identifies new, updated, or deleted records so only dirty data is processed.
- **Sync vs async**: You choose between live (immediate, transactional updates) or background (periodic, non-blocking updates).
- **\*\* [Background workers](#) \*\***: Background workers are dedicated Postgres processes that execute AI tasks — such as OCR, document parsing, and embedding API calls — asynchronously, without affecting the performance of your main database.
- **Resource isolation**: They run independently of user sessions, ensuring your app stays fast even during massive data ingestions.
- **Batching & retries**: They group data into efficient batches to save on API costs and automatically retry if an external service (like OpenAI) is temporarily down.
- **\*\* [Monitoring and observability](#) \*\***: It provides detailed status of your automated workflows. Since these processes are running in the background, observability allows you to track health, speed, and accuracy.
- **Status tracking**: Use views like `aidb.pipeline_metrics` to see how many rows are in the backlog and the total count of records in the destination.
- **\*\* [Pipeline error log](#) \*\***: It captures exactly why a specific file failed to parse, for example, a corrupted PDF or a timeout, allowing for quick troubleshooting. Inspect, clear, and requeue from a SQL surface.

To see orchestration applied in a complete pipeline workflow, see [Examples](#).

## Auto-processing

Auto-processing is a set of capabilities designed to keep source data (in tables or volumes) and AI pipeline outputs (like embeddings in a knowledge base) in sync automatically.

## Key features

- **Full sync:** Automatically handles inserts, updates, and deletes so that knowledge bases remain accurate without manual intervention.
- **Change detection:** Processes only new or modified records, avoiding unnecessary reprocessing of unchanged data.
- **Quick turnaround:** Results become available immediately after a batch finishes processing. There is no need for a full source scan before starting.
- **Batch processing:** Groups records into batches to be processed concurrently, which is optimal for GPU-based inference tasks.

## Auto-processing modes

There are three main modes for managing how data moves through the pipeline:

Mode	Description	Pros	Cons
<i>Live</i>	Uses Postgres triggers to process changes immediately within the same transaction that modifies the data.	Transactional guarantee, zero lag.	Can block/delay the original data modification transaction.
<i>Background</i>	Uses a Postgres background worker to process data asynchronously at a configurable interval ( <i>background_sync_interval</i> ).	Does not block user transactions and ideal for huge data sets.	Results are delayed by the sync interval.
<i>Disabled</i>	No automatic sync. Data must be processed manually using <i>aidb.run_pipeline()</i> .	Full control over when resources are used.	High manual overhead, requires full reprocessing for table sources.

## Configuration

The auto-processing mode is set via the `auto_processing` parameter when creating or updating a pipeline.

### Setting the mode at pipeline creation:

```
SELECT aidb.create_pipeline(
 name => my_pipeline,
 source => my_source_table,
 ...
 auto_processing => Background,
 background_sync_interval => 5 minutes
);
```

### Changing the mode for an existing pipeline:

```
SELECT aidb.update_pipeline(
 my_pipeline,
 auto_processing => Live
);
```

For more details on these functions and their full parameter lists, see [Pipelines API reference](#).

Parameter	Type	Description
<code>auto_processing</code>	<code>aidb.PipelineAutoProcessingMode</code>	Auto-processing mode: <i>Live</i> , <i>Background</i> , or <i>Disabled</i> .
<code>background_sync_interval</code>	<code>interval</code>	Interval between background executions. Must be between 1 second and 2 days. Only applies when <code>auto_processing</code> is <i>Background</i> .

## Background workers

Background worker is the execution engine for asynchronous AI pipelines in EDB Postgres AI. It allows for high-volume data processing without blocking standard database transactions.

A background worker is activated automatically when a pipeline is created or updated with `auto_processing => 'Background'`. Before enabling Background mode, confirm that the required Postgres server settings are in place.

## Postgres prerequisites

Two `postgresql.conf` settings must be in place before background pipelines can run.

### shared\_preload\_libraries

`shared_preload_libraries = 'aidb'` must be set in `postgresql.conf` before background workers can start. This is configured during AIDB installation — see [Configuring AIDB](#). Confirm it is in place before enabling Background mode on any pipeline.

### max\_worker\_processes

Each pipeline in Background mode requires one dedicated Postgres background worker process. The `max_worker_processes` setting controls how many background worker processes are allowed across the entire cluster.

The default value is 8. If you plan to run multiple background pipelines, increase this value:

```
max_worker_processes = 20
```

Restart Postgres after changing this setting. To check the current value:

```
SHOW max_worker_processes;
```

If `max_worker_processes` is exhausted, newly created background pipelines will not start a worker and will silently skip processing until capacity is freed.

## Core functionality

- **Asynchronous execution:** When a pipeline is set to Background mode, processing occurs independently of the user session. This method ensures that queries or data modifications on the source table are not delayed by embedding generation or OCR tasks.
- **Batch processing:** Background workers group records into configurable batch sizes. This optimizes throughput, especially when interacting with GPU-based models or remote AI service APIs.
- **Parallel operations:** Within each batch, the worker runs pipeline steps (data retrieval, embedding computation, and storage) as parallel operations to maximize performance.
- **Continuous polling:** The worker continuously monitors the source for changes based on a defined `background_sync_interval`.

## Change detection

The background worker handles different source types using specific detection logic:

- **Table sources:** Lightweight triggers capture change events like inserts, updates, and deletes and place them in a backlog. The background worker then processes this backlog at the next interval.
- **Volume sources:** The background worker performs a scan of the external storage. It compares the `last_modified` timestamps of files against a state table to identify new or changed documents.

## Configuration & constraints

### Required

- **Postgres prerequisites:** Background workers require `shared_preload_libraries = 'aidb'` and a sufficient `max_worker_processes` value in `postgresql.conf`. See [Postgres prerequisites](#) on this page.

### Optional tuning

`background_sync_interval` controls how often the worker polls for new or changed data. This is configured per pipeline via `aidb.create_pipeline()` or `aidb.update_pipeline()`, not in `postgresql.conf`.

```
SELECT aidb.create_pipeline(
 name => my_pipeline,
 source => my_table,
 auto_processing => Background,
 background_sync_interval => 30 seconds,
 ...
);
```

The default interval is suitable for most table sources. For volume sources backed by cloud object stores like AWS S3, each scan incurs a list operation which may have cost or rate-limit implications. In those cases, consider a longer interval:

```
SELECT aidb.update_pipeline(
 my_s3_pipeline,
 background_sync_interval => 1 day
);
```

`background_sync_interval` accepts any Postgres interval value ('5 minutes', '1 hour', '1 day', etc.).

## Monitoring and observability

You can track the status and health of background workers using the `aidb.pipeline_metrics` view (also accessible as `aidb.pipem`). Key metrics include:

- **Table- unprocessed rows:** For table sources, the number of rows not yet processed.
- **Volume- scans completed:** For volume sources, the number of full scans completed.
- **Count (source records):** Total number of records in the source.
- **Count (destination records):** Total number of records in the destination.
- **Status:** Current pipeline status.

## Observability

See detailed status and progress output for all auto-processing modes.

Two views cover most on-demand observability needs — each computes its numbers at query time:

- *aidb.pipeline\_metrics* (alias *aidb.pipem*) — one row per pipeline with backlog, source/destination counts, status, and error counts. This is the primary view for tracking processing progress.
- *aidb.knowledge\_base\_metrics* (alias *aidb.kbm*) — one row per knowledge base with pipeline count, embedding count, and aggregated status across all attached pipelines.

## On-demand metrics

### Per-pipeline progress

```
SELECT * FROM aidb.pipem;
__OUTPUT__
 pipeline | auto processing | table: unprocessed rows | volume: scans |
 completed | count(source records) | count(destination records) | Status |
 count(record errors) | count(blocking errors) | last run completed
-----+-----+-----+-----+-----+
 feedback_na_pipe | Live | | 0 |
 | | | | 412 | UpToDate |
 | | | | 0 | 2026-07-21 10:14:58.001+00
 product_docs_pipe | Background | | 1842 | UpToDate |
 | | | | 6 | 2026-07-21 10:10:02.774+00
 | | | | 0 |
(2 rows)
```

The *Change detection* mechanism is different for volume and table sources, so the view exposes these metrics:

- *table: unprocessed rows* : How many unique rows are listed in the backlog of change events. Auto-processing captures no new change events when Disabled.
- *volume: scans completed* : How many full listings of the source have been completed.
- *count(source records)* : How many records exist in the source. Accurate for table sources at all times; updated after each full scan for volume sources.
- *count(destination records)* : How many rows the pipeline has written to its destination (embeddings, chunks, and so on).

## Knowledge base overview

```
SELECT * FROM aidb.kbm;
__OUTPUT__
 name | pipelines | embeddings | status
-----+-----+-----+-----
 public.product_catalog_kb | 1 | 1842 | UpToDate
 public.customer_feedback_kb | 2 | 915 | UpToDate
(2 rows)
```

The *status* column rolls up the worst pipeline status across all pipelines attached to that KB — drill into *aidb.pipeline\_metrics* to find which pipeline is responsible.

## Materialized metrics

`aidb.pipeline_metrics` and `aidb.knowledge_base_metrics` compute their numbers on demand, at query time. For external monitoring — Prometheus scraping through the EDB Postgres AI agent, for example — polling those views on an interval means recomputing every pipeline’s stats on every scrape, and requires the scraping role to reach each pipeline owner’s underlying tables.

`aidb.pipeline_metrics_collection` avoids both costs. A background worker snapshots the same statistics from every pipeline, querying as that pipeline’s owning role, and writes them into this table every 15 seconds. Any role granted access to the table — `pg_monitor` gets `SELECT` by default — can then read pre-computed metrics without recalculating anything or needing per-pipeline permissions.

```
SELECT * FROM aidb.pipeline_metrics_collection;
__OUTPUT__
 pipeline | auto processing | table: unprocessed rows | volume: scans_
→ completed | count(source records) | count(destination records) | count(record_
→ errors) | count(blocking errors) | Status | owner_role | collected_at
-----+-----+-----+-----+-----+-----+-----+-----+
→ -----+-----+-----+-----+-----+-----+-----+
→ -----+-----+-----+-----+-----+-----+-----+
feedback_na_pipe | Live | | 0 |
→ | 412 | | 412 | | 0 |
→ | 0 | UpToDate | app_owner | 2026-07-21 10:15:03.221+00
product_docs_pipe | Background | | |
→ 6 | | 1842 | | 1842 | | 0 |
→ | 0 | UpToDate | app_owner | 2026-07-21 10:15:03.402+00
(2 rows)
```

Columns match `aidb.pipeline_metrics`, plus:

- `owner_role`: the pipeline’s owning role, same as `aidb.pipelines.owner_role`.
- `collected_at`: when this row was last refreshed.

Rows for deleted pipelines are removed automatically on the next collection cycle.

## Enabling the collector

The collector is opt-in and disabled by default, so `aidb.pipeline_metrics_collection` stays empty until you turn it on. Enable it for the whole instance:

```
ALTER SYSTEM SET aidb.enable_background_metrics_collector = on;
```

Or scope it to one database, which overrides the instance-wide default for that database:

```
ALTER DATABASE my_db SET aidb.enable_background_metrics_collector = on;
```

!!!note Restart Postgres to apply the new configuration.

## Inspecting failed records

Per-record failure details (which step failed, the operation, the underlying error message, and retry state) live in the *Pipeline error log*.

## Pipeline error log

Each pipeline has an error log table. When a record fails at any step, the log captures the source record ID, which step failed, the operation type, and the error message. The rest of the batch keeps processing.

### How it works

- When a record fails, it's logged to `{source_schema}.pipeline_{pipeline_name}_errors` with the source ID, failing step number, operation type, and error message.
- A single bad record doesn't block the pipeline. Other records in the batch continue through all remaining steps.
- The error log table is created with the pipeline and dropped when the pipeline is deleted. Pipelines created before 7.2.0 get their table during the upgrade.
- Writes happen inside the failing step's transaction, so a successful step never leaves orphan log rows and a failure never loses its record.
- Each logged error also emits a Postgres `WARNING`. Toggle with `aidb.pipeline_error_warnings`; errors persist to the log regardless.
- The query and summary functions return empty results for any pipeline whose error log table is missing.
- Don't drop the error log table manually. As of 7.4.0, the source `DELETE` handlers and post-step cleanup raise an error if the table is missing.
- Errors get one of four `error_category` values (see [aidb.ErrorBlocking](#)):

Category	Scope	Blocks pipeline?	Description
<i>RecordTemporary</i>	Record	No	Transient failure for a single record. May succeed on retry.
<i>RecordPermanent</i>	Record	No	Permanent failure for a single record (for example, corrupt data).
<i>PipelineTemporary</i>	Pipeline	Yes	Transient failure that stopped the entire step (for example, a service outage). May resolve on retry.
<i>PipelinePermanent</i>	Pipeline	Yes	Permanent failure that stopped the entire step (for example, invalid model config).

Record-level errors don't block the pipeline; other records keep processing. Pipeline-level errors block the entire step. Temporary errors may resolve on retry; permanent errors point to bad data or misconfiguration. Specifically: `RecordTemporary` currently fires only for model rate-limit errors, `RecordPermanent` covers bad input data (for example, corrupt PDF, malformed HTML, invalid UTF-8), `PipelineTemporary` covers transient external faults (for example, object store outage, model service timeout), and `PipelinePermanent` is the general case for misconfiguration (for example, deleted model, invalid step config).

### Querying errors

Use `aidb.get_error_logs()` to inspect a pipeline's error log.

```
SELECT * FROM aidb.get_error_logs(my_pipeline);
```

\_\_OUTPUT\_\_

```
id | source_id | part_ids | pipeline_step | step_operation | error_message
```

(continues on next page)

(continued from previous page)

```

↪ | error_category | failed_at | retry_count |
↪last_retry_at
- - - - -
↪
↪
3 | doc_99 | {0,2} | 2 | ChunkText | text segment exceeded
↪maximum length | RecordPermanent | 2026-04-10 14:32:01.123+00 | 0 |
2 | doc_42 | {0} | 1 | ParsePdf | failed to extract text
↪from page 0 | RecordPermanent | 2026-04-10 14:31:58.456+00 | 0 |
1 | doc_7 | | 1 | ParsePdf | corrupt PDF header
↪ | RecordPermanent | 2026-04-10 14:31:55.789+00 | 0 |
(3 rows)

```

Filter by source record, step number, or error category:

```

SELECT * FROM aidb.get_error_logs(
 my_pipeline,
 p_source_id => doc_42
);

```

```

SELECT * FROM aidb.get_error_logs(
 my_pipeline,
 p_pipeline_step => 1::SMALLINT,
 p_error_category => RecordPermanent
);

```

Paginate with `p_limit` and `p_offset`:

```

SELECT * FROM aidb.get_error_logs(my_pipeline, p_limit => 20, p_offset => 40);

```

See the [Operations reference](#) for full parameter and return type details.

## Error summaries

Get error counts grouped by step, operation, and category for one pipeline:

```

SELECT * FROM aidb.get_error_log_summary(my_pipeline);

__OUTPUT__
pipeline_step | step_operation | error_category | error_count | latest_failed_at
- - - - -
↪
↪ 1 | ParsePdf | RecordPermanent | 2 | 2026-04-10
↪14:31:58.456+00
↪ 2 | ChunkText | RecordPermanent | 1 | 2026-04-10
↪14:32:01.123+00
(2 rows)

```

Or get the same rollup across all pipelines at once:

```

SELECT * FROM aidb.get_all_error_summaries();

```

(continues on next page)

(continued from previous page)

__OUTPUT__					
pipeline_name	pipeline_step	step_operation	error_category	error_count	
↪ latest_failed_at					
-----+-----+-----+-----+-----+-----					
↪ -----					
my_pipeline	1	ParsePdf	RecordPermanent	2	↪
↪ 2026-04-10 14:31:58.456+00					
my_pipeline	2	ChunkText	RecordPermanent	1	↪
↪ 2026-04-10 14:32:01.123+00					
pdf_ingester	1	ParsePdf	PipelinePermanent	1	↪
↪ 2026-04-10 15:00:12.000+00					
(3 rows)					

See the *Operations reference* and *aidb.get\_all\_error\_summaries* for full parameter and return type details.

### Requeuing failed records

New in 7.5.0. `aidb.requeue_pipeline_errors()` deletes the targeted record-level error rows and marks their source records dirty on the pipeline's state surface. The next normal run re-processes them; the call itself does no processing.

```
SELECT * FROM aidb.requeue_pipeline_errors(my_pipeline, ARRAY[1, 2, 99]);
```

__OUTPUT__	
error_id	action
-----+-----	
1	requeued
2	requeued
99	not_found
(3 rows)	

Results are ordered by ascending `error_id`. Each row reports one of three actions:

- `requeued`: record-level row deleted and source marked dirty. Table sources enqueue a state event; volume sources set a one-shot `retry_requested_at` marker on the state row.
- `skipped_pipeline_level`: the entry has a NULL `source_id` (see below) and is left in place.
- `not_found`: the id was absent from the error log, or the pipeline has no error log table (legacy pipelines or Empty sources like Semantic KB).

Background pipelines drain requeued rows automatically on the next worker tick. Disabled or Live pipelines need a follow-up `aidb.run_pipeline('my_pipeline')` to actually re-process.

Requeue is a deferred mark-dirty operation, not a synchronous retry. The call itself does no processing, so per-record success or failure only surfaces after the follow-up run. To confirm what happened after re-processing, re-query `aidb.get_error_logs`.

Concurrent requeues on overlapping ids serialize on per-row `FOR UPDATE` locks. Non-overlapping id sets proceed in parallel. Requeue doesn't block `aidb.run_pipeline` or the background worker.

## Pipeline-level errors

Pipeline-level errors (NULL `source_id`) record batch failures that aren't tied to a specific record. They can't be requeued per record, so `aidb.requeue_pipeline_errors` returns `action='skipped_pipeline_level'` and leaves the entry in the log:

```
SELECT * FROM aidb.requeue_pipeline_errors(my_pipeline, ARRAY[7]);

__OUTPUT__
error_id | action
-----+-----
 7 | skipped_pipeline_level
(1 row)
```

Fix the underlying issue (for example, model availability, config, deleted model) and re-run the pipeline. The NULL row clears automatically once the step next executes without any errors. To dismiss it without re-running, use `aidb.clear_error_logs`.

## Disabled-mode volume caveat

On a Disabled-mode pipeline with a volume source, `aidb.requeue_pipeline_errors` sets the `retry_requested_at` marker on the marked files but can't schedule a drain, so it emits a NOTICE pointing at `aidb.run_pipeline`. Manual `aidb.run_pipeline` re-reads the whole volume rather than only the marked files, which costs a full-volume scan on large volumes.

For large Disabled-mode volumes, switch the pipeline to Background mode, call `aidb.requeue_pipeline_errors`, let the worker drain the marked files, then switch the pipeline back to Disabled.

See the [Operations reference](#) for full parameter and return type details.

## Clearing errors

Delete specific error log entries by ID:

```
SELECT aidb.clear_error_logs(my_pipeline, ARRAY[1, 3]);

__OUTPUT__
clear_error_logs

 2
(1 row)
```

Returns the number of deleted rows. Clearing is ID-based; review entries with `get_error_logs()` before removing them.

See the [Operations reference](#) for full parameter and return type details.

## Automatic cleanup

Error log entries clear without `aidb.clear_error_logs` or `aidb.requeue_pipeline_errors` in three cases:

- Source record deleted. The live DELETE trigger, background worker DELETE processing, and volume stale-state cleanup all remove matching error log entries alongside their destination rows.
- Source envelope reaches the destination on a later run. The pipeline executor clears entries scoped by `(source_id, part_ids)` so part-granular successes don't wipe sibling errors.
- Pipeline-level rows (NULL `source_id`) for a step clear after that step next runs cleanly. Errors written by a concurrent failing executor during that run are preserved.

Pipeline deletion drops the entire error log table.

## Pipeline status integration

Error counts also appear in the `aidb.pipeline_metrics` view. The pipeline status reflects error severity:

- `PartialErrors`: some records failed; the rest processed normally.
- `BlockingErrors`: a pipeline-level error stopped the step entirely.

See *Monitoring and observability* for more on monitoring pipeline health.

## End-to-end example

A short walkthrough using a three-document source where one row has a corrupt PDF, one parses but produces an oversized chunk, and a third hits a model rate limit on the embedding step.

Run the pipeline. None of the three failures abort the batch. Then inspect the resulting errors:

```
SELECT aidb.run_pipeline(my_pipeline);

SELECT id, source_id, error_category
FROM aidb.get_error_logs(my_pipeline);
__OUTPUT__
id | source_id | error_category
--+-+-----+-----
 3 | doc_200 | RecordTemporary
 2 | doc_42 | RecordPermanent
 1 | doc_7 | RecordPermanent
(3 rows)
```

The pipeline now reports `PartialErrors` and surfaces the counts:

```
SELECT pipeline, "Status", "count(record errors)", "count(blocking errors)"
FROM aidb.pipeline_metrics WHERE pipeline = my_pipeline;
__OUTPUT__
pipeline | Status | count(record errors) | count(blocking errors)
--+-+-----+-----+-----
my_pipeline | PartialErrors | 3 | 0
(1 row)
```

After fixing the upstream rate-limit issue, requeue the rate-limited row. This marks the source dirty; it doesn't re-process on its own:

```
SELECT * FROM aidb.requeue_pipeline_errors(my_pipeline, ARRAY[3]);
__OUTPUT__
error_id | action
--+-+-----
 3 | requeued
(1 row)
```

Drain the requeued row — a Background pipeline picks this up automatically; Disabled or Live pipelines need an explicit call — then dismiss the two permanent failures after triage:

```
SELECT aidb.run_pipeline(my_pipeline);
```

(continues on next page)

(continued from previous page)

```
SELECT aidb.clear_error_logs(my_pipeline, ARRAY[1, 2]);
__OUTPUT__
clear_error_logs

 2
(1 row)
```

The status returns to UpToDate :

```
SELECT pipeline, "Status" FROM aidb.pipeline_metrics WHERE pipeline = my_pipeline;
__OUTPUT__
pipeline | Status
-----+-----
my_pipeline | UpToDate
(1 row)
```

## Pipelines API reference

This page covers the full public API for AI pipelines: the types and enums used across the API, the views for inspecting pipeline state, the core CRUD and execution functions, and the configuration helper functions for pipeline steps and vector indexes. For model configuration helpers, see the [Models reference](#).

### Types

#### aidb.PipelineAutoProcessingMode

Controls how a pipeline automatically processes new or changed data.

```
CREATE TYPE PipelineAutoProcessingMode AS ENUM (
 Live,
 Background,
 Disabled
);
```

Value	Description
<i>Live</i>	Processes new data immediately as it arrives, using Postgres triggers.
<i>Background</i>	Continuously processes data in the background using Postgres workers.
<i>Disabled</i>	No automated processing. Use <i>aidb.run_pipeline()</i> to trigger manually.

#### aidb.PipelineDataFormat

Specifies the format of source data the pipeline processes.

```
CREATE TYPE PipelineDataFormat AS ENUM (
 Text,
 Image,
 Pdf
);
```

Value	Description
<i>Text</i>	Plain text data.
<i>Image</i>	Image data (bytes).
<i>Pdf</i>	PDF documents.

### aidb.PipelineSourceType

Indicates the type of data source a pipeline reads from.

```
CREATE TYPE PipelineSourceType AS ENUM (
 Table,
 Volume,
 Empty
);
```

Value	Description
<i>Table</i>	A Postgres table or view.
<i>Volume</i>	A PGFS storage volume.
<i>Empty</i>	No source; the pipeline generates its own data.

### aidb.PipelineDestinationType

Indicates the type of destination a pipeline writes to.

```
CREATE TYPE PipelineDestinationType AS ENUM (
 Table,
 Volume,
 Empty
);
```

Value	Description
<i>Table</i>	A Postgres table.
<i>Volume</i>	A PGFS storage volume.
<i>Empty</i>	No destination; output is discarded.

### aidb.PipelineStepOperation

Defines the operation performed by a pipeline step.

```
CREATE TYPE PipelineStepOperation AS ENUM (
 ChunkText,
 SummarizeText,
 ParseHtml,
 ParsePdf,
 PerformOcr,
 KnowledgeBase,
 PdfToImage,
 SemanticKB
);
```

Value	Description
<i>ChunkText</i>	Splits text into smaller chunks.
<i>SummarizeText</i>	Summarizes text using a language model.
<i>ParseHtml</i>	Extracts text content from HTML.
<i>ParsePdf</i>	Extracts text or images from PDFs.
<i>PerformOcr</i>	Runs optical character recognition on images.
<i>KnowledgeBase</i>	Computes and stores embeddings in a knowledge base.
<i>PdfToImage</i>	Converts PDF pages to images.
<i>SemanticKB</i>	Indexes schema metadata into a semantic knowledge base.

### aidb.PipelineStatus

Represents the current processing state of a pipeline.

```
CREATE TYPE PipelineStatus AS ENUM (
 Stale,
 Processing,
 UpToDate,
 NoResults,
 Failed,
 Unknown,
 PartialErrors,
 BlockingErrors
);
```

Value	Description
<i>Stale</i>	Source data has changed and the pipeline needs to run.
<i>Processing</i>	The pipeline is currently executing.
<i>UpToDate</i>	All source data has been processed successfully.
<i>NoResults</i>	Processing completed but produced no output.
<i>Failed</i>	The last execution failed.
<i>Unknown</i>	Status can't be determined.
<i>PartialErrors</i>	Some records failed with record-level errors. The pipeline is otherwise operational.
<i>BlockingErrors</i>	A pipeline-level error stopped processing. Resolve the issue, re-run, then clear the entry.

### aidb.ErrorBlocking

Categorizes how a logged error affects pipeline processing. Two dimensions: scope (record vs pipeline) and temporality (temporary vs permanent).

```
CREATE TYPE ErrorBlocking AS ENUM (
 RecordTemporary,
 RecordPermanent,
 PipelineTemporary,
 PipelinePermanent
);
```

Value	Description
<i>RecordTemporary</i>	Transient record-level failure. Other records keep processing. Currently fires only for model rate-limit errors.
<i>RecordPermanent</i>	Permanent record-level failure (for example, bad input data). Other records keep processing.
<i>PipelineTemporary</i>	Transient pipeline-level failure (for example, network timeout, service outage). Blocks the step; may resolve on retry.
<i>PipelinePermanent</i>	Permanent pipeline-level failure (for example, deleted model, invalid config). Blocks the step.

## aidb.DistanceOperator

Specifies the distance metric used for vector similarity search.

```
CREATE TYPE DistanceOperator AS ENUM (
 L2,
 InnerProduct,
 Cosine,
 L1,
 Hamming,
 Jaccard
);
```

Value	Description
<i>L2</i>	Euclidean distance.
<i>InnerProduct</i>	Inner product.
<i>Cosine</i>	Cosine similarity.
<i>L1</i>	L1 (Manhattan) distance.
<i>Hamming</i>	Hamming distance.
<i>Jaccard</i>	Jaccard distance.

## Domains

### aidb.pipeline\_name\_50

A TEXT domain enforcing that pipeline names are no longer than 50 characters.

### aidb.background\_sync\_interval

An INTERVAL domain enforcing that background sync intervals are between 1 second and 2 days (inclusive).

## Views

### aidb.pipelines

Also accessible as `aidb.pipes`. Lists all registered pipelines and their configuration, including source, destination, processing mode, and step definitions.

Column	Type	Description
<i>id</i>	integer	Internal pipeline identifier.
<i>name</i>	text	Name of the pipeline.
<i>source_type</i>	<i>aidb.PipelineSourceType</i>	Whether the source is a table or volume.
<i>source_schema</i>	text	Schema of the source table.
<i>source</i>	text	Name of the source table or volume.
<i>source_key_column</i>	text	Column used as the unique key in the source.
<i>source_data_column</i>	text	Column containing the data to process.
<i>destination_type</i>	<i>aidb.PipelineDestinationType</i>	Whether the destination is a table or volume.
<i>destination_schema</i>	text	Schema of the destination table.
<i>destination</i>	text	Name of the destination table or volume.
<i>destination_key_column</i>	text	Key column in the destination table.
<i>destination_data_column</i>	text	Column in the destination where processed data is written.
<i>steps</i>	jsonb	Ordered array of pipeline step definitions.
<i>auto_processing</i>	<i>aidb.PipelineAutoProcessingMode</i>	Auto-processing mode.
<i>batch_size</i>	integer	Number of records processed per batch.
<i>background_sync_interval</i>	interval	Interval between executions in background mode.
<i>owner_role</i>	text	Postgres role that owns this pipeline.

### Example

```
SELECT name, source, destination, auto_processing FROM aidb.pipelines;
```

### aidb.pipeline\_metrics

Also accessible as `aidb.pipem`. Shows current processing statistics for each pipeline.

Column	Type	Description
<i>pipeline</i>	text	Name of the pipeline.
<i>auto_processing</i>	text	Current auto-processing mode.
<i>table: unprocessed rows</i>	bigint	For table sources: number of rows not yet processed.
<i>volume: scans completed</i>	bigint	For volume sources: number of full scans completed.
<i>count(source records)</i>	bigint	Total number of records in the source.
<i>count(destination records)</i>	bigint	Total number of records in the destination.
<i>Status</i>	text	Current <i>Pipeline status integration</i> .
<i>count(record errors)</i>	bigint	Number of record-level errors logged for this pipeline ( <i>RecordTemporary</i> plus <i>RecordPermanent</i> ).
<i>count(blocking errors)</i>	bigint	Number of pipeline-level errors logged for this pipeline ( <i>PipelineTemporary</i> plus <i>PipelinePermanent</i> ).
<i>last run completed</i>	timestampz	When the pipeline's last run finished. Introduced in 7.6.0.

## Example

```
SELECT * FROM aidb.pipeline_metrics;
```

\_\_OUTPUT\_\_

```

 pipeline | auto processing | table: unprocessed rows | volume: scans_
→ completed | count(source records) | count(destination records) | Status |
→ count(record errors) | count(blocking errors) | last run completed
-----+-----+-----+-----+-----+
→ -----+-----+-----+-----+-----+
→ -----+-----+-----+-----+-----+
pipeline__7471a | Background | 0 | 5 | UpToDate |
→ | 5 | 0 | 5 | UpToDate |
→ 0 | 0 | 2026-07-21 10:15:03.221+00
pipeline__7471b | Background | 0 | 5 | UpToDate |
→ | 5 | 0 | 5 | UpToDate |
→ 0 | 0 | 2026-07-21 10:15:03.402+00
animal_facts_kb | Disabled | 0 | 99 | UpToDate |
→ | 99 | 0 | 99 | UpToDate |
→ 0 | 0 | 2026-07-20 09:02:11.010+00
animal_facts_kb_bert | Disabled | 0 | 99 | UpToDate |
→ | 99 | 0 | 99 | UpToDate |
→ 0 | 0 | 2026-07-20 09:02:11.010+00
mpkb_pipe_int | Disabled | 0 | 2 | UpToDate |
→ | 2 | 0 | 4 | UpToDate |
→ 0 | 0 | 2026-07-19 16:40:55.884+00
mpkb_pipe_text | Disabled | 0 | 2 | UpToDate |
→ | 2 | 0 | 4 | UpToDate |
→ 0 | 0 | 2026-07-19 16:40:55.884+00
(6 rows)
```

## Functions

### aidb.create\_pipeline

Creates a new pipeline with a source, up to 10 sequential processing steps, and an optional destination.

### Parameters

Parameter	Type	Default	Description
<i>name</i>	TEXT	Required	Name of the pipeline. Max 46 characters.
<i>source</i>	TEXT	Required	Name of the source table or volume.
<i>step_1</i>	<i>aidb.PipelineStep-Operation</i>	Required	Operation for the first pipeline step.
<i>source_key_column</i>	TEXT	NULL	Unique key column in the source table.
<i>source_data_column</i>	TEXT	NULL	Column containing the data to process.

continues on next page

Table 25 – continued from previous page

Parameter	Type	Default	Description
<i>destination</i>	TEXT	NULL	Name of the destination table or volume.
<i>auto_processing</i>	<i>aidb.PipelineAuto-ProcessingMode</i>	NULL	Auto-processing mode.
<i>batch_size</i>	INT	NULL	Number of records to process per batch.
<i>background_sync_interval</i>	INTERVAL	NULL	Interval between background executions. Must be between 1 second and 2 days.
<i>owner_role</i>	TEXT	NULL	Role to own and execute this pipeline.
<i>step_1_options</i>	JSONB	NULL	Configuration for step 1 (use the appropriate <i>Pipeline step config helpers</i> ). Any step can also set <i>intermediate_destination</i> here to persist its output — see <i>End-to-end example: Multi-step pipeline with intermediate storage</i> .
<i>step_2 ... step_10</i>	<i>aidb.PipelineStep-Operation</i>	NULL	Operation for steps 2–10.
<i>step_2_options ... step_10_options</i>	JSONB	NULL	Configuration for steps 2–10.

## Returns

Column	Type	Description
<i>name</i>	text	Name of the created pipeline.
<i>destination_type</i>	text	Type of the pipeline destination.
<i>destination_schema</i>	text	Schema of the destination.
<i>destination</i>	text	Name of the destination.
<i>destination_key_column</i>	text	Key column in the destination.
<i>destination_data_column</i>	text	Data column in the destination.

## Example

```
-- Single-step pipeline: chunk text from a table into a destination table
SELECT aidb.create_pipeline(
 name => my_chunker,
 source => source_docs,
 source_key_column => id,
 source_data_column => body,
 destination => chunked_docs,
 step_1 => ChunkText,
 step_1_options => aidb.chunk_text_config(200, 250, 25),
 auto_processing => Live
);
```

(continues on next page)

(continued from previous page)

```

-- Multi-step pipeline: parse PDF, then embed into a knowledge base
SELECT aidb.create_pipeline(
 name => pdf_to_kb,
 source => pdf_volume,
 destination => my_kb,
 step_1 => ParsePdf,
 step_1_options => aidb.pdf_parse_config(),
 step_2 => KnowledgeBase,
 step_2_options => aidb.knowledge_base_config(my_model, Text),
 auto_processing => Background,
 background_sync_interval => 60 seconds
);

```

### aidb.update\_pipeline

Updates the auto-processing settings for an existing pipeline.

#### Parameters

Parameter	Type	Default	Description
<i>name</i>	TEXT	Required	Name of the pipeline to update.
<i>auto_processing</i>	<i>aidb.PipelineAuto-ProcessingMode</i>	NULL	New auto-processing mode.
<i>batch_size</i>	INT	NULL	New batch size.
<i>background_sync_interval</i>	INTERVAL	NULL	New background sync interval.

#### Example

```

SELECT aidb.update_pipeline(my_chunker, auto_processing => Background, background_
↪sync_interval => 5 minutes);

```

### aidb.delete\_pipeline

Deletes a pipeline and its configuration. Doesn't delete the source or destination tables.

#### Parameters

Parameter	Type	Default	Description
<i>name</i>	TEXT	Required	Name of the pipeline to delete.

#### Example

```

SELECT aidb.delete_pipeline(my_chunker);

```

## aidb.run\_pipeline

Manually triggers a pipeline to execute immediately, regardless of its `auto_processing` mode.

### Parameters

Parameter	Type	Default	Description
<i>pipeline_name</i>	TEXT	Required	Name of the pipeline to run.

### Example

```
SELECT aidb.run_pipeline(my_chunker);
```

## aidb.get\_pipeline\_metrics

Returns current processing statistics for a pipeline.

### Parameters

Parameter	Type	Default	Description
<i>name</i>	TEXT	Required	Name of the pipeline.

### Returns

Column	Type	Description
<i>name</i>	text	Name of the pipeline.
<i>auto_processing</i>	text	Current auto-processing mode.
<i>count_events</i>	bigint	For table sources: number of unprocessed change events.
<i>count_source</i>	bigint	Total records in the source (for table sources).
<i>count_known_objects</i>	bigint	Total known objects (for volume sources).
<i>last_full_run_id</i>	bigint	ID of the last completed full scan (for volume sources).
<i>count_destination</i>	bigint	Total records in the destination.
<i>status</i>	text	Current <i>Pipeline status integration</i> .
<i>count_errors_record</i>	bigint	Number of record-level errors logged for this pipeline ( <i>RecordTemporary</i> plus <i>RecordPermanent</i> ).
<i>count_errors_pipeline</i>	bigint	Number of pipeline-level errors logged for this pipeline ( <i>PipelineTemporary</i> plus <i>PipelinePermanent</i> ).
<i>last_run_completed</i>	timestampz	When the pipeline's last run finished. Introduced in 7.6.0.

## Example

```
SELECT * FROM aidb.get_pipeline_metrics(my_chunker);
```

## Error log functions

Functions for inspecting, clearing, and requeuing failed pipeline records (`aidb.get_error_logs`, `aidb.clear_error_logs`, `aidb.get_error_log_summary`, `aidb.get_all_error_summaries`, `aidb.requeue_pipeline_errors`) are documented in the [Operations reference](#). See [Pipeline error log](#) for the end-to-end guide-style workflow.

## Pipeline step config helpers

These functions return a JSONB configuration object for use in `step_N_options` parameters of `aidb.create_pipeline`. None of them have a parameter for `intermediate_destination`—see [End-to-end example: Multi-step pipeline with intermediate storage](#) for how to add it to a step's options.

### `aidb.chunk_text_config`

Configures a `ChunkText` step to split text into smaller segments.

### Parameters

Parameter	Type	Default	Description
<i>desired_length</i>	INTEGER	Required	Target chunk size.
<i>max_length</i>	INTEGER	NULL	Maximum allowed chunk size.
<i>overlap_length</i>	INTEGER	NULL	Number of units to overlap between consecutive chunks.
<i>strategy</i>	TEXT	NULL	Chunking unit: <code>'chars'</code> (default) or <code>'words'</code> .

## Example

```
-- Chunk into ~200 character segments, max 250, with 25-character overlap
SELECT aidb.chunk_text_config(200, 250, 25, chars);
```

### `aidb.summarize_text_config`

Configures a `SummarizeText` step to summarize text using a language model.

### Parameters

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	Name of the model to use for summarization.
<i>chunk_config</i>	JSONB	NULL	Optional chunking config (from <i>aidb.chunk_text_config</i> ) applied before summarizing.
<i>prompt</i>	TEXT	NULL	Custom prompt to guide the summarization.
<i>strategy</i>	TEXT	NULL	'append' (default) or 'reduce'.
<i>reduction_factor</i>	INTEGER	NULL	With 'reduce' strategy: aggressiveness of each reduction pass (default: 3).
<i>inference_config</i>	JSONB	NULL	Optional inference settings (from <i>aidb.inference_config</i> ).

### Example

```
SELECT aidb.summarize_text_config(
 my_llm,
 chunk_config => aidb.chunk_text_config(100, 100, 10, words),
 prompt => Summarize concisely,
 strategy => reduce,
 reduction_factor => 4
);
```

### aidb.ocr\_config

Configures a PerformOcr step to extract text from images using a model.

#### Parameters

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	Name of the OCR model to use.

### Example

```
SELECT aidb.ocr_config(my_ocr_model);
```

### aidb.html\_parse\_config

Configures a ParseHtml step to extract text from HTML content.

#### Parameters

Parameter	Type	Default	Description
<i>method</i>	TEXT	NULL	Parsing method to use. If <i>NULL</i> , uses the default method.

### Example

```
SELECT aidb.html_parse_config();
```

### aidb.pdf\_parse\_config

Configures a `ParsePdf` step to extract content from PDF documents.

### Parameters

Parameter	Type	Default	Description
<i>method</i>	TEXT	NULL	Parsing method to use. If <i>NULL</i> , uses the default method.
<i>allow_partial_parsing</i>	BOOLEAN	NULL	When <i>true</i> , returns partial results if some pages can't be parsed.

### Example

```
SELECT aidb.pdf_parse_config(allow_partial_parsing => true);
```

### aidb.knowledge\_base\_config

Configures a `KnowledgeBase` step to compute and store embeddings.

### Parameters

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	Name of the embedding model to use.
<i>data_format</i>	<i>aidb.Pipeline-DataFormat</i>	Required	Format of the data being embedded.
<i>distance_operator</i>	<i>aidb.DistanceOperator</i>	NULL	Distance function for similarity search. Defaults to <i>L2</i> .
<i>vector_index</i>	JSONB	NULL	Vector index configuration (from a <i>Vector index config helpers</i> ).

### Example

```
SELECT aidb.knowledge_base_config(
 my_embedding_model,
 Text,
 distance_operator => Cosine,
```

(continues on next page)

(continued from previous page)

```
vector_index => aidb.vector_index_hnsw_config(m => 16, ef_construction => 64)
);
```

### aidb.knowledge\_base\_config\_from\_kb

Configures a KnowledgeBase step to attach a pipeline to an existing knowledge base rather than creating a new one.

#### Parameters

Parameter	Type	Default	Description
<i>data_format</i>	<i>aidb.Pipeline-DataFormat</i>	Required	Format of the data being embedded.

#### Example

```
SELECT aidb.knowledge_base_config_from_kb(Text);
```

### aidb.inference\_config

Builds an inference configuration object for use with language model steps such as `SummarizeText`. All parameters are optional.

#### Parameters

Parameter	Type	Default	Description
<i>system_prompt</i>	TEXT	NULL	System prompt prepended to each request.
<i>temperature</i>	DOUBLE PRECISION	NULL	Sampling temperature (higher = more random).
<i>max_tokens</i>	INTEGER	NULL	Maximum number of tokens to generate.
<i>top_p</i>	DOUBLE PRECISION	NULL	Nucleus sampling threshold.
<i>seed</i>	BIGINT	NULL	Random seed for reproducible outputs.
<i>repeat_penalty</i>	REAL	NULL	Penalty for repeated tokens.
<i>repeat_last_n</i>	INTEGER	NULL	Number of recent tokens to apply repeat penalty over.
<i>thinking</i>	BOOLEAN	NULL	Enable extended reasoning (supported models only).
<i>extra_args</i>	JSONB	NULL	Additional provider-specific inference arguments.

## Example

```
SELECT aidb.inference_config(
 system_prompt => You are a technical summarizer.,
 temperature => 0.3,
 max_tokens => 512
);
```

## Vector index config helpers

These functions return a JSONB configuration for the `vector_index` parameter of `aidb.knowledge_base_config`.

### `aidb.vector_index_hnsw_config`

Configures an **HNSW** index (pgvector).

#### Parameters

Parameter	Type	Default	Description
<i>vector_data_type</i>	TEXT	NULL	Vector storage type.
<i>m</i>	INTEGER	NULL	Maximum number of connections per node (default: 16).
<i>ef_construction</i>	INTEGER	NULL	Build-time search depth (default: 64).
<i>ef_search</i>	INTEGER	NULL	Query-time search depth.

!!!note HNSW supports a maximum of 2000 dimensions. For higher-dimensional vectors, use `aidb.vector_index_disabled_config()`.

The following table shows how each `distance_operator` value maps to a pgvector ops class:

<i>distance_operator</i>	Index ops class
<i>L2</i>	<i>vector_l2_ops</i>
<i>InnerProduct</i>	<i>vector_ip_ops</i>
<i>Cosine</i>	<i>vector_cosine_ops</i>
<i>L1</i>	<i>vector_l1_ops</i>

## Example

```
SELECT aidb.vector_index_hnsw_config(m => 16, ef_construction => 64);
```

**aidb.vector\_index\_ivfflat\_config**

Configures a pgvector [IVFFlat](#) index.

**Parameters**

Parameter	Type	Default	Description
<i>vector_data_type</i>	TEXT	NULL	Vector storage type.
<i>lists</i>	INTEGER	NULL	Number of clusters (inverted lists).
<i>probes</i>	INTEGER	NULL	Number of clusters to search at query time.

**Example**

```
SELECT aidb.vector_index_ivfflat_config(lists => 100);
```

**aidb.vector\_index\_chord\_hnsw\_config**

Configures a VectorChord HNSW index.

**Parameters**

Parameter	Type	Default	Description
<i>vector_data_type</i>	TEXT	NULL	Vector storage type.
<i>m</i>	INTEGER	NULL	Maximum connections per node.
<i>ef_construction</i>	INTEGER	NULL	Build-time search depth.
<i>max_connections</i>	INTEGER	NULL	Maximum connections in the graph.
<i>ml</i>	DOUBLE PRECISION	NULL	Level multiplier controlling graph layer structure.

**Example**

```
SELECT aidb.vector_index_chord_hnsw_config(m => 16, ef_construction => 64);
```

**aidb.vector\_index\_chord\_vchordq\_config**

Configures a VectorChord `vchordq` index.

**Parameters**

Parameter	Type	Default	Description
<i>vector_data_type</i>	TEXT	NULL	Vector storage type.
<i>lists</i>	TEXT	NULL	Number of clusters.
<i>spherical_centroids</i>	BOOLEAN	NULL	Use spherical (normalized) centroids when clustering.

### Example

```
SELECT aidb.vector_index_chord_vchordq_config(lists => 100, spherical_centroids =>
→ true);
```

### aidb.vector\_index\_hsphere\_optimized\_config

Configures an HSphere Optimized index.

#### Parameters

Parameter	Type	Default	Description
<i>clusters</i>	INTEGER	Required	Number of clusters.
<i>precision_val</i>	DOUBLE PRECISION	Required	Indexing precision value.
<i>vector_data_type</i>	TEXT	NULL	Vector storage type.

### Example

```
SELECT aidb.vector_index_hsphere_optimized_config(clusters => 256, precision_val => 0.
→ 95);
```

### aidb.vector\_index\_disabled\_config

Disables automatic vector index creation. Use this when your embedding dimensions exceed 2000 or when you want to manage indexes manually.

### Example

```
SELECT aidb.vector_index_disabled_config();
```

### Model config helpers

Model config helpers have moved to the [Models reference](#) page.

### Configuration

Session-settable GUCs that control AI pipeline behavior.

### aidb.pipeline\_error\_warnings

Property	Value
Type	boolean

continues on next page

Table 46 – continued from previous page

Property	Value
Default	<i>true</i>
Context	<i>SUSET</i>

Superusers can change this setting per session, per role, per database, or in `postgresql.conf`. Other roles need an explicit `GRANT SET ON PARAMETER aidb.pipeline_error_warnings` before they can `SET` it.

When enabled, each error logged to a pipeline’s error log table also emits a Postgres `WARNING`. Errors persist to the log regardless of this setting. Disable to silence the warnings without affecting persistence:

```
SET aidb.pipeline_error_warnings = false;
```

## Examples

Pipelines are the primary way to define AI workflows in EDB Postgres AI. Each pipeline consists of one or more sequential steps where data flows linearly: the output of step 1 becomes the input for step 2, and so on.

### Creating a pipeline (table source)

To create a pipeline using a standard Postgres table as your data source, use the `aidb.create_pipeline()` function. The following examples show how to set up both a single-step and a multi-step pipeline.

#### Single-step pipeline

This example creates a knowledge base from a Postgres table. After running the pipeline, you can query the resulting embeddings for semantic search.

```
SELECT aidb.create_pipeline(
 name => kb_pipeline_table,
 source => source_table,
 source_key_column => id,
 source_data_column => content,
 step_1 => KnowledgeBase,
 step_1_options => aidb.knowledge_base_config(
 model => bert, -- this is a pre-defined locally running model
 data_format => Text
)
);
```

#### Multi-step pipeline

You can chain multiple operations. This example first parses HTML and then chunks the resulting text.

```
SELECT aidb.create_pipeline(
 name => html_processing_pipeline,
 source => web_data_table,
 source_key_column => id,
 source_data_column => html_content,
 step_1 => ParseHtml,
 step_2 => ChunkText
);
```

## Creating a pipeline (volume source)

If your data lives in external storage (like an S3 bucket), you must first define a storage location and a volume.

### Step 1: Define storage and volume

Create the PGFS storage location and AIDB volume.

```
SELECT pgfs.create_storage_location(s3_bucket_location, s3://my-ai-data,
 options => {"region": "us-east-1", "skip_signature": "true"}
);
SELECT aidb.create_volume(source_volume, s3_bucket_location, /, Text);
```

### Step 2: Create the pipeline

```
SELECT aidb.create_pipeline(
 name => pipeline_from_s3,
 source => source_volume,
 step_1 => KnowledgeBase,
 step_1_options => aidb.knowledge_base_config(
 model => dummy,
 data_format => Text
)
);
```

## Running and updating pipelines

Pipelines are disabled by default upon creation. You can trigger them manually or enable automatic processing.

### Manual execution

Run the pipeline once on all existing data:

```
SELECT aidb.run_pipeline(kb_pipeline_table);
```

### Enable auto-processing

Keep the pipeline up-to-date as source data changes:

```
SELECT aidb.update_pipeline(
 name => kb_pipeline_table,
 auto_processing => Live
);
```

## Monitoring and deletion

To view your existing pipelines and their configurations, query the `aidb.pipelines` view:

```
SELECT * FROM aidb.pipelines;
__OUTPUT__
 name | source_type | source_schema | source | source_key_

column | source_data_column | destination_type | destination_schema |
```

(continues on next page)

(continued from previous page)

```

→ destination | destination_key_column | destination_data_column |
→ | steps |
→ | auto_processing | batch_size | background_sync_interval |
→ owner_role
-----+-----+-----+-----+
→ -+-----+-----+-----+-----+
→ -+-----+-----+-----+-----+
→ -+-----+-----+-----+-----+
→ -+-----+-----+-----+-----+
→ -----
pipeline__8989 | Table | public | source_table__8989 | id
→ | content | Table | public | pipeline_pipeline__
→ 8989 | source_id | value | [{"options": {"max_length
→ ": null, "desired_length": 1000}, "operation": "ChunkText", "step_order": 1}] |
→ Live | 42 | @ 42 mins | role_pipeline_management_
→ single_step
(1 row)

```

```
SELECT name, source, auto_processing FROM aidb.pipelines;
__OUTPUT__
 name | source | auto_processing
-----+-----+-----
 pipeline__8989 | source_table__8989 | Live
(1 row)
```

Deleting a pipeline will also drop any destination tables generated by the pipeline steps. To delete a pipeline:

```
SELECT aedb.delete_pipeline(kb_pipeline_table);
```

## End-to-end example: Knowledge base for RAG

This example builds a complete retrieval-augmented generation (RAG) flow over an internal support knowledge base, entirely in SQL and entirely local: vector **retrieval** finds candidate articles, a cross-encoder **reranker** refines their order, and a **generation** model synthesizes an answer grounded in the retrieved articles. A reduced version of this example appears on the AI pipelines landing page .

## Step 1: Register the models

Each stage uses a different kind of model. The embedding model, `bge-small-en-v1.5-f16`, is pre-registered on every AIDB install, so only the reranking and generation models need registering. `aidb.create_model()` downloads and validates each GGUF file from Hugging Face (the reranker is ~420 MB, Llama-3.2-3B ~3.4 GB); the `revision` pins make the example reproducible — without them, the model repository’s mutable `main` branch is used.

```
- - Reranking model (cross-encoder)
SELECT aidb.create_model(
 name => bge-reranker-v2-m3-Q4,
 provider => llamacpp_reranking,
 config => jsonb_build_object(
 model, gpustack/bge-reranker-v2-m3-GGUF,
 model_file, bge-reranker-v2-m3-Q4_K_M.gguf,
 revision, 3093af03b1a635e67b084b1d8c03c5f5e020fd05,
 n_ctx, 2048
```

(continues on next page)

(continued from previous page)

```

)
);

-- Generation model (instruct LLM)
SELECT aidb.create_model(
 name => llama-3.2-3b-instruct-Q8_0,
 provider => llamacpp_generate,
 config => jsonb_build_object(
 model, unsloth/Llama-3.2-3B-Instruct-GGUF,
 model_file, Llama-3.2-3B-Instruct-Q8_0.gguf,
 revision, e7d0997e49c9cb00d88b4c1a6a16aa894b0bbc31,
 n_ctx, 8192,
 temperature, 0.0
)
);

```

!!!note Model sizing for the generation stage 1B-class generation models (including the pre-registered llama-3.2-1b-instruct-Q8\_0 ) aren't reliable for the generation stage of this example — in testing, they recommended actions that contradict the question's constraints. Llama-3.2-3B-Instruct answers correctly and is still a modest local model.

## Step 2: Create the source data

Sixteen short support articles about database operations. Several deliberately share vocabulary (disk space, vacuum, backups) — that's what makes the reranking stage visibly matter later:

```

CREATE TABLE support_articles (
 id INT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
 title TEXT NOT NULL,
 body TEXT NOT NULL
);

INSERT INTO support_articles (title, body) VALUES
(Reclaiming disk space with VACUUM FULL,
 VACUUM FULL rewrites a table into a new file and returns unused disk space to the
→operating system. It takes an ACCESS EXCLUSIVE lock, so all reads and writes to the
→table are blocked until it completes.),
(Reducing table bloat without downtime,
 The pg_repack extension rebuilds bloated tables and indexes online. It holds only
→brief locks, so applications can continue reading and writing while space is
→reclaimed. It requires free disk space roughly equal to the table size.),
(Monitoring database disk usage,
 Use pg_database_size() and pg_total_relation_size() to track how much disk space
→databases and tables occupy. Configure alerts well before storage volumes reach
→capacity.),
(Resolving "too many connections" errors,
 The error "FATAL: sorry, too many clients already" means the max_connections limit
→is reached. Find idle sessions in pg_stat_activity and terminate them with pg_
→terminate_backend(), or introduce a connection pooler.),
(Connection pooling with PgBouncer,
 PgBouncer multiplexes thousands of client connections onto a small pool of database
→sessions. Transaction pooling mode provides the highest connection density for

```

(continues on next page)

(continued from previous page)

```

→typical web application workloads.),
(Diagnosing streaming replication lag,
 Compare pg_current_wal_lsn() on the primary with pg_last_wal_replay_lsn() on the
→standby to measure replication lag. Common causes include network saturation and
→long-running queries on the standby holding back WAL replay.),
(Creating a hot standby with pg_basebackup,
 Run pg_basebackup to clone the primary server, then start the clone with a standby.
→signal file. The standby serves read-only queries while continuously replaying WAL
→from the primary.),
(Replicating selected tables with logical replication,
 Logical replication publishes row changes for selected tables to subscriber
→databases and keeps them continuously synchronized. Unlike streaming replication,
→it can replicate a subset of tables and works across major versions.),
(Point-in-time recovery with WAL archiving,
 With continuous WAL archiving enabled, you can restore a base backup and set
→recovery_target_time to roll the database forward to a specific moment, such as
→just before an accidental data deletion.),
(Logical backups with pg_dump,
 pg_dump exports a database or single tables to a portable archive that can be
→restored into newer PostgreSQL versions. Dumps are consistent snapshots but offer
→no incremental or point-in-time capability.),
(Reading query plans with EXPLAIN ANALYZE,
 EXPLAIN ANALYZE executes a query and reports the actual time spent in each plan node.
→ Look for row-estimate mismatches and sequential scans on large tables to find
→missing indexes.),
(Finding and removing unused indexes,
 Query pg_stat_user_indexes for indexes with zero scans. Unused indexes slow down
→every write and consume disk space; dropping them is a safe, immediate win.),
(Tuning work_mem for large sorts,
 When sorts or hash joins exceed work_mem, they spill to temporary files on disk and
→slow down dramatically. Raise work_mem session-locally for reporting queries rather
→than globally.),
(Restricting data access with row-level security,
 Row-level security policies filter which rows each role can see or modify. Policies
→are enforced by the server itself, so they apply uniformly to every client and
→application.),
(Auditing database activity with pgaudit,
 The pgaudit extension produces detailed audit logs of reads, writes, and DDL for
→compliance requirements. Configure log classes narrowly to control log volume.),
(Upgrading major versions with pg_upgrade,
 pg_upgrade migrates a cluster to a new major version by relinking data files instead
→of dumping and reloading. With the --link option, downtime is typically minutes
→even for large clusters.);

```

### Step 3: Create and run the knowledge base pipeline

```

SELECT aidb.create_pipeline(
 name => support_kb,
 source => support_articles,
 source_key_column => id,
 source_data_column => body,

```

(continues on next page)

(continued from previous page)

```

step_1 => KnowledgeBase,
step_1_options => aidb.knowledge_base_config(
 model => bge-small-en-v1.5-f16,
 data_format => Text
),
auto_processing => Disabled
);

SELECT aidb.run_pipeline(support_kb);

```

## Step 4: Retrieval

Semantic search finds the right article even when the query shares few or no keywords with it. Paste raw error text, get the resolution:

```

SELECT a.title, r.distance::numeric(5,3) AS distance
FROM aidb.retrieve_key(support_kb,
 application logs show: FATAL: sorry, too many clients already, 3) r
JOIN support_articles a ON a.id = r.key::int
ORDER BY r.distance;
__OUTPUT__

```

title	distance
Resolving "too many connections" errors	0.507
Tuning work_mem for large sorts	0.844
Connection pooling with PgBouncer	0.908

(3 rows)

Or describe a symptom that has no tool names or keywords in common with the answer:

```

SELECT a.title, r.distance::numeric(5,3) AS distance
FROM aidb.retrieve_key(support_kb,
 A developer accidentally deleted rows from a table yesterday. How do we get
 the data back?, 3) r
JOIN support_articles a ON a.id = r.key::int
ORDER BY r.distance;
__OUTPUT__

```

title	distance
Point-in-time recovery with WAL archiving	0.795
Reducing table bloat without downtime	0.806
Finding and removing unused indexes	0.846

(3 rows)

(aidb.retrieve\_text() returns the article body directly, with no join needed; the join variant is shown because titles read better in output tables.)

## Step 5: Reranking

Demo question: “The database is running out of disk space, but we cannot take the application offline to fix it.”

Vector search alone ranks *Point-in-time recovery* — clearly unhelpful for a disk-space problem — at #1, with the correct article (*pg\_repack*) second in a near-tie:

```

SELECT a.title, r.distance::numeric(5,3) AS distance
FROM aidb.retrieve_key(support_kb,
 The database is running out of disk space, but we cannot take the application
 ↪offline to fix it., 5) r
JOIN support_articles a ON a.id = r.key::int
ORDER BY r.distance;
__OUTPUT__

```

title	distance	
Point-in-time recovery with WAL archiving	0.808	← wrong
Reducing table bloat without downtime	0.838	← correct answer, near-tie
Tuning work_mem for large sorts	0.851	
Monitoring database disk usage	0.865	
Resolving "too many connections" errors	0.883	

(5 rows)

The cross-encoder reads the question and each article together and reorders decisively. `aidb.rerank_text()` returns `(text, logit_score, id)`, where `id` is the 0-based position in the input array and a higher `logit_score` means more relevant — the `id + 1` maps that index back into the 1-based SQL arrays:

```

WITH candidates AS (
 SELECT array_agg(a.title ORDER BY r.distance) AS titles,
 array_agg(a.body ORDER BY r.distance) AS bodies
 FROM aidb.retrieve_key(support_kb,
 The database is running out of disk space, but we cannot take the
 ↪application offline to fix it., 5) r
 JOIN support_articles a ON a.id = r.key::int
)
SELECT c.titles[r.id + 1] AS title, r.logit_score::numeric(7,3) AS logit_score
FROM candidates c,
 aidb.rerank_text(bge-reranker-v2-m3-Q4,
 The database is running out of disk space, but we cannot take the
 ↪application offline to fix it.,
 c.bodies) r
ORDER BY r.logit_score DESC;
__OUTPUT__

```

title	logit_score	
Reducing table bloat without downtime	-2.742	← promoted to #1, clear
Monitoring database disk usage	-3.116	
Point-in-time recovery with WAL archiving	-7.037	← demoted
Tuning work_mem for large sorts	-8.836	
Resolving "too many connections" errors	-9.216	

(5 rows)

This is the classic two-stage retrieval pattern: over-fetch candidates cheaply with the vector index (5 here; 20–100 in production), then let the more accurate but more expensive cross-encoder pick the final order. A reranker runs a full model forward pass per candidate — never rerank a whole table.

## Step 6: Generation

Full RAG in a single statement: retrieve 5 candidates, rerank and keep the best 3, and generate a grounded answer. The first two CTEs are exactly the Step-4 and Step-5 commands above. (\set is a psql feature — in other clients, inline the question text directly.)

```
\set question The database is running out of disk space, but we cannot take the
↪application offline to fix it. What should we do?

WITH candidates AS (
 -- Stage 1: vector search over the knowledge base (over-fetch 5 candidates)
 SELECT array_agg(a.title ORDER BY r.distance) AS titles,
 array_agg(a.body ORDER BY r.distance) AS bodies
 FROM aidb.retrieve_key(support_kb, :question, 5) r
 JOIN support_articles a ON a.id = r.key::int
),
reranked AS (
 -- Stage 2: cross-encoder reranking, keep the best 3
 SELECT c.titles[r.id + 1] AS title,
 c.bodies[r.id + 1] AS body,
 r.logit_score
 FROM candidates c,
 aidb.rerank_text(bge-reranker-v2-m3-Q4, :question, c.bodies) r
 ORDER BY r.logit_score DESC
 LIMIT 3
),
context AS (
 SELECT string_agg(### || title || E\n || body, E\n\n
 ORDER BY logit_score DESC) AS articles
 FROM reranked
)
-- Stage 3: grounded answer generation
SELECT aidb.generate_text(
 llama-3.2-3b-instruct-Q8_0,
 EKnowledge base articles:\n\n || context.articles ||
 E\n\nQuestion: || :question ||
 E\n\nAnswer the question using only the facts in the articles above. Recommend
 ↪one course of action and mention its caveats.,
 {
 "system_prompt": "You are a database support assistant. Be accurate and
 ↪concise; do not invent facts that are not in the provided articles.",
 "max_tokens": 250,
 "temperature": 0.0
 }::json
) AS answer
FROM context;
```

The output (deterministic at temperature: 0.0 — repeat runs produce identical answers):

```
Based on the provided articles, I recommend using the `pg_repack` extension to reduce
↪table
bloat without downtime.
```

```
According to the article, `pg_repack` rebuilds bloated tables and indexes online,
```

(continues on next page)

(continued from previous page)

```

↪holding only
brief locks, allowing applications to continue reading and writing while space is_
↪reclaimed.
This approach does not require taking the application offline.

However, there is a caveat: `pg_repack` requires free disk space roughly equal to the_
↪table
size. This means that you need to ensure there is sufficient free disk space_
↪available before
running `pg_repack`, which may require some planning and preparation.

```

No single article contains this answer: the model combined the constraint from the question (“cannot go offline”) with facts from two competing articles (VACUUM FULL blocks reads and writes; pg\_repack works online) and carried over the free-disk-space caveat — a synthesis, not a verbatim return.

The same statement works with any registered generation model, including hosted ones — only the model name changes. One caveat for hosted GPT-5.x reasoning models: omit "temperature" from the inference config; they reject the parameter.

## Step 7: Teardown

```

SELECT aidb.delete_pipeline(support_kb);
DROP TABLE support_articles;
SELECT aidb.delete_model(llama-3.2-3b-instruct-Q8_0);
SELECT aidb.delete_model(bge-reranker-v2-m3-Q4);

```

(The embedding model isn’t deleted — bge-small-en-v1.5-f16 is one of the pre-registered defaults.)

## End-to-end example: Multi-step pipeline with intermediate storage

This end-to-end example demonstrates how to configure a multi-step pipeline in EDB Postgres AI that utilizes intermediate storage.

By defining an `intermediate_destination`, you can persist the results of individual steps (like text chunking) before they pass to the next stage (like summarization), which is useful for debugging or reusing processed data. It’s off by default, and no step config helper (`aidb.chunk_text_config()`, and so on) has a parameter for it — both examples below add it as plain JSON alongside the helper’s own options. See [End-to-end example: Multi-step pipeline with intermediate storage](#) for the full reference, including how this interacts with `deduplicate` at query time.

### Key features of this example:

- **Multi-step workflow:** Chains `ChunkText` (Step 1) into `SummarizeText` (Step 2).
- **Auto-processing:** The pipeline automatically triggers whenever new data is inserted into the source table.
- **Intermediate storage:** Demonstrates how to explicitly name an intermediate table versus allowing the system to auto-generate one.

### Example 1: Named intermediate destination

In this scenario, we will manually name the table that stores the output of the chunking step.

- Create a source table for your raw text:

```
CREATE TABLE source_table_demo(
 id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
 content TEXT NOT NULL
);
```

- Define the pipeline with a custom intermediate table:

```
SELECT * from aidb.create_pipeline(
 name => complex_pipeline_v1,
 source => source_table_demo,
 source_key_column => id,
 source_data_column => content,
 auto_processing => Live,
 step_1 => ChunkText,
 step_1_options => jsonb_build_object(
 desired_length, 20,
 intermediate_destination, jsonb_build_object(
 enabled, true,
 destination, my_custom_chunk_storage -- Explicitly named
)
),
 step_2 => SummarizeText
);
```

- Insert data to trigger auto-processing:

```
INSERT INTO source_table_demo (content)
VALUES (This is a long text example that will be split into segments for easier_
→processing.);
```

- View the chunks and final summaries:

```
-- View intermediate chunked output
SELECT * FROM my_custom_chunk_storage;

-- View the final summarized output
SELECT * FROM pipeline_complex_pipeline_v1;
```

## Example 2: Automatic intermediate destination

If you enable intermediate storage but don't provide a name, the system generates a table following the pattern `pipeline_[name]_step_[n]`.

- Create a pipeline with auto-named intermediate storage:

```
SELECT * from aidb.create_pipeline(
 name => auto_named_pipeline,
 source => source_table_demo_2,
 source_key_column => id,
 source_data_column => content,
 auto_processing => Live,
 step_1 => ChunkText,
 step_1_options => jsonb_build_object(
```

(continues on next page)

(continued from previous page)

```

 desired_length, 20,
 intermediate_destination, jsonb_build_object(enabled, true)
),
 step_2 => SummarizeText
);

```

- Access the auto-generated intermediate table:

```

-- View intermediate chunked output (auto-generated table name)
SELECT * FROM pipeline_auto_named_pipeline_step_1;

```

## End-to-end example: Scanned PDFs to a searchable knowledge base

This example shows how to build a full pipeline that turns a collection of scanned PDF documents into a vector-searchable knowledge base. It uses three sequential steps:

1. **PdfToImage**: renders each PDF page as an image.
2. **PerformOcr**: extracts text from each rendered image using an OCR model.
3. **KnowledgeBase**: embeds the extracted text and stores it for semantic search.

This pattern is ideal for scanned documents, legacy PDFs without a text layer, and any content where **ParsePdf** returns empty or low-quality text. By leveraging intermediate storage, you can inspect the OCR output before it feeds into the knowledge base, ensuring data quality and debugging any issues in the workflow.

## Prerequisites

You need a registered NVIDIA NIM PaddleOCR model and a registered embedding model. If you don't have these yet, register them first:

```

-- Register the OCR model
SELECT aidb.create_model(
 my_paddle_ocr_model,
 nim_paddle_ocr,
 credentials => {"api_key": "<NVIDIA_NIM_API_KEY>"}::JSONB -- or credentials_env_
 -- to read this from an environment variable instead
);

```

The embedding model (bert) is pre-installed and needs no registration. If you're using a different model, register it here.

## Step 1: Create the source table

Create a table to hold the raw PDF data:

```

CREATE TABLE scanned_pdfs (
 id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
 name TEXT NOT NULL,
 pdf_data BYTEA NOT NULL
);

```

## Step 2: Load PDFs into the table

Insert PDF documents as binary data. The example below uses `pg_read_binary_file()` to load files from the server's file system:

```
INSERT INTO scanned_pdfs (name, pdf_data)
VALUES (quarterly_report.pdf, pg_read_binary_file(/path/to/quarterly_report.pdf));
```

You can also load PDFs from an S3-compatible volume — see [Configuring external storage](#) and [Creating a pipeline \(volume source\)](#).

## Step 3: Create the pipeline

```
SELECT aidb.create_pipeline(
 name => scanned_pdf_kb,
 source => scanned_pdfs,
 source_key_column => id,
 source_data_column => pdf_data,
 step_1 => PdfToImage,
 step_1_options => [{"dpi": 150}>::jsonb,
 step_2 => PerformOcr,
 step_2_options => aidb.ocr_config(model => my_paddle_ocr_model),
 step_3 => KnowledgeBase,
 step_3_options => aidb.knowledge_base_config(
 model => bert,
 data_format => Text,
 distance_operator => Cosine
),
 auto_processing => Live
);
```

With `auto_processing` set to `Live`, the pipeline will automatically run whenever new PDFs are inserted into the `scanned_pdfs` table.

## Step 4: Run the pipeline on existing data

If you inserted rows before enabling the pipeline, process them now:

```
SELECT aidb.run_pipeline(scanned_pdf_kb);
```

## Step 5: Query the knowledge base

After the pipeline finishes, query the results using semantic search:

```
SELECT * FROM aidb.retrieve_text(
 scanned_pdf_kb,
 quarterly revenue figures,
 5 -- return top 5 matches
);
```

Each row in the result corresponds to a text block extracted from a PDF page, ranked by semantic similarity to your query.



---

Knowledge bases

---

A knowledge base (KB) stores embeddings in a vector table, along with its configuration — the model, distance operator, and vector index. Pipelines write to it, and you query it by name using `aidb.retrieve_text()` or `aidb.retrieve_key()`.

A knowledge base starts with a single pipeline and a *KnowledgeBase*. You can attach additional pipelines to the same knowledge base — letting different sources or processing chains all contribute embeddings to one queryable vector table.

Page	What it covers
<i>Multi-pipeline knowledge bases</i>	Attach multiple pipelines to one KB; compatibility rules and deletion semantics
<i>Hybrid search</i>	Combining semantic search with relational filters and full-text search
<i>Examples</i>	End-to-end worked examples for table and volume sources

## 11.1 Retrieval functions

Once a pipeline has run, query the knowledge base using `aidb.retrieve_text()` or `aidb.retrieve_key()`. Both use vector similarity to find results based on meaning rather than exact keywords, and support both `TEXT` and `BYTEA` (image) as the query input.

### 11.1.1 Flow of retrieval functions

When a retrieval function is called, the system performs the following steps internally:

- **Embedding:** The input query (text or image) is converted into a vector using the specific embedding model configured for that knowledge base.
- **Similarity search:** A vector similarity search is performed against the knowledge base's internal vector table to find the Top K nearest neighbors.
- **Source lookup (text only):** For `retrieve_text`, the system identifies the source table and retrieves the raw content corresponding to the matched keys.

### 11.1.2 `aidb.retrieve_text()`

Use this function when you need to retrieve the actual source text associated with the closest vector matches.

- **Process:** The function embeds your query, performs a similarity search, and then executes a second phase to look up the source text from the original table using the `pipeline_id`.
- **Returns:** A set of columns including:
  - **key:** The identifier from the source table.
  - **value:** The actual source text.
  - **distance:** The similarity score. A lower value usually indicates a closer match.
  - **part\_ids:** An array of IDs indicating which specific chunks or parts were matched.
  - **pipeline\_name:** The name of the pipeline that supplied the data.
  - **intermediate\_steps:** A JSONB column containing data from steps occurring before the knowledge base, for any step configured with *intermediate\_destination* — for example, `ChunkText`. Empty if no step in the pipeline has intermediate storage enabled. See *Deduplication and intermediate results* below for how this interacts with `deduplicate`.

### 11.1.3 `aidb.retrieve_key()`

Use this function for high-performance searches where you only need the unique identifiers of the matches, rather than the full source content.

- **Returns:** A set of columns including:
  - **key:** The identifier from the source table.
  - **distance:** The similarity score. A lower value usually indicates a closer match.
  - **part\_ids:** An array of IDs indicating which specific chunks or parts were matched.
  - **pipeline\_name:** The name of the pipeline that supplied the data.

## 11.2 Advanced querying: Joining intermediate steps

For pipelines that include intermediate transformations such as `ChunkText` or `ParseHtml`, you can access specific transformed segments by joining retrieval results with intermediate pipeline tables using the `part_ids` column. This is a manual alternative to the `intermediate_steps` column that `aidb.retrieve_text()` already returns automatically for any step with *End-to-end example: Multi-step pipeline with intermediate storage* enabled — reach for a manual join only when you need something `intermediate_steps` doesn't give you, such as joining on additional columns of the intermediate table.

### 11.2.1 Example syntax

The following query joins the retrieval results with an intermediate step table to access specific chunked values:

```
SELECT
 r.key,
 r.value,
 r.distance,
 r.part_ids,
 int_step.value AS chunked_content
FROM aidb.retrieve_text(my_kb, search query, 5) AS r
JOIN pipeline_my_pipeline_step_1 AS int_step
```

(continues on next page)

(continued from previous page)

```
ON int_step.source_id = r.key
AND int_step.part_ids = (r.part_ids)[:1];
```

Like `intermediate_steps`, this join only ever sees the one row `deduplicate` let through — see the next section.

## 11.3 Deduplication and intermediate results

`deduplicate` (default `true` on both `aidb.retrieve_key()` and `aidb.retrieve_text()`) collapses retrieval results to one row per source record, even when a step like `ChunkText` split that record into several independently embedded parts and more than one of those parts matches the query. Only one part “wins” — whichever is closest to the query — and its `part_ids` and `intermediate_steps` are the only ones returned for that source record.

This matters as soon as you enable *End-to-end example: Multi-step pipeline with intermediate storage* on a chunking or parsing step: with the default `deduplicate => true`, you only ever see **one** matching chunk’s intermediate content per source record, even if several of its chunks matched. To see every matching part — and each one’s own `intermediate_steps` entry — call retrieval with `deduplicate => false`.

For example, a pipeline chunks a set of short articles about animals, with `intermediate_destination` enabled so each chunk’s text is available in `intermediate_steps`. Querying for `'ocean'` with the default deduplication, the `whale` article’s best-matching chunk wins and no other `whale` chunk appears, even though a second one also matched:



As a rule of thumb: keep `deduplicate => true` (the default) when you want one representative result per source record, and use `deduplicate => false` whenever you need every matching chunk or part from the same source record — for example, to show all the passages within a long document that matched a query, not just the single best one.

## 12.1 Multi-pipeline knowledge bases

A knowledge base (KB) can have more than one pipeline writing into it. Each pipeline keeps its own source, processing chain, and auto-processing mode, but all of them store embeddings in the same vector table. Retrieval queries the KB once and aggregates results across all attached pipelines.

This is useful when you want to:

- Combine multiple source tables into a single searchable index.
- Mix data formats — for example, one pipeline indexes `Text` rows and another indexes `Pdf` files via a *Configuring external storage*.
- Apply different preprocessing per source — one pipeline runs `ChunkText` before embedding, another embeds short texts directly.

### 12.1.1 Attaching a pipeline to an existing KB

The first pipeline creates the KB; subsequent pipelines attach by writing to the same `destination`. Use `aidb.knowledge_base_config_from_kb()` for subsequent pipelines so they inherit the KB's model, distance operator, and vector index.

```
-- Pipeline 1 creates the KB from the North America customer feedback table
SELECT aidb.create_pipeline(
 name => feedback_na_pipe,
 source => customers_na,
 source_key_column => id,
 source_data_column => feedback,
```

(continues on next page)

(continued from previous page)

```

destination => customer_feedback_kb,
step_1 => KnowledgeBase,
step_1_options => aidb.knowledge_base_config(model => bert, data_format => Text)
);

-- Pipeline 2  attaches the Europe customer feedback table to the same KB
SELECT aidb.create_pipeline(
 name => feedback_eu_pipe,
 source => customers_eu,
 source_key_column => id,
 source_data_column => feedback,
 destination => customer_feedback_kb,
 step_1 => KnowledgeBase,
 step_1_options => aidb.knowledge_base_config_from_kb(data_format => Text)
);

SELECT aidb.run_pipeline(feedback_na_pipe);
SELECT aidb.run_pipeline(feedback_eu_pipe);

```

The query returns rows from both pipelines, with `pipeline_name` identifying the source:

```

SELECT key, value, pipeline_name
FROM aidb.retrieve_text(public.customer_feedback_kb, shipping was slow, topk => 3);
__OUTPUT__

```

key	value	pipeline_name
1042	Delivery was held up <b>in</b> customs <b>for</b> over a week	feedback_eu_pipe
318	<b>Order</b> took twice <b>as</b> long <b>to</b> arrive <b>as</b> estimated	feedback_na_pipe
1057	Tracking <b>number</b> never updated <b>until</b> it arrived	feedback_eu_pipe

Different step chains can share one KB — for example, one pipeline chunks long documents with `ChunkText` before the `KnowledgeBase` step while another embeds short rows directly. The `intermediate_steps` JSONB column from `aidb.retrieve_text()` returns per-pipeline intermediate results, for any step with *End-to-end example: Multi-step pipeline with intermediate storage* enabled. `deduplicate` (on by default) is scoped per pipeline, not across the whole KB — see *Deduplication and intermediate results*.

### 12.1.2 Compatibility rules

All pipelines attached to a KB must agree on the embedding shape. When you create a pipeline that targets an existing KB, AIDB validates:

Constraint	Error if violated
Model	<i>Model mismatch: existing KB uses '&lt;X&gt;' but pipeline specifies '&lt;Y&gt;'</i>
Distance operator	<i>Distance operator mismatch: ...</i>
Vector index config	<i>Vector index mismatch: ...</i>
Embedding dimensions	<i>Embedding dimension mismatch: existing KB vector column has N dimensions but model '&lt;X&gt;' produces M</i>

`aidb.knowledge_base_config_from_kb()` inherits model, distance operator, and vector index from the existing

KB, so it always passes the first three checks. The dimension check still runs and catches the edge case where a model is recreated under the same name but with a different output size.

### 12.1.3 Deletion semantics

Action	Effect
<code>aidb.delete_pipeline('one_of_many')</code>	Removes that pipeline and only its rows from the shared vector table. KB and other pipelines remain.
<code>aidb.delete_pipeline('the_last_one')</code>	Drops the KB registry entry and the vector table along with the pipeline.
<code>aidb.delete_knowledge_base('schema.table')</code>	Cascades — deletes every attached pipeline, the KB, and the vector table.

These operations never touch source tables or source volumes.

### 12.1.4 Inspecting attached pipelines

```
SELECT name, model_name, distance_operator, pipeline_names
FROM aidb.knowledge_bases
WHERE name = public.customer_feedback_kb;
__OUTPUT__
 name | model_name | distance_operator | pipeline_
--names
-----+-----+-----+-----
public.customer_feedback_kb | bert | L2 | {feedback_eu_pipe,
feedback_na_pipe}
```

For aggregate counts and status, query `aidb.knowledge_base_metrics` ; for per-pipeline progress, query `aidb.pipeline_metrics` .

### 12.1.5 Vector table schema

The shared vector table includes a `pipeline_id` column so you can trace each row back to its originating pipeline. The primary key is `(pipeline_id, source_id, part_ids)` .

```
SELECT pipeline_id, source_id, part_ids
FROM customer_feedback_kb
ORDER BY pipeline_id, source_id;
__OUTPUT__
pipeline_id | source_id | part_ids
-----+-----+-----
1 | 318 | {0}
1 | 412 | {0}
2 | 1042 | {0}
2 | 1057 | {0}
```

## 12.2 Hybrid search

Hybrid search combines semantic (vector/embedding) search on knowledge base embeddings with one or more additional filter mechanisms:

- **Relational predicate filtering** — Filter results by any column in the source table, for example `WHERE category = 'vegetarian' or WHERE published_date > '2024-01-01' .`
- **Full-text search** — Rank results by keyword relevance using native PostgreSQL FTS, ensuring that specific words or phrases appear in the results.

These two approaches can be used independently or together alongside semantic search.

AI pipelines handle knowledge base construction, including embedding generation, as normal. Hybrid search is applied at query time by constructing custom SQL that combines the vector table with your source table. It is not a separately installed feature and no additional setup is required beyond the prerequisites below.

### 12.2.1 Prerequisites

pgvector iterative index scans are required for all hybrid query patterns. Native FTS is only needed if your queries use the keyword leg.

#### pgvector iterative index scans

Requires pgvector 0.8.0 or later. Enable iterative index scans before running hybrid queries — they allow the vector index to keep producing candidates until your `LIMIT` is satisfied, even after filtering. See [Writing hybrid search queries](#) for the `SET` commands.

#### Native PostgreSQL FTS (no extension required)

Standard PostgreSQL full-text search (FTS) (`tsvector/ts_rank`) is available in every Postgres installation without additional extensions. See [Setting up native PostgreSQL FTS](#) for setup steps.

### 12.2.2 Volume source limitation

Hybrid search with relational predicate filtering or full-text search is not supported for volume-sourced knowledge bases. Volume sources have no backing Postgres table to join against, so there's no source table to filter or index.

Volume sourced knowledge bases continue to support standard semantic search via `aidb.retrieve_text()` and `aidb.retrieve_key()` .

## 12.3 Hybrid search

### 12.3.1 Next steps

- [Setting up native PostgreSQL FTS](#) — Add a `tsvector` column, GIN index, and trigger to your source table.
- [Writing hybrid search queries](#) — SQL patterns for every combination: relational filter, Reciprocal Rank Fusion (RRF), weighted linear fusion. Includes a worked example with sample data and expected output.
- [Handling embedding service outages](#) — A three-tier fallback chain to keep queries returning results gracefully.
- [Pipelines API reference](#) — Full reference for `aidb.kb_query_encode()` , `aidb.rerank_text()` , and the `aidb.knowledge_bases` view.

## Setting up native PostgreSQL FTS

Configure your source table with a `tsvector` column, a GIN index, and an autoupdate trigger to use native PostgreSQL full-text search (FTS) as the keyword leg in hybrid search queries. No additional extensions are required — all steps below are SQL commands run in your database. The setup includes field weighting so title matches rank higher than body matches, and a trigger so the search column stays current as rows are inserted or updated.

### 1. Add a `tsvector` column

Add a `tsvector` column to store the pre-processed text representation of each row. This is the column the FTS index operates on.

```
ALTER TABLE my_docs ADD COLUMN search_vector tsvector;
```

### 1. Populate the column with weighted fields

Populate the column using weighted fields to control how different fields affect relevance. Use weight A (highest) through D (lowest) — assign higher weights to fields like title and summary, and lower weights to longer body fields.

```
UPDATE my_docs
SET search_vector =
 setweight(to_tsvector(english, coalesce(title,)), A) ||
 setweight(to_tsvector(english, coalesce(summary,)), B) ||
 setweight(to_tsvector(english, coalesce(body,)), C);
```

### 1. Create a GIN index

Create a GIN (Generalized Inverted Index) on the `search_vector` column to enable fast full-text lookups. Without this index, FTS queries perform a sequential scan and won't scale.

```
CREATE INDEX my_docs_fts_idx ON my_docs USING gin(search_vector);
```

### 1. Create a trigger to keep the column current

Set up a trigger to tokenize new and updated rows automatically, so the FTS index stays in sync without manual updates:

```
CREATE OR REPLACE FUNCTION my_docs_search_vector_update()
RETURNS trigger AS $$
BEGIN
 NEW.search_vector :=
 setweight(to_tsvector(english, coalesce(NEW.title,)), A) ||
 setweight(to_tsvector(english, coalesce(NEW.summary,)), B) ||
 setweight(to_tsvector(english, coalesce(NEW.body,)), C);
 RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER my_docs_search_vector_trigger
BEFORE INSERT OR UPDATE ON my_docs
FOR EACH ROW EXECUTE FUNCTION my_docs_search_vector_update();
```

### 1. Run a test query

Use `plainto_tsquery` for plain phrase matching, or `websearch_to_tsquery` for Google-style syntax ("exact phrase", -exclude, OR). Use whichever matches your application's query input format.

```
SELECT id, title,
 ts_rank_cd(search_vector, query) AS score
```

(continues on next page)

(continued from previous page)

```

FROM my_docs,
 plainto_tsquery(english, connection pooling) query
WHERE search_vector @@ query
ORDER BY score DESC
LIMIT 10;

```

For an AIDB pipeline running alongside a native FTS index, AIDB handles embedding generation and vector search. The FTS trigger and the pipeline operate independently, so new rows are automatically covered by both.

!!!note The trigger pattern above ensures FTS coverage is always current, even for rows that arrive before embeddings are generated. This makes native FTS a reliable fallback for new content. See [Handling embedding service outages](#) for more.

## Next steps

- *Writing hybrid search queries* — Use your FTS index in hybrid queries, including weighted linear fusion.
- *Handling embedding service outages* — Use native FTS as a fallback when the embedding service is unavailable.

## Writing hybrid search queries

The patterns below cover every supported combination of vector search with keyword and relational filters. Complete the relevant setup page before using a pattern that requires native full-text search (FTS):

- *Setting up native PostgreSQL FTS*

### Vector + relational filter

Filter semantic search results by any column in the source table. The filter is applied at the same query level as the vector ORDER BY, allowing iterative index scans to produce exactly LIMIT results.

```

SET hnsw.iterative_scan = relaxed_order;

SELECT s.id, s.description,
 kb.value <=> aidb.kb_query_encode(public.my_kb_vector, summer fashion)::vector_
↪ AS distance
FROM my_kb_vector kb
JOIN my_source_table s ON s.id::TEXT = kb.source_id
WHERE s.category_id = 2 -- relational filter on source table
ORDER BY kb.value <=> aidb.kb_query_encode(public.my_kb_vector, summer_
↪ fashion)::vector
LIMIT 10;

```

### Vector + native PostgreSQL FTS

Use standard PostgreSQL tsvector / ts\_rank as the keyword leg. This example assumes a search\_vector tsvector column on the source table, populated and indexed as described in [Setting up native PostgreSQL FTS](#).

```

SET hnsw.iterative_scan = relaxed_order;

SELECT s.id, s.title,

```

(continues on next page)

(continued from previous page)

```

 kb.value <=> aidb.kb_query_encode(public.pipeline_docs, connection_
↪pooling)::vector AS distance,
 ts_rank_cd(s.search_vector, plainto_tsquery(english, connection pooling))
↪
 AS fts_score
FROM public.pipeline_docs AS kb
JOIN my_docs AS s ON s.id::TEXT = kb.source_id
WHERE s.search_vector @@ plainto_tsquery(english, connection pooling) -- keyword_
↪filter
ORDER BY kb.value <=> aidb.kb_query_encode(public.pipeline_docs, connection_
↪pooling)::vector
LIMIT 10;

```

The WHERE clause acts as a hard keyword filter: only rows that contain the search terms are considered. Results are then ordered by semantic similarity. To surface results that match either signal — not just both — use *Weighted linear fusion* instead.

### Weighted linear fusion

Weighted linear fusion combines normalized vector similarity and full-text scores into a single composite score using two explicit weights — one for semantic relevance, one for keyword relevance — that must sum to 1.0. Unlike RRF, which uses only rank positions, weighted linear fusion respects actual score magnitudes, making it useful when you want to control exactly how much each signal contributes to the final ranking.

### When to use weighted linear fusion versus RRF

	Weighted linear fusion	RRF
Uses actual score values	Yes	No (rank positions only)
Requires score normalization	Yes	No
Tunable per use case	Yes (explicit weights)	Partially (smoothing constant $k$ )
Robust to score scale differences	No (normalize carefully)	Yes (rank-based, scale-independent)
Best for	Workloads where score magnitude carries information	General-purpose merging of heterogeneous result sets

Adjust the weights based on your workload:

Use case	Suggested weights
Concept/meaning is most important	0.7 semantic, 0.3 keyword
Exact terminology matters as much as meaning	0.5 semantic, 0.5 keyword
Keyword precision is critical (for example, code or rule search)	0.3 semantic, 0.7 keyword

### With native PostgreSQL FTS:

Because vector distances and `ts_rank` values use different scales, both legs are normalized to  $[0, 1]$  before combining. Cosine distance (range  $[0, 2]$ ) is converted to similarity using  $(2.0 - \text{distance}) / 2.0$ . `ts_rank_cd` already returns values in roughly  $[0, 1]$ .

```

WITH semantic AS (
 SELECT source_id,
 (2.0 - (value <=> aidb.kb_query_encode(public.pipeline_docs, connection_
→pooling)::vector)) / 2.0
 AS vec_score
 FROM public.pipeline_docs
 ORDER BY value <=> aidb.kb_query_encode(public.pipeline_docs, connection_
→pooling)::vector
 LIMIT 50
),
keyword AS (
 SELECT id::TEXT AS source_id,
 ts_rank_cd(search_vector, plainto_tsquery(english, connection pooling)) AS_
→fts_score
 FROM my_docs
 WHERE search_vector @@ plainto_tsquery(english, connection pooling)
 ORDER BY fts_score DESC
 LIMIT 50
)
SELECT
 COALESCE(s.source_id, k.source_id) AS id,
 0.6 * COALESCE(s.vec_score, 0) + 0.4 * COALESCE(k.fts_score, 0) AS fusion_score
FROM semantic s
FULL OUTER JOIN keyword k ON s.source_id = k.source_id
ORDER BY fusion_score DESC
LIMIT 10;

```

### Worked example: relational predicate filtering

This example shows how to combine semantic vector search with a relational `WHERE` filter to narrow results to a specific category. A pipeline embeds product names into a knowledge base, and queries then retrieve semantically similar results filtered by category.

#### Setting up the source table and pipeline

1. Create the source table:

```

CREATE TABLE products (
 id SERIAL PRIMARY KEY,
 product_name TEXT NOT NULL,
 category TEXT NOT NULL
);

```

1. Populate it with sample data:

```

INSERT INTO products (product_name, category) VALUES
 (Hamburger, meat),
 (Cheeseburger, meat),
 (Fish n Chips, fish),
 (Fries, vegetarian),
 (Burrito, meat),
 (Pizza, meat),

```

(continues on next page)

(continued from previous page)

```
(Sandwich, meat),
(Veggie Burger, vegetarian),
(Kebab, meat);
```

1. Create a pipeline to embed `product_name` into a knowledge base:

```
SELECT aidb.create_pipeline(
 name => food_products,
 source => products,
 source_key_column => id,
 source_data_column => product_name,
 step_1 => KnowledgeBase,
 step_1_options => aidb.knowledge_base_config(
 model => bert,
 data_format => Text
)
);
```

1. Run the pipeline to generate embeddings:

```
SELECT aidb.run_pipeline(food_products);
```

### Standard semantic query (no filter)

This baseline query returns the top semantic matches with no constraints, so you can compare the result set against the filtered query that follows.

```
SELECT * FROM aidb.retrieve_text(public.pipeline_food_products, fast food, topk => 4);
__OUTPUT__
```

key	value	distance	part_ids	pipeline_name	intermediate_steps
8	Veggie Burger	0.9955745765192575	{0}	food_products	[]
6	Pizza	1.0280680840689795	{0}	food_products	[]
4	Fries	1.0630385212203115	{0}	food_products	[]
7	Sandwich	1.0903594734917037	{0}	food_products	[]

### Custom query with relational filter

This query adds a `WHERE` clause on the source table to restrict results to a single category, while keeping the vector `ORDER BY` at the same query level so iterative index scans can satisfy the `LIMIT` efficiently.

```
SET hnsf.iterative_scan = relaxed_order;

SELECT vector.source_id, source.product_name, source.category
FROM public.pipeline_food_products AS vector
JOIN products AS source ON source.id::TEXT = vector.source_id
WHERE source.category = vegetarian
-- relational filter
ORDER BY vector.value <=> aidb.kb_query_encode(public.pipeline_food_products, fast
food)::vector
LIMIT 10;
__OUTPUT__
```

(continues on next page)

(continued from previous page)

source_id	product_name	category
8	Veggie Burger	vegetarian
4	Fries	vegetarian

Only vegetarian items are returned, ordered by semantic relevance to “fast food”.

## Handling embedding service outages

When the embedding service is unavailable or embeddings haven’t been generated for recent rows, hybrid search queries fail or return incomplete results. Implement a three-tier fallback to keep your application returning results in those situations.

## Defining the fallback chain

A recommended degradation chain:

1. **Hybrid** — Vector + full-text search (FTS) — the primary path; requires the embedding service.
2. **FTS-only** — Keyword search using `tsvector` (no embedding service required).
3. **Regex** — `ILIKE` pattern match (last resort, no index required).

## Implementing the fallback

Implement this strategy in application code by catching connection errors from the embedding step and routing to the appropriate fallback, or in SQL using a CASE expression or a wrapper function. The three queries below are the building blocks for each tier:

### Primary: hybrid (vector + native FTS)

```
WITH semantic AS (
 SELECT source_id,
 (2.0 - (value <=> aidb.kb_query_encode(public.pipeline_docs, :query)::vector)) / 2.0 AS vec_score
 FROM public.pipeline_docs
 ORDER BY value <=> aidb.kb_query_encode(public.pipeline_docs, :query)::vector
 LIMIT 50
),
keyword AS (
 SELECT id::TEXT AS source_id,
 ts_rank_cd(search_vector, plainto_tsquery(english, :query)) AS fts_score
 FROM my_docs
 WHERE search_vector @@ plainto_tsquery(english, :query)
 LIMIT 50
)
SELECT COALESCE(s.source_id, k.source_id) AS id,
 0.6 * COALESCE(s.vec_score, 0) + 0.4 * COALESCE(k.fts_score, 0) AS score
FROM semantic s
FULL OUTER JOIN keyword k ON s.source_id = k.source_id
ORDER BY score DESC
LIMIT 10;
```

### Fallback 1: FTS-only (no embedding service)

```

SELECT id, title,
 ts_rank_cd(search_vector, plainto_tsquery(english, :query)) AS score
FROM my_docs
WHERE search_vector @@ plainto_tsquery(english, :query)
ORDER BY score DESC
LIMIT 10;

```

### Fallback 2: regex (no index)

```

SELECT id, title
FROM my_docs
WHERE title ILIKE % || :query || %
 OR body ILIKE % || :query || %
LIMIT 10;

```

The regex fallback performs a full sequential scan and doesn't scale to large tables. Reserve it for emergency use or small datasets, and add a timeout guard in your application to prevent long-running queries.

!!!note The `tsvector` trigger pattern described in [Setting up native PostgreSQL FTS](#) ensures FTS coverage is always current, even for rows that arrive before embeddings are generated. This makes it a reliable fallback for new content.

## Hybrid search reference

AIDB provides helper functions and a metadata view for constructing hybrid search queries. Use these to encode queries against a specific knowledge base, rerank retrieved results, and inspect knowledge base configuration without hardcoding model names or table paths.

### `aidb.kb_query_encode()`

Encodes a text query using the embedding model configured for a specific knowledge base, without needing to look up the model name:

```

SELECT aidb.kb_query_encode(public.my_kb_vector, test query);

```

Use this in place of `aidb.encode_text_query('<model>', '<query>')` when constructing custom vector queries.

### `aidb.rerank_text()`

After retrieving candidates, pass them to a reranking model for a final relevance-ordered result:

```

SELECT aidb.rerank_text(my_reranking_model, query text, ARRAY[result1, result2, ...]);

```

Reranking models such as `nim_reranking` and `hf_tei_reranking` are supported as a separate post-processing step after retrieval.

### `aidb.knowledge_bases` view

The `aidb.knowledge_bases` view (also accessible as `aidb.kbs`) exposes all the metadata needed to construct hybrid queries:

Column	Type	Description
<i>id</i>	integer	Internal identifier
<i>name</i>	text	Knowledge base name
<i>vector_schema</i>	text	Schema of the embeddings table
<i>vector_table</i>	text	Embeddings storage table
<i>model_name</i>	text	Embedding model used by this KB
<i>distance_operator</i>	aidb.distanceoperator	Distance metric enum value
<i>distance_operator_sql</i>	text	The distance operator in SQL-operator syntax
<i>vector_data_column</i>	text	Column storing the embeddings
<i>vector_key_column</i>	text	Column linking embeddings to source rows
<i>vector_index</i>	jsonb	Vector index configuration
<i>pipeline_ids</i>	integer[]	IDs of all pipelines attached to this knowledge base
<i>pipeline_names</i>	text[]	Names of all pipelines attached to this knowledge base

```
SELECT name, vector_schema, vector_table, distance_operator_sql
FROM aidb.knowledge_bases;
```

## 12.4 Examples

The following examples illustrate how to perform bulk knowledge base (KB) embedding using AI pipelines, including the expected outputs for each stage. These examples cover ingesting data from a standard Postgres table and an external S3 bucket volume.

### 12.4.1 Example 1: Bulk embedding from a Postgres table

This workflow is used when your source data is already stored within your database.

#### Step 1: Initialize the AI model

Register a model that supports text encoding operations:

```
SELECT aidb.create_model(model__6545, bert_local);
__OUTPUT__
create_model

model__6545
(1 row)
```

#### Step 2: Prepare the source data

Create a table and insert the records you wish to embed:

```
CREATE TABLE source_table__6545 (
 id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
 content TEXT NOT NULL
);

INSERT INTO source_table__6545 (id, content)
VALUES (43941, Catwalk Women Brown Heels),
 (55018, Lakme 3 in 1 Orchid Aqua Shine Lip Color),
 (19337, United Colors of Benetton Men Stripes Black Jacket);
```

### Step 3: Define the pipeline

Create a pipeline that connects the source table to a knowledge base destination:

```
SELECT aidb.create_pipeline(
 name => pipeline__6545,
 source => source_table__6545,
 source_key_column => id,
 source_data_column => content,
 step_1 => KnowledgeBase,
 step_1_options => aidb.knowledge_base_config(
 model => dummy,
 data_format => Text
)
);
__OUTPUT__
INFO: Created pipeline "pipeline__6545"
INFO: Created knowledge base "public.pipeline_pipeline__6545" for pipeline "pipeline_
↪__6545"

 create_pipeline

(pipeline__6545,Table,public,pipeline_pipeline__6545,source_id,value)
(1 row)
```

### Step 4: Execute and verify

Trigger the processing engine and inspect the generated vector table.

```
SELECT aidb.run_pipeline(pipeline__6545);

- - View the generated embeddings
SELECT * FROM pipeline_pipeline__6545 ORDER BY id;
__OUTPUT__
 id | pipeline_id | source_id | part_ids | value
-----+-----+-----+-----+-----
 1 | 49 | 43941 | {0} | [0.1,0.2,0.3]
 2 | 49 | 55018 | {0} | [0.1,0.2,0.3]
 3 | 49 | 19337 | {0} | [0.1,0.2,0.3]
(3 rows)
```

## 12.4.2 Example 2: Bulk embedding from an S3 bucket (PGFS)

This workflow is used for unstructured data stored in external cloud storage.

### Step 1: Configure external storage and volume

Define the S3 bucket location and map it to an AIDB volume.

```
SELECT pgfs.create_storage_location(
 bucket__6545,
 s3://noah-html-test,
 options => {"region": "eu-central-1", "skip_signature": "true"}
);
```

(continues on next page)

(continued from previous page)

```
SELECT aidb.create_volume(volume__6545, bucket__6545, /, Text);
```

## Step 2: List volume content

The volume is a foreign table — query it directly to verify the files available for processing:

```
SELECT key, size, last_modified FROM volume__6545 ORDER BY key;
```

```
__OUTPUT__
 key | size | last_modified
-----+-----+-----
Bilge Ince.html | 8878 | 2025-02-19 09:58:29+00
all_tags.html | 11170 | 2025-02-19 09:58:29+00
hello_world.html | 108 | 2025-02-19 09:58:30+00
simple.html | 662 | 2025-02-19 09:58:29+00
(4 rows)
```

## Step 3: Create and run the volume pipeline

```
SELECT aidb.create_pipeline(
 name => volume_pipeline__6545,
 source => volume__6545,
 step_1 => KnowledgeBase,
 step_1_options => aidb.knowledge_base_config(
 model => dummy,
 data_format => Text
)
);
```

```
SELECT aidb.run_pipeline(volume_pipeline__6545);
```

## Step 4: Verify embeddings

```
SELECT source_id, part_ids, value FROM pipeline_volume_pipeline__6545 ORDER BY source_
↪id;
```

```
__OUTPUT__
 source_id | part_ids | value
-----+-----+-----
Bilge Ince.html | {0} | [0.1,0.2,0.3]
all_tags.html | {0} | [0.1,0.2,0.3]
hello_world.html | {0} | [0.1,0.2,0.3]
simple.html | {0} | [0.1,0.2,0.3]
(4 rows)
```

## Semantic knowledge bases

A semantic knowledge base (semantic KB) indexes your database's *schema* (tables, views, columns, and their comments) into a searchable vector store, so you or an *Agents reference* can find the right relations and columns by meaning rather than by exact name. It's the foundation for **text-to-SQL**: turning a natural-language question into a correct SQL query against your own schema.

### 13.1 Semantic KB vs. vector knowledge base

A semantic KB and a *Knowledge bases reference* both store embeddings and answer similarity searches, but they index different things and solve different problems.

	Vector knowledge base	Semantic knowledge base
What it embeds	Your <b>data</b> (rows, documents, images)	Your <b>schema</b> (table, view, and column metadata)
What you search	Content, by meaning	Schema structure, by meaning
Created by	<code>aidb.create_pipeline()</code> with a <i>AI pipelines</i>	<code>aidb.create_semantic_kb()</code>
Typical use	Retrieval-augmented generation, semantic search	Natural-language schema discovery, agent tools, text-to-SQL

If you want to search *what's in* your database, use a vector knowledge base. If you want to search *how your database is shaped*, to find the tables and columns a question maps to, use a semantic KB.

### 13.2 What a semantic KB indexes

When a semantic KB crawls its schemas, it embeds one entry per structural element:

- **Tables** contribute their table definition and any table comment.
- **Views** contribute their view definition and any view comment.
- **Columns** contribute each column's definition (name, type, nullability, default) and any column comment.

Each entry has an `entity_type` of `Table`, `View`, or `Column`. Comments matter: a well-commented schema produces a far more useful semantic KB, because the natural-language intent in `COMMENT ON` statements is embedded alongside the structural definition and can be matched directly.

## 13.3 Create and manage a semantic KB

Create a semantic KB over one or more schemas with `aidb.create_semantic_kb()`. It embeds the schema metadata with the model you pass, builds the KB's internal vector table, and crawls the schemas immediately:

```
SELECT aidb.create_semantic_kb(
 name => analytics_kb,
 model => my_embedding_model,
 schemas => ARRAY[sales],
 auto_processing => Live
);
```

`model` is the embedding model and is always required. A semantic KB uses a **single embedding model** for all of its metadata (and for any *Semantic aliases* that are members of it), so search always compares vectors in the same space.

The management functions are:

Function	Purpose
<code>aidb.create_semantic_kb(name, model, schemas, auto_processing, bypass_triggers, vector_index)</code>	Create a KB over one or more schemas. Returns the KB name.
<code>aidb.list_semantic_kbs()</code>	List every KB with its model, schemas, and processing mode.
<code>aidb.refresh_semantic_kb(name)</code>	Re-crawl the KB's schemas from scratch to pick up structural changes.
<code>aidb.update_semantic_kb_auto_processing(name, auto_processing)</code>	Change how the KB keeps itself current.
<code>aidb.semantic_kb_stats(kb_name)</code>	Return entity counts ( <i>total</i> , <i>tables</i> , <i>views</i> , <i>columns</i> ) and the number of <i>pending</i> schema changes.
<code>aidb.delete_semantic_kb(name)</code>	Delete a KB and its indexed metadata.

### 13.3.1 The single-KB shortcut

When exactly one semantic KB exists, you can omit `name/kb_name` from every function, and the call resolves to that KB. An omitted name at creation registers the KB under the fixed name `default_semkb`. Once a second KB exists, calls that omit the name fail with an ambiguity error, so pass the name explicitly from then on.

## 13.4 Keeping a KB current

The `auto_processing` mode controls how a KB reacts to schema changes:

Mode	Behavior
<i>Disabled</i> (default)	No automatic updates. Call <code>aidb.refresh_semantic_kb()</code> when the schema changes.
<i>Live</i>	DDL triggers re-crawl affected relations as soon as the schema changes.
<i>Background</i>	A background worker processes queued DDL events, so re-indexing happens off the write path.

Internally, `Background` mode is driven by AIDB's pipeline machinery, where a `SemanticKB` pipeline step consumes the queued schema-change events. You don't create or manage that pipeline yourself; `aidb.create_semantic_kb()` sets it up.

## 13.5 How you use a semantic KB

- **Directly, in SQL.** Call the *Searching a semantic knowledge base* (`aidb.semantic_kb_search()` and friends) to find the schema entities a question maps to.
- **From an agent.** The KB search functions are registered as native *agent tools*, so an AIDB agent can discover schema and answer questions with *Example: a text-to-SQL agent*.
- **Through saved queries.** Capture recurring questions as *Semantic aliases*: parameterized SQL with a natural-language description that search can find by meaning.

## 13.6 In this section

Page	What it covers
<i>Searching a semantic knowledge base</i>	The schema-search functions: <code>semantic_kb_search</code> , <code>get_metadata</code> , comment search, and filters
<i>Semantic aliases</i>	Named, parameterized SQL queries with vectorized descriptions, enabling reusable, governed text-to-SQL
<i>Example: a text-to-SQL agent</i>	How an AIDB agent uses the KB's search tools to discover schema and generate correct SQL
<i>Examples</i>	A worked end-to-end example over a sample schema

## 13.7 Related documentation

- *Agents reference*, which covers building the in-database agents that call semantic KB search functions as tools.
- *Semantic knowledge base tools*, which lists the native tool catalog entries for the KB functions.
- *Knowledge bases reference*, which covers vector knowledge bases for semantic and hybrid search over your data.

### 13.7.1 Searching a semantic knowledge base

Once a semantic KB has crawled your schemas, you search it with a natural-language query. The KB embeds your query with the same model it used to index the schema, then returns the tables, views, and columns whose metadata is closest to the query. Every search function takes the KB name as `kb_name`, which you can omit when only one KB exists.

**aidb.semantic\_kb\_search()**

`semantic_kb_search()` is the composite entry point: it runs the per-source ranked searches (schema metadata and curated *Semantic aliases*) and fuses them into one ranked list with Reciprocal Rank Fusion (RRF). Use it when you want a single ranked answer to “what in my schema relates to this?”

```
SELECT source_type, entity_type, schema_name, relation_name, column_name, rank
FROM aidb.semantic_kb_search(
 query_text => which customers spent the most,
 kb_name => analytics_kb,
 top_k => 5
);
```

Parameter	Type	Default	Description
<i>query_text</i>	<i>text</i>	—	The natural-language query.
<i>kb_name</i>	<i>text</i>	<i>NULL</i>	The KB to search. Omit to target the single KB when only one exists.
<i>top_k</i>	<i>int</i>	<i>10</i>	Number of ranked rows to return.
<i>sources</i>	<i>text[]</i>	<i>NULL</i>	Sources to search. Active: <i>schema</i> , <i>alias</i> . Omit for all. ( <i>history</i> , <i>relationship</i> are reserved.)
<i>entity_types</i>	<i>text[]</i>	<i>NULL</i>	Entity types to include. Active: <i>Table</i> , <i>View</i> , <i>Column</i> , <i>Alias</i> . Omit for all.
<i>rrf_k</i>	<i>int</i>	<i>60</i>	Rank-fusion damping constant.
<i>min_similarity</i>	<i>double precision</i>	<i>NULL</i>	Optional similarity floor; rows scoring below it are dropped. Omit for no floor.

It returns one row per match:

Column	Description
<i>source_type</i>	Which source produced the row, either <i>schema</i> or <i>alias</i> .
<i>entity_type</i>	<i>Table</i> , <i>View</i> , <i>Column</i> , or <i>Alias</i> .
<i>schema_name</i>	The schema the entity belongs to.
<i>relation_name</i>	The table or view name.
<i>column_name</i>	The column name (empty for table- and view-level matches).
<i>object_ref</i>	A qualified reference to the matched object.
<i>definition</i>	The structural definition that was embedded.
<i>comment</i>	The <i>COMMENT ON</i> text for the entity, if any.
<i>score</i>	The fused RRF score.
<i>rank</i>	The 1-based rank within the fused result set.
<i>components</i>	JSONB detail of the per-source scores that fed the fusion.

Because `semantic_kb_search()` returns both schema entities and matching aliases in one ranked list, it’s the natural first call in a *Example: a text-to-SQL agent* workflow.

## Schema-metadata functions

For narrower needs, four functions search only schema metadata. They share the same optional `kb_name` and a similarity score (cosine similarity, higher is closer), with `min_similarity` as an optional floor and `top_k / offset` for paging.

### `aidb.get_metadata()`

This is the broadest schema search, returning columns, tables, and views together with full metadata.

```
SELECT schema_name, relation_name, column_name, entity_type, similarity
FROM aidb.get_metadata(
 kb_name => analytics_kb,
 query_text => customer email address,
 top_k => 10
);
```

Parameters: `kb_name`, `query_text`, `min_similarity`, `top_k` (default 10), `offset` (default 0). Returns `schema_name`, `relation_name`, `column_name`, `entity_type`, `definition`, `comment`, `definition_vector`, and `similarity`.

### `aidb.get_entity_definitions()`

Matches **tables and views** only.

```
SELECT schema_name, relation_name, entity_type, similarity
FROM aidb.get_entity_definitions(
 kb_name => analytics_kb,
 query_text => orders and line items,
 entity_types => ARRAY[Table, View]
);
```

Parameters: `kb_name`, `query_text`, `min_similarity`, `entity_types` (default `ARRAY['Table', 'View']`), `top_k`, `offset`. Returns `schema_name`, `relation_name`, `entity_type`, `definition`, `comment`, `similarity`.

### `aidb.get_column_definitions()`

Returns just the matching column and table **definitions**, the leanest result, useful as compact grounding for SQL generation.

```
SELECT definition
FROM aidb.get_column_definitions(
 kb_name => analytics_kb,
 query_text => order total amount
);
```

Parameters: `kb_name`, `query_text`, `min_similarity`, `top_k`, `offset`. Returns a single definition column.

### `aidb.search_by_comment()`

Matches only the natural-language `COMMENT` ON text attached to schemas, tables, and columns.

```
SELECT relation_name, column_name, comment, similarity
FROM aidb.search_by_comment(
 kb_name => analytics_kb,
 query_text => order status values
);
```

Parameters: `kb_name`, `query_text`, `min_similarity`, `top_k`, `offset`. Returns `schema_name`, `relation_name`, `column_name`, `entity_type`, `definition`, `comment`, `similarity`.

This is the most direct way to leverage documentation you’ve already written into your schema. For example, a column commented “*order lifecycle state: pending, confirmed, shipped, delivered, cancelled*” surfaces for a query like “*order status values*” even if the column itself is named `st`. Because comment search depends on well-documented objects, keeping `COMMENT ON` statements current is the highest-leverage way to improve a semantic KB’s results.

### Tuning results

- “`min_similarity`” trades recall for precision. Lower it (for example 0.3) when a query returns nothing; raise it when results are too broad.
- “`top_k`” / “`offset`” page through results.
- “`entity_types`” narrows to the kinds you care about, such as `ARRAY['Column']`, when you’re hunting for a specific field.

### Next steps

- Save frequently asked questions as reusable, parameterized queries with *Semantic aliases*.
- Let an agent drive these searches end to end in the *Example: a text-to-SQL agent* workflow.

## 13.7.2 Semantic aliases

A **semantic alias** is a named, parameterized SQL query paired with a natural-language description. The description is embedded, so an alias can be found by meaning — the same way schema metadata is. Aliases are the reusable, governed layer of text-to-SQL: instead of generating fresh SQL for every question, a recurring question resolves to a query you’ve already written and reviewed.

Each alias is a member of *every* semantic KB that owns the schema its SQL reads — not just one. AIDB infers the owning KB(s) from the query and embeds the alias’s description once per KB, using that KB’s model, so each embedding stays comparable to that KB’s schema vectors. A schema owned by two KBs yields two embeddings, and the alias is then findable — with `aidb.semantic_kb_search()` — from either KB.

!!!note Semantic aliases are managed and run with the `aidb` SQL functions below. Unlike the KB *search* functions, alias functions are **not** exposed as *agent tools* — by product decision, an agent discovers and reads schema, but doesn’t create or execute aliases itself.

### Creating an alias

`aidb.create_semantic_alias()` takes a unique name, a description to embed, the SQL query text, and its parameter definitions. Placeholders in the query use `${name}` syntax. The query must be a single read-only `SELECT`.

```
SELECT aidb.create_semantic_alias(
 name => monthly_revenue,
 description => Total revenue for a given month and year,
 query_text => ${
 SELECT SUM(amount) AS total
```

(continues on next page)

(continued from previous page)

```

FROM sales.orders
WHERE EXTRACT(MONTH FROM order_date) = ${month}
 AND EXTRACT(YEAR FROM order_date) = ${year}
$$,
params => [
 {"name": "month", "param_type": "integer", "description": "Month number (1-12)"
→ },
 {"name": "year", "param_type": "integer", "description": "Four-digit year"}
]
);

```

Parameter	Type	Default	Description
<i>name</i>	<i>text</i>	—	Unique alias name.
<i>description</i>	<i>text</i>	—	Human-readable description — this is what gets embedded for search.
<i>query_text</i>	<i>text</i>	—	A single read-only <i>SELECT</i> , with <i>\${name}</i> placeholders for parameters.
<i>params</i>	<i>jsonb</i>	<i>NULL</i>	Array of parameter definitions (see below).
<i>kb_name</i>	<i>text</i>	<i>NULL</i>	Optional; narrows the alias to just this KB. Omit to embed it for every KB that owns the query's schema(s).

Each entry in the `params` JSONB array has:

Field	Required	Description
<i>name</i>	Yes	Matches a <i>\${name}</i> placeholder in <i>query_text</i> .
<i>param_type</i>	Yes	PostgreSQL type: <i>text</i> , <i>integer</i> , <i>numeric</i> , <i>date</i> , and so on.
<i>description</i>	No	Human-readable description of the parameter.
<i>enum_values</i>	No	Allowed values, when the parameter is constrained to a set.

If the query's schema maps to more than one KB, the alias is embedded once for each of them — creation never fails on ambiguity; pass `kb_name` only to narrow it to a single KB. A schema not yet owned by any KB leaves the alias unembedded, until a KB covering it is created or adopts it later.

## Finding an alias

`aidb.search_semantic_aliases()` searches aliases by natural-language query. Without `kb_name`, it searches `default_semkb` — the name a KB gets when created without an explicit one — if that KB exists, or every KB with stored alias embeddings otherwise; the query is embedded once per distinct model among those KBs, so the search stays dimension-safe, and results are de-duplicated to the best match per alias.

```

SELECT name, query_text, similarity
FROM aidb.search_semantic_aliases(

```

(continues on next page)

(continued from previous page)

```

 query_text => how much money did we make last month,
 kb_name => analytics_kb,
 top_k => 5
);

```

Parameter	Type	Default	Description
<i>query_text</i>	<i>text</i>	—	Natural-language query.
<i>min_similarity</i>	<i>double precision</i>	<i>NULL</i>	Optional similarity floor.
<i>top_k</i>	<i>int</i>	<i>10</i>	Maximum results.
<i>offset</i>	<i>int</i>	<i>0</i>	Paging offset.
<i>kb_name</i>	<i>text</i>	<i>NULL</i>	Optional; narrows the search to this KB. Omit to resolve <i>default_semkb</i> , or all KBs with alias embeddings, per the precedence above.

It returns *name*, *query\_text*, and *similarity*. Aliases also appear in *aidb.semantic\_kb\_search()* results with *source\_type* = 'alias', so a single composite search finds both schema and aliases at once.

## Running an alias

*aidb.execute\_semantic\_alias()* runs an alias by name, substituting its parameters. It returns a set of *result* rows, each a JSONB object:

```

SELECT result
FROM aidb.execute_semantic_alias(
 alias_name => monthly_revenue,
 args => {"month": 3, "year": 2025}
);

```

Parameter	Type	Default	Description
<i>alias_name</i>	<i>text</i>	—	Alias to run.
<i>args</i>	<i>jsonb</i>	<i>NULL</i>	Object of parameter values keyed by name.
<i>execute_role</i>	<i>text</i>	<i>NULL</i>	PostgreSQL role to run the query as — requires the appropriate <i>SET ROLE</i> grants.

Use *execute\_role* to run an alias under a least-privilege reporting role rather than the connecting user. Because every alias is validated as a single read-only *SELECT* at creation and re-validated at execution, an alias can't be used to run writes.

## Managing aliases

Function	Purpose
<i>aidb.get_semantic_aliases()</i>	List all aliases with <i>name</i> , <i>description</i> , <i>query_text</i> , and <i>param_count</i> .

continues on next page

Table 11 – continued from previous page

Function	Purpose
<i>aidb.get_semantic_alias(alias_name)</i>	Get one alias's <i>name</i> , <i>description</i> , <i>query_text</i> , and <i>params</i> .
<i>aidb.update_semantic_alias(name, description, query_text, params, kb_name)</i>	Update an alias; changing <i>query_text</i> or <i>kb_name</i> re-infers its owning KB(s) and reconciles embeddings — adding KBs it newly belongs to and dropping ones it no longer does. Changing <i>description</i> re-embeds it in every KB it's already in.
<i>aidb.delete_semantic_alias(name)</i>	Delete an alias by name.

Deleting a KB removes only that KB's embedding of each alias — the alias definition, and its embeddings in any other owning KBs, survive. A later KB covering the same schema adopts the alias, adding a fresh embedding for it, even if the alias is already embedded for other KBs.

### Next steps

- See how an agent combines schema search and aliases into an end-to-end *Example: a text-to-SQL agent* workflow.
- Walk through aliases and search together in the *Examples* .

### 13.7.3 Text-to-SQL

Text-to-SQL turns a natural-language question, such as “*Which customers spent the most last quarter?*”, into a correct query against your own schema. A semantic KB makes this reliable: rather than guessing at table and column names, the question is first grounded in the schema the KB indexed.

Two complementary paths build on the *Searching a semantic knowledge base* :

- **Agent-driven.** An *Agents reference* calls the KB's search functions as tools to discover the relevant schema, then generates and runs SQL. Best for open-ended, conversational analytics.
- **Application-driven.** Your application calls *Finding an alias* and *Running an alias* to resolve a question to a reviewed, parameterized query. Best for recurring questions that need deterministic, governed answers.

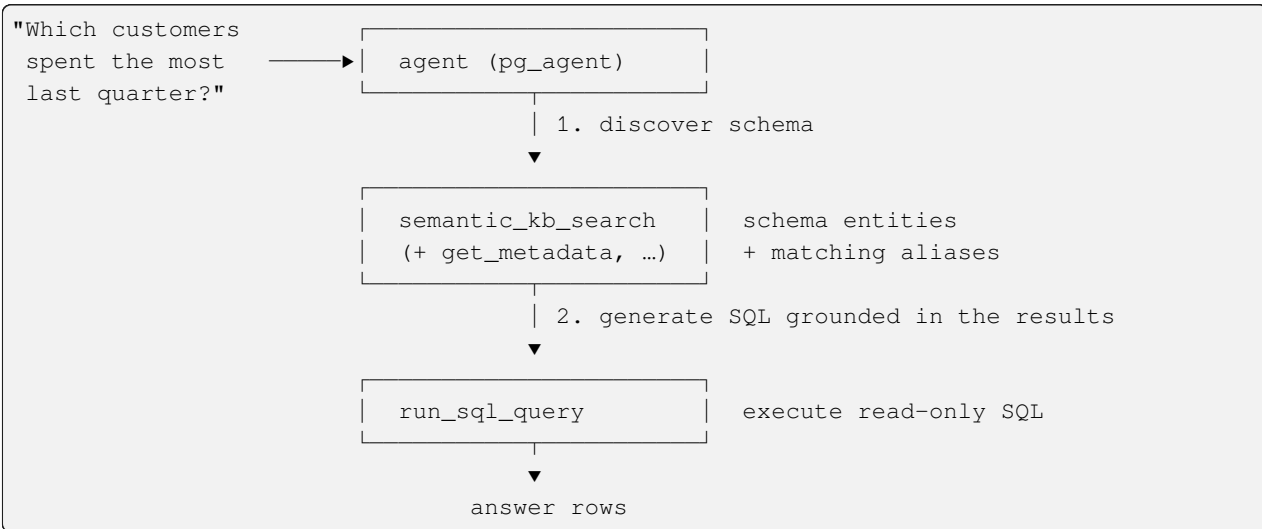
### Agents as the text-to-SQL engine

The semantic KB search functions are registered as native *agent tools* . An agent given these tools can, on its own, find the tables and columns a question maps to and then write SQL grounded in them. The tools it draws on:

Tool	Role in text-to-SQL
<i>AI pipelines</i>	One ranked list of the schema entities (and any matching aliases) relevant to the question.
<i>AI pipelines / AI pipelines / AI pipelines</i>	Narrower schema lookups when the agent needs only tables, only columns, or lean definitions.
<i>AI pipelines</i>	Finds entities through the intent captured in <i>COMMENT ON</i> text.
<i>run_sql_query</i>	Runs the read-only SQL the agent generates and returns the rows.

All the search tools are read-only, so an agent can explore schema freely without any risk of a write.

## The workflow



1. **Discover.** The agent calls `semantic_kb_search` (and the narrower functions as needed) to retrieve the exact tables, columns, and comments the question maps to, plus any *Semantic aliases* that already match, which appear in the results with `source_type = 'alias'`.
2. **Generate, grounded.** The agent writes SQL against the retrieved definitions. Generating against real schema context (actual column names, types, and comments) is far more accurate than generating against table names alone.
3. **Run.** The agent executes the SQL with `run_sql_query` and returns the rows.

To build such an agent, give it the semantic KB tools when you create it. See *Managing agents* and the *tool catalog*.

## When to reach for aliases instead

Aliases are the deterministic counterpart to agent generation. A *Semantic aliases* is a reviewed, parameterized `SELECT`, so it returns dependable results with no per-request generation to verify. Use aliases when:

- The same question recurs and deserves one canonical query.
- You need governed execution, where an alias runs as a single read-only `SELECT` and can run under a least-privilege role via `execute_role`.

Aliases surface inside `semantic_kb_search` results, so an agent can *see* that a curated query exists for a question. Executing an alias, though, is done directly through `aidb.execute_semantic_alias()`. The alias functions are intentionally not agent tools.

## How this compares to raw generation

- **Grounded, not guessed.** The model sees the actual definitions and comments of the relevant relations, so it references columns that exist, with the right types.
- **Trusted answers accrue.** Every question you capture as an alias is one that no longer needs generation, shrinking the surface where a model might generate wrong SQL.
- **Read-only by construction.** Schema-search tools and aliases are all read-only, so the discovery and answer paths can't mutate data.

## Next steps

- Walk through the whole flow on a sample schema in the *Examples* .
- Review the search functions in *Searching a semantic knowledge base* and the alias functions in *Semantic aliases*

## 13.7.4 Example

This example walks through a complete semantic knowledge base workflow over a small `sales` schema: documenting the schema, indexing it, searching it by meaning, and capturing a recurring question as a reusable alias.

### Step 1: Register an embedding model

A semantic KB embeds schema metadata with a text embedding model. Register one first:

```
SELECT aidb.create_model(sales_embeddings, bert_local);
__OUTPUT__
 create_model

sales_embeddings
(1 row)
```

### Step 2: Create and document the schema

Comments are embedded alongside the structural metadata, so documenting the schema directly improves search quality.

```
CREATE SCHEMA sales;

CREATE TABLE sales.customers (
 id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
 email TEXT NOT NULL,
 region TEXT
);

CREATE TABLE sales.orders (
 id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
 customer_id INT NOT NULL REFERENCES sales.customers(id),
 amount NUMERIC(12,2) NOT NULL,
 status TEXT NOT NULL,
 order_date DATE NOT NULL
);

COMMENT ON TABLE sales.orders IS One row per customer order, with its
↳monetary amount and lifecycle status.;
COMMENT ON COLUMN sales.orders.amount IS Order total in USD.;
COMMENT ON COLUMN sales.orders.status IS Order lifecycle state: pending, confirmed,
↳shipped, delivered, cancelled.;
COMMENT ON COLUMN sales.customers.email IS Customer contact email address.;
```

### Step 3: Create the semantic KB

Create the KB over the `sales` schema. `model` is required; `Live` auto-processing keeps the index current as the schema changes.

```
SELECT aidb.create_semantic_kb(
 name => analytics_kb,
 model => sales_embeddings,
 schemas => ARRAY[sales],
 auto_processing => Live
);
__OUTPUT__
create_semantic_kb

analytics_kb
(1 row)
```

### Step 4: Check what was indexed

`aidb.semantic_kb_stats()` reports entity counts and any pending schema changes:

```
SELECT * FROM aidb.semantic_kb_stats(kb_name => analytics_kb);
__OUTPUT__
total | tables | views | columns | pending
-----+-----+-----+-----+-----
10 | 2 | 0 | 8 | 0
(1 row)
```

### Step 5: Search by meaning

Find the schema entities that map to a concept, even when names don't match the words used. `semantic_kb_search()` returns schema entities and any matching aliases in one ranked list:

```
SELECT source_type, entity_type, schema_name, relation_name, column_name, rank
FROM aidb.semantic_kb_search(
 query_text => customer email address,
 kb_name => analytics_kb,
 top_k => 3
);
__OUTPUT__
source_type | entity_type | schema_name | relation_name | column_name | rank
-----+-----+-----+-----+-----+-----
schema | Column | sales | customers | email | 1
schema | Table | sales | customers | | 2
schema | Column | sales | orders | customer_id | 3
(3 rows)
```

Comment search finds entities through the intent you documented, not just their names:

```
SELECT relation_name, column_name, comment
FROM aidb.search_by_comment(
 query_text => order status values,
 kb_name => analytics_kb,
 top_k => 1
```

(continues on next page)

(continued from previous page)

```
);
__OUTPUT__
relation_name | column_name | comment
-----+-----
orders | status | Order lifecycle state: pending, confirmed, shipped,
delivered...
(1 row)
```

### Step 6: Save a recurring question as an alias

The question “how much revenue did we make in a given month?” recurs, so capture it as a *Semantic aliases* rather than regenerating SQL each time. AIDB infers that the alias is a member of `analytics_kb`, the KB that owns the sales schema the query reads:

```
SELECT aidb.create_semantic_alias(
 name => monthly_revenue,
 description => Total revenue for a given month and year,
 query_text => $$
 SELECT SUM(amount) AS total
 FROM sales.orders
 WHERE EXTRACT(MONTH FROM order_date) = ${month}
 AND EXTRACT(YEAR FROM order_date) = ${year}
 $$,
 params => [
 { "name": "month", "param_type": "integer", "description": "Month number (1-12)" },
 { "name": "year", "param_type": "integer", "description": "Four-digit year" }
]
);
__OUTPUT__
create_semantic_alias

monthly_revenue
(1 row)
```

### Step 7: Find and run the alias

Search resolves a natural-language question to the alias:

```
SELECT name, similarity
FROM aidb.search_semantic_aliases(
 query_text => how much money did we make last month,
 kb_name => analytics_kb,
 top_k => 1
);
__OUTPUT__
name | similarity
-----+-----
monthly_revenue | 0.83
(1 row)
```

Run it with parameters:

```
SELECT result
FROM aadb.execute_semantic_alias(
 alias_name => monthly_revenue,
 args => {"month": 3, "year": 2025}
);
__OUTPUT__
 result

{"total": 48250.00}
(1 row)
```

### Using it from an agent

Everything above is also available to an *Agents reference* . Give an agent the semantic KB *search tools* and `run_sql_query` , and it can answer open-ended questions itself: it calls `semantic_kb_search` to discover the relevant tables and columns, generates SQL grounded in those definitions, and runs it — the *Example: a text-to-SQL agent* workflow.

### What you built

- A semantic KB (`analytics_kb` ) indexing a documented `sales` schema, kept current in `Live` mode.
- Natural-language schema search that finds relations and columns by meaning.
- A reusable, parameterized alias for a recurring question, findable by meaning and runnable with parameters.

# CHAPTER 14

## SQL functions

AIDB functions are available as standalone SQL functions that you can call directly in queries for one-off transformations, testing and exploration, or to build custom workflows outside of the pipeline framework.

Each page here focuses on usage context and examples. For terse parameter tables and return types, see the API reference:

- Transformation functions (`chunk_text`, `parse_html`, `parse_pdf`, `perform_ocr`, `summarize_text`) are in [Functions reference](#).
- Inference functions (`encode_text`, `generate_text`, `rerank_text`) are in [Reference → Models](#).

Page	Functions	What it covers
<a href="#">Text chunking</a>	<code>aidb.chunk_text()</code>	Splitting text into segments for LLM context windows
<a href="#">Data parsing</a>	<code>aidb.parse_html()</code> , <code>aidb.parse_pdf()</code>	Extracting readable text from HTML and PDF inputs
<a href="#">Performing OCR</a>	<code>aidb.perform_ocr()</code>	Extracting text from images using an OCR-capable model
<a href="#">Summarization</a> <a href="#">cook-book</a>	<code>aidb.summarize_text()</code> , <code>aidb.summarize_text_aggregate()</code>	Generating and aggregating AI-powered text summaries
<a href="#">AI inference functions</a>	<code>aidb.encode_text()</code> , <code>aidb.generate_text()</code> , and variants	Embedding text and images into vectors and generating text completions
<a href="#">Text reranking</a>	<code>aidb.rerank_text()</code>	Scoring and reordering candidate passages by relevance

!!!note To use these operations as steps inside a pipeline, see [Pipeline steps](#).

## 14.1 Text chunking

Use the `aidb.chunk_text()` function to intelligently divide long strings into smaller, semantically coherent segments. This process is essential for staying within the context window limits of Large Language Models (LLMs) and optimizing vector storage.

```
SELECT * FROM aidb.chunk_text (
 input => This is a significantly longer text example that might require splitting
 into smaller chunks. The purpose of this function is to partition text data into
 segments of a specified maximum length, for example, this sentence 145 is
 characters. This enables processing or storage of data in manageable parts.,
 options => {"desired_length": 120, "max_length": 150}
);
```

__OUTPUT__	
part_id	chunk
0	This is a significantly longer text example that might require splitting
1	The purpose of this function is to partition text data into segments of a
2	This enables processing or storage of data in manageable parts.

(3 rows)

### 14.1.1 Configuration options

- `desired_length` (required): The target size for each segment. The splitter attempts to reach this size, while preserving semantic boundaries. If `max_length` is omitted, this becomes a strict upper limit.
- `max_length` (optional): The upper bound for chunk size. If specified, the function will try to generate chunks close to `desired_length` but may extend up to `max_length` to preserve larger semantic units (like full sentences or paragraphs). Chunks will exceed `desired_length` only when it's necessary to avoid cutting across meaningful boundaries. The unit depends on the `strategy` used.
- Specifying `desired_length = max_length` results in fixed-size chunks (for example, when filling a context window exactly for embeddings).
- Use a larger `max_length` if you want to stay within a soft limit but allow some flexibility to preserve higher semantic units, common in RAG, summarization, or Q&A applications.
- `overlap_length` (optional): The amount of content to overlap between consecutive chunks. This helps preserve context across chunk boundaries by duplicating a portion of the content from the end of one chunk to the beginning of the next chunk. Defaults to 0 (no overlap). The unit depends on the `strategy` used.
- `strategy` (optional): The chunking strategy to use. Can be either "chars" (default) for character-based chunking that splits strictly by character count, or "words" for word-based chunking that preserves word boundaries. This determines the unit for `desired_length`, `max_length`, and `overlap_length`: characters for "chars", words for "words".

#### Algorithm summary

- Text is split using a hierarchy of semantic boundaries: characters, graphemes, words, sentences, and increasingly long newline sequences (for example, paragraphs).
- The splitter attempts to form the largest semantically valid chunk that fits within the specified size range.

- Chunks may be returned that are shorter than `desired_length` if adding the next semantic unit would exceed `max_length`.

### 💡 Tip

This operation transforms the shape of the data, automatically unnesting collections by introducing a *part\_id* column. See [Creating pipelines](#) for more details on how pipelines handle multi-part output.

## 14.2 Data parsing

EDB Postgres AI provides built-in functions to parse and extract text from various data formats, including HTML and PDF. These functions are designed to handle the complexities of different formats while providing clean, structured text output that can be easily processed in subsequent pipeline steps.

### 14.2.1 Parse HTML

The `aidb.parse_html()` function extracts readable text from HTML strings, stripping away tags while preserving the logical structure of the content.

```
SELECT * FROM aidb.parse_html(
 html =>
 <h1>Hello, world!</h1>
 <p>This is my first web page.</p>
 <p>
 It contains some bold text, some italic test,
 and a link.
 </p>

 List item
 List item
 List item
 ,
 options => {"method": "StructuredPlaintext"} -- Default
);
```

\_\_OUTPUT\_\_

parse_html	
Hello, world!	+
This <b>is</b> my <b>first</b> web page.	+
It <b>contains some</b> bold text, <b>some</b> italic test, <b>and</b> a link.	+
Postgres Logo Image	+
List item	+
List item	+
List item	+

(continues on next page)

(continued from previous page)

(1 row)

- The method determines how the HTML is parsed:
  - `StructuredPlaintext` (default) — Uses an algorithmic approach to extract raw text from HTML, ideal for simple data ingestion.
  - `StructuredMarkdown` — Converts HTML into Markdown-like text. This retains important syntactical context (like headers and lists), which can help LLMs better understand the documents's original hierarchy.

## 14.2.2 Parse PDF

The `aidb.parse_pdf()` function extracts text from binary PDF data. Because PDFs are often structurally complex, this function offers robust error handling.

```
SELECT * FROM aidb.parse_pdf(
 bytes => decode(255044462d312e340a25b89a929d0a312030206f626a3c3c2f547970652f436174616c6f672f5061676573203320
 hex),
 options => {"method": "Structured", "allow_partial_parsing": true} -- Default
);
```

part_id	text
0	Hello World!+

(1 row)

- The method determines how the PDF is parsed:
  - `Structured` method (default): Relies on the standard PDF specification to identify text blocks. Structural PDF parsing that relies on the PDF specification can struggle with PDFs that are not spec-compliant, especially if they include complex or proprietary encodings and different fonts.
  - `allow_partial_parsing`: A safety flag that tells the system whether to continue to parse PDFs when the parser encounters errors on one or more pages. Setting this to `true` (default) ensures you get as much data as possible from non-compliant or complex files.
- Page mapping: The resulting `part_id` column maps directly to the specific page index from which the text was extracted, allowing you to maintain the original document reference in your AI pipeline.

### Tip

Both parsing operations perform unnesting. This means they transform a single input (like a multi-page PDF) into multiple rows of data, automatically generating a `part_id` to keep the segments in the correct order.

## 14.3 Performing OCR

Use the `aidb.perform_ocr()` function to extract text from images. It takes raw image bytes and a registered OCR model name as input, and returns a table with a `part_id` column (identifying each text block) and a `text` column (the extracted string). Because the function uses unnesting, a single image may produce multiple rows — one per text





(continued from previous page)

```

→p8mGz//xAAUEQEAAAAAAAAAAAAAAAAAAACg/9oACAEC AQE/ABK//8QAFBEBAAAAAAAAAAAAAAAAAAoP/
→aAAgBAwEBPwASv//Z, base64),
 options => {"model": "my_paddle_ocr_model"}
);

__OUTPUT__
create_model

my_paddle_ocr_model
(1 row)

part_id | text
-----+-----
 0 | Tesseract sample
(1 row)

```

## 14.4 Text summarization

`aidb.summarize_text()` condenses a single text string into a concise summary using a registered language model. `aidb.summarize_text_aggregate()` is the SQL aggregate form — use it with `GROUP BY` to summarize text across multiple rows, returning one summary per group.

For parameter tables and return types, see [Reference → Functions](#).

### 14.4.1 `aidb.summarize_text()`

**Step 1: Register a model**

```
SELECT aidb.create_model(my_t5_model, t5_local);
```

**Step 2: Summarize a text string**

## 14.5 Text summarization

```

Control model behavior at runtime using `inference_config`:

```

``` sql SELECT * FROM aidb.summarize_text( input => 'Long article text...', options => aidb.summarize_text_config( 'my_llm', NULL, 'Create a brief summary', NULL, NULL, aidb.inference_config( temperature => 0.3, max_tokens => 50, seed => 42 ) )::json );

```

## 14.6 Text summarization

- Fails at call time `SELECT * FROM aidb.summarize_text( input => 'Hello world', options => aidb.summarize_text_config(model => 'bert_model')::json );`

**OUTPUT ERROR:** The requested adapter is not supported by the model provider: bert\_local ```

## 14.7 Text summarization

<i>The requested adapter is not supported by the model provider: X</i>	Embedding-only provider used for summarization	Use <i>completions</i> , <i>openai completions</i> , <i>t5_local</i> , or another language provider
<i>Invalid parameters: 'desired_length' must be greater than 0</i>	Invalid chunk config	Use a positive integer for <i>desired_length</i>
<i>Invalid parameters: 'max_length' must be greater than or equal to 'desired_length'</i>	<i>max_length</i> is smaller than <i>desired_length</i>	Set <i>max_length</i> $\geq$ <i>desired_length</i>
<i>Invalid parameters: 'overlap_length' must be less than 'desired_length'</i>	Overlap too large	Reduce <i>overlap_length</i> (10–20% of <i>desired_length</i> is typical)
<i>Text summarization failed to reduce size</i>	<i>reduce</i> strategy produced output longer than input	Increase chunk size (use $\geq 50$ words or $\geq 800$ chars)

### 14.7.1 Summarization use cases

`aidb.summarize_text_aggregate()` turns text summarization into a native SQL aggregate — the same `GROUP BY` semantics every analyst already knows. This page covers the business value and enterprise use cases.

#### The problem

Enterprises store massive volumes of unstructured text inside PostgreSQL — customer feedback, support tickets, sales notes, compliance logs, clinical narratives. Extracting insight from this data traditionally requires a fragile pipeline: extract from the database, transform and batch it, send to an external LLM API, parse the response, then load results back. Every step introduces latency, failure modes, security exposure, and engineering cost.

Analysts who know SQL are locked out entirely — they need a data engineer and an ML engineer just to answer a question like “what are customers in the Northeast complaining about?”

#### What AIDB does differently

AIDB brings the LLM to the data instead of moving the data to the LLM. Summarization becomes a native SQL aggregate:

```
SELECT region,
 aidb.summarize_text_aggregate(
 customer_feedback,
 aidb.summarize_text_config(gpt-4o)::json
) AS feedback_summary
FROM support_tickets
WHERE created_at > NOW() - INTERVAL 7 days
GROUP BY region;
```

One query. No pipeline. No Python. No external orchestration. Each row contains an LLM-generated summary of all feedback for that region — returned directly to the SQL client.

The same data, a different question, a different prompt:

```

SELECT region,
 aidb.summarize_text_aggregate(
 customer_feedback,
 aidb.summarize_text_config(
 gpt-4o,
 prompt => Identify the top 3 product issues and suggest fixes
)::json
) AS issues
FROM support_tickets
WHERE created_at > NOW() - INTERVAL 7 days
GROUP BY region;

```

Same function. One string changed. Entirely new analysis — no code deployed, no pipeline modified.

## Business value

**Data never leaves the database** — In regulated industries (healthcare, finance, government), moving sensitive data to an external API can require months of compliance review. AIDB keeps all data inside PostgreSQL. The LLM endpoint can be on-premise, in a private VPC, or running locally.

**SQL is the universal interface** — There are roughly 10 million SQL-literate professionals worldwide. AIDB turns every analyst, DBA, and BI developer into an AI-powered analyst without new tools, notebooks, or API keys to manage.

**Eliminates an entire category of infrastructure** — No Airflow DAGs, Lambda functions, SQS queues, or Kubernetes jobs. The database is the compute layer.

**Real-time, ad-hoc intelligence** — Traditional NLP pipelines are batch-oriented. AIDB enables ad-hoc queries against live data. An executive can ask a new question at 3 PM and have the answer by 3:01 PM.

**Scales with PostgreSQL** — Leverages PostgreSQL's mature query planner, parallel workers, connection pooling, and role-based access control. Enterprises already invested in PostgreSQL infrastructure get AI capabilities at zero marginal infrastructure cost.

## Enterprise use cases

**Retail and e-commerce** — GROUP BY product category across millions of reviews and feedback entries. Get per-category summaries of preferences, complaints, and emerging trends — updated in real time as new reviews arrive.

**Financial services** — GROUP BY risk segment across loan officer notes, borrower communications, and underwriting comments. Surface patterns that quantitative models miss — narrative signals in free-text fields that precede defaults.

**Healthcare and life sciences** — GROUP BY facility, department, or condition across patient feedback, clinical notes, and discharge summaries. Identify care quality variations without exposing PHI to external services.

**Legal and compliance** — GROUP BY jurisdiction or regulation type across thousands of compliance documents, audit findings, and regulatory correspondence. Reduce weeks of manual review to a single query.

**Sales and customer success** — GROUP BY sales rep, region, or deal stage across CRM notes, call transcripts, and email summaries. Extract win/loss patterns and coaching opportunities from the unstructured text in every CRM.

**Manufacturing and supply chain** — GROUP BY supplier or component across procurement correspondence, quality reports, and shipment notes. Detect supply chain risks from narrative signals before they show up in delivery metrics.

**HR and talent** — GROUP BY department, tenure band, or role level across exit interview transcripts and employee survey responses. Surface the real reasons people leave.

## How it works

AIDB implements a true PostgreSQL aggregate using the database's native state-accumulation framework:

- **State function** — accumulates text from each row in a group, maintaining per-group state in memory
- **Finalize function** — sends the accumulated text to the configured LLM and returns the result
- **Chunking engine** — automatically splits large text groups into manageable chunks (configurable by word count or character count, with overlap for context preservation)
- **Reduction strategies** — `append` concatenates chunk summaries; `reduce` iteratively re-summarizes until the output converges — a true map-reduce pattern running inside the database
- **Custom prompts** — each query can specify its own instruction: summarize, translate, extract sentiment, categorize, identify trends — making the same function infinitely flexible
- **Model-agnostic** — works with any OpenAI-compatible endpoint (GPT-4o, Llama, Mistral, on-prem models) configurable per query

## The wow moment

Start with a table of customer preference data — 126 rows across 20 customers and 5 product categories. The raw data is a wall of unstructured text.

Run this query:

```
SELECT name,
 aidb.summarize_text_aggregate(
 preferences,
 aidb.summarize_text_config(gpt-4o)::json
) AS style_profile
FROM customer_preferences
GROUP BY name;
```

In seconds, 126 rows become 20 rows — each a rich, coherent style profile synthesized from every data point for that customer. No pipeline. No engineering team. No data left the database.

Now change the prompt:

```
SELECT category,
 aidb.summarize_text_aggregate(
 preferences,
 aidb.summarize_text_config(
 gpt-4o,
 prompt => Identify the 3 strongest purchasing signals and recommend_
 ↪inventory actions
)::json
) AS buying_signals
FROM customer_preferences
GROUP BY category;
```

Same data. Same function. Different question. Different insight. The analyst changed one string and got an entirely new analysis — no code deployed, no pipeline modified, no ticket filed.

That is the moment it clicks: the entire organization's unstructured data becomes queryable with natural language, using the tool analysts already know.

## Example with chunking and custom prompt

For large text groups, combine chunking with a targeted prompt:

```
SELECT product_category,
 aidb.summarize_text_aggregate(
 review_text,
 aidb.summarize_text_config(
 local-llama,
 aidb.chunk_text_config(120, 120, 15, words),
 Identify the top 3 quality issues and suggest improvements,
 reduce
)::json
) AS quality_analysis
FROM product_reviews
GROUP BY product_category;
```

## Competitive position

Capability	Traditional ETL + LLM API	AIDB in-database aggregate
Data movement	Required (security risk)	None (data stays in place)
User interface	Python / custom app	SQL (universal)
Infrastructure	Airflow + queues + compute	PostgreSQL (existing)
Time to first insight	Days to weeks	Minutes
Ad-hoc queries	Requires new pipeline	Change the SQL
Access control	Separate system	PostgreSQL RBAC / RLS
Audit trail	Must be built	PostgreSQL native logging
Compliance	Requires separate review	Inherits DB compliance posture
Model flexibility	Locked to provider	Any OpenAI-compatible endpoint

## 14.7.2 Summarization cookbook

Real-world recipes for using `aidb.summarize_text_aggregate()` beyond basic summarization. Each recipe is a self-contained pattern you can adapt to your data.

All examples use a sales dataset with tables: `sales_notes (note_text, note_id, order_id)`, `sales_orders (order_id, status, product_id, customer_id)`, `products (product_id, product_name)`, `customers (company_name)`.

### Recipes

#### 1. Summarize by category

**When to use:** You want a summary of text grouped by any dimension — product, customer, region, time period.

```
SELECT p.product_name,
 count(sn.note_id) AS note_count,
 aidb.summarize_text_aggregate(
 sn.note_text,
 aidb.summarize_text_config(my-llm)::json
) AS summary
```

(continues on next page)

(continued from previous page)

```

FROM sales_notes sn
JOIN sales_orders so ON sn.order_id = so.order_id
JOIN products p ON so.product_id = p.product_id
GROUP BY p.product_name
ORDER BY note_count DESC;

```

- Without chunking, all text in each group is sent as a single LLM call.
- For groups with fewer than 500 notes (avg ~250 chars each), no chunking is fastest.
- For larger groups, add chunking to avoid context window overflow (see Recipe 7).

## 2. Deal loss analysis

**When to use:** You want to understand why deals are being lost, with actionable reasons.

```

SELECT p.product_name,
 aidb.summarize_text_aggregate(
 sn.note_text,
 aidb.summarize_text_config(
 my-llm,
 prompt => Analyze these sales notes and answer: Why are we losing
↳deals?
 What are the top reasons for failure?
 Provide specific, actionable insights.
)::json
) AS loss_analysis
FROM sales_notes sn
JOIN sales_orders so ON sn.order_id = so.order_id
JOIN products p ON so.product_id = p.product_id
WHERE so.status IN ('lost', 'closed_lost')
GROUP BY p.product_name;

```

- Filter to lost deals only (WHERE status IN ('lost', 'closed\_lost')) so the LLM focuses on failure patterns.
- The prompt should be specific: “top reasons” gives better results than “analyze”.
- For more constrained output, see [Recipe 8](#).

## 3. Sentiment analysis

**When to use:** You want to gauge overall sentiment (positive/negative/mixed) from grouped text.

```

-- Compare sentiment between won and lost deals
SELECT p.product_name,
 so.status AS deal_outcome,
 aidb.summarize_text_aggregate(
 sn.note_text,
 aidb.summarize_text_config(
 my-llm,
 prompt => Analyze the overall sentiment of these entries. Provide:
 1) Overall sentiment (positive/negative/neutral/mixed),
 2) Key positive themes,

```

(continues on next page)

(continued from previous page)

```

 3) Key negative themes,
 4) Notable concerns if any.

)::json
) AS sentiment
FROM sales_notes sn
JOIN sales_orders so ON sn.order_id = so.order_id
JOIN products p ON so.product_id = p.product_id
WHERE so.status IN (won, lost)
GROUP BY p.product_name, so.status
ORDER BY p.product_name, so.status;

```

- Comparing won vs lost sentiment side-by-side reveals what differentiates successful deals.
- The numbered format in the prompt guides structured output, though LLM adherence varies at scale (see *Known limitations*).

#### 4. Key insights extraction

**When to use:** You want a concise list of the most important findings across many entries.

```

SELECT p.product_name,
 aidb.summarize_text_aggregate(
 sn.note_text,
 aidb.summarize_text_config(
 my-llm,
 prompt => Extract the top 5 key insights from these entries as a
↳numbered list.
 Focus on actionable findings, patterns, and important trends.
)::json
) AS insights
FROM sales_notes sn
JOIN sales_orders so ON sn.order_id = so.order_id
JOIN products p ON so.product_id = p.product_id
GROUP BY p.product_name
ORDER BY count(sn.note_id) DESC
LIMIT 5;

```

- “Top N” and “numbered list” in the prompt constrain output length.
- Works best with moderate group sizes (50–500 notes). Very large groups may cause the LLM to produce generic summaries instead of specific insights.

#### 5. Issue categorization

**When to use:** You want to classify issues or themes into predefined buckets.

```

SELECT p.product_name,
 aidb.summarize_text_aggregate(
 sn.note_text,
 aidb.summarize_text_config(
 my-llm,
 prompt => Categorize these sales notes into the following categories:
 Pricing Concern, Technical Issue, Competitive Loss,

```

(continues on next page)

(continued from previous page)

```

Timing/Budget, Feature Gap, Relationship/Trust, Integration
↳Challenge.
For each category provide the count and percentage of
↳entries that match,
plus a brief description.
)::json
) AS categories
FROM sales_notes sn
JOIN sales_orders so ON sn.order_id = so.order_id
JOIN products p ON so.product_id = p.product_id
WHERE so.status IN (lost, closed_lost)
GROUP BY p.product_name;

```

- Define your categories explicitly in the prompt — don't let the LLM choose.
- Request counts/percentages to get quantitative output.
- The LLM may not perfectly follow the requested format. For more reliable formatting, use wrapper functions ([Recipe 8](#)).

## 6. Executive summaries (reduce strategy)

**When to use:** You need a short, compressed summary rather than a detailed one. Good for dashboards and reports.

```

SELECT p.product_name,
 aidb.summarize_text_aggregate(
 sn.note_text,
 aidb.summarize_text_config(
 my-llm,
 aidb.chunk_text_config(100, 100, 10, words),
 NULL,
 reduce,
 3 -- reduction_factor: each chunk targets desired_length/3 words
)::json
) AS executive_summary
FROM sales_notes sn
JOIN sales_orders so ON sn.order_id = so.order_id
JOIN products p ON so.product_id = p.product_id
GROUP BY p.product_name;

```

How it works: text is split into 100-word chunks; each chunk is summarized targeting `desired_length / reduction_factor` words ( $100/3 \approx 33$  words per chunk); the summaries are combined, re-chunked, and re-summarized until the final result fits within `desired_length`. Output is ~400–900 characters (highly compressed).

- `reduction_factor=3` compresses each chunk to ~33% of its size; `rf=2` to ~50%.
- Always use word-based chunking with `reduce` — char-based can fail (see [Known limitations](#)).
- Minimum safe chunk size: 50 words.

## 7. Detailed summaries (append strategy)

**When to use:** You want comprehensive coverage that preserves detail from all parts of the text.

```

SELECT p.product_name,
 aidb.summarize_text_aggregate(
 sn.note_text,
 aidb.summarize_text_config(
 my-llm,
 aidb.chunk_text_config(200, 200, 20, words),
 NULL,
 append
)::json
) AS detailed_summary
FROM sales_notes sn
JOIN sales_orders so ON sn.order_id = so.order_id
JOIN products p ON so.product_id = p.product_id
GROUP BY p.product_name;

```

How it works: text is split into 200-word chunks with 20-word overlap; each chunk is summarized independently; summaries are concatenated. Output is ~5,000–15,000 characters (verbose, detailed).

- Larger chunks (200 words) = fewer LLM calls = faster, but less granular.
- Smaller chunks (50 words) = more LLM calls = slower, but more detailed.
- The 20-word overlap ensures context isn't lost at chunk boundaries.

## 8. Reusable config wrapper functions

**When to use:** You want consistent, repeatable analysis with constrained output formats.

Create SQL functions that return a pre-built configuration, then pass them directly to `summarize_text_aggregate()`. This pattern wraps the prompt logic inside a function that returns JSON, so you can use it cleanly with `GROUP BY` without repeating long prompt strings.

### Focused analysis

```

CREATE OR REPLACE FUNCTION analyze_top_reasons_config(
 p_question TEXT,
 p_n INTEGER DEFAULT 3,
 p_model TEXT DEFAULT my-llm
) RETURNS JSON AS $$
 SELECT aidb.summarize_text_config(
 p_model,
 prompt => format(
 Answer this question about the text: %s
 Give exactly %s reasons, each as a single sentence.
 No preamble, no extra commentary.,
 p_question, p_n
)
)::json;
$$ LANGUAGE sql;

-- Usage: pass the config function directly to the aggregate
SELECT product_name,
 aidb.summarize_text_aggregate(
 note_text,
 analyze_top_reasons_config(Why are customers churning?, 3)

```

(continues on next page)

(continued from previous page)

```

) AS churn_reasons
FROM sales_notes sn
JOIN sales_orders so ON sn.order_id = so.order_id
JOIN products p ON so.product_id = p.product_id
WHERE so.status = lost
GROUP BY product_name;

```

### Structured sentiment

```

CREATE OR REPLACE FUNCTION analyze_sentiment_config(
 p_model TEXT DEFAULT my-llm
) RETURNS JSON AS $$
 SELECT aidb.summarize_text_config(
 p_model,
 prompt => Analyze sentiment. Respond ONLY in this format:
OVERALL: [positive/negative/neutral/mixed]
SCORE: [1-10]
TOP_POSITIVE: [one sentence]
TOP_NEGATIVE: [one sentence]
TREND: [improving/declining/stable]
)::json;
$$ LANGUAGE sql;

-- Usage
SELECT product_name,
 aidb.summarize_text_aggregate(note_text, analyze_sentiment_config()) AS
 sentiment
FROM sales_notes sn
JOIN sales_orders so ON sn.order_id = so.order_id
JOIN products p ON so.product_id = p.product_id
GROUP BY product_name;

```

### Issue categorizer

```

CREATE OR REPLACE FUNCTION categorize_issues_config(
 p_categories TEXT[] DEFAULT ARRAY[
 pricing, technical, competition,
 timing, budget, feature_gap, other],
 p_model TEXT DEFAULT my-llm
) RETURNS JSON AS $$
 SELECT aidb.summarize_text_config(
 p_model,
 prompt => format(
 Categorize the issues in this text into ONLY these categories: %s.
 For each category, give a count and one representative example.
 Format: CATEGORY: count - "example quote",
 array_to_string(p_categories, ,)
)
)::json;
$$ LANGUAGE sql;

-- Usage

```

(continues on next page)

(continued from previous page)

```

SELECT product_name,
 aidb.summarize_text_aggregate(note_text, categorize_issues_config()) AS
→categories
FROM sales_notes sn
JOIN sales_orders so ON sn.order_id = so.order_id
JOIN products p ON so.product_id = p.product_id
WHERE so.status IN (lost, closed_lost)
GROUP BY product_name;

```

- Config wrapper functions give you a consistent interface and let you tune prompts without changing queries.
- The return type must be JSON (not JSONB ) to match summarize\_text\_aggregate 's parameter type.
- “No preamble”, “ONLY”, “respond in this format” are effective steering phrases.

## 9. Text scrubbing before summarization

**When to use:** Your text data contains filler words, greetings, signatures, or boilerplate that waste LLM tokens.

Create a cleanup function and apply it before the aggregate:

```

CREATE OR REPLACE FUNCTION clean_note_text(p_text TEXT)
RETURNS TEXT AS $$
BEGIN
 RETURN regexp_replace(
 regexp_replace(
 regexp_replace(
 regexp_replace(p_text,
 -- Remove email signatures
 E(Best regards|Sincerely|Thanks|Cheers), ?\s*\n.*$,
 , gi),
 -- Remove greeting lines
 E^(Hi|Hello|Dear|Hey)\s+[A-Za-z]+, ?\s*\n,
 , gi),
 -- Remove filler phrases
 E\b(basically|essentially|actually|literally|you know|I mean|sort of|kind
→of)\b,
 , gi),
 -- Collapse whitespace
 E\s+, , g);
END;
$$ LANGUAGE plpgsql IMMUTABLE;

-- Use it:
SELECT product_name,
 aidb.summarize_text_aggregate(
 clean_note_text(note_text),
 aidb.summarize_text_config(my-llm)::json
) AS summary
FROM sales_notes sn
JOIN sales_orders so ON sn.order_id = so.order_id
JOIN products p ON so.product_id = p.product_id
GROUP BY product_name;

```

- IMMUTABLE lets PostgreSQL optimize repeated calls.
- Customize the regex patterns for your data (e.g., internal jargon, standard headers).
- Typical reduction: 15–30% fewer tokens, which means fewer chunks and faster processing.

### Combining recipes: the full optimization stack

For maximum speed and quality, combine scrubbing + filtering + a config wrapper:

```
SELECT p.product_name,
 aidb.summarize_text_aggregate(
 clean_note_text(sn.note_text),
 analyze_top_reasons_config(Why are we losing these deals?, 3)
) AS loss_reasons
FROM sales_notes sn
JOIN sales_orders so ON sn.order_id = so.order_id
JOIN products p ON so.product_id = p.product_id
WHERE so.status IN (lost, closed_lost)
GROUP BY p.product_name;
```

This pipeline: filters to relevant rows (lost deals); strips filler text per row via `clean_note_text()` (15–30% fewer tokens); accumulates all cleaned text, then sends it to the LLM; applies a constrained prompt via the config wrapper for shorter, focused output.

### Performance benchmarks

All data from a test suite run against a sales dataset (18,503 notes, avg 266 chars/note) using `granite4:1b` via Ollama on AIDB 6.1.0.

#### No-chunking performance

With no chunking, all text in a group is sent as a single LLM call. Fast, but limited by the model’s context window.

Group size	Duration	Result size	Notes
77–84 notes	21–28s	1,100–2,500 chars	Safe range, full data processed
100 notes	28s	2,200 chars	~26K chars input, well within context
2,915 notes	98s	3,200 chars	Exceeds 131K token context window (see below)
3,248 notes	91s	1,700 chars	Only partial data processed

#### Chunking strategy comparison (50 notes)

Config	Est. LLM calls	Duration	ms/call	Result chars	Status
No chunking	1	12s	11,947	1,443	PASS
words/50/append	32	152s	4,759	12,159	PASS
words/100/append	16	77s	4,832	7,021	PASS
words/200/append	8	48s	6,044	4,781	PASS
words/50/reduce rf=2	63	217s	3,452	545	PASS
words/100/reduce rf=2	31	102s	3,280	475	PASS

continues on next page

Table 5 – continued from previous page

Config	Est. LLM calls	Duration	ms/call	Result chars	Status
words/200/reduce rf=2	15	51s	3,367	767	PASS
words/50/reduce rf=3	50	170s	3,396	410	PASS
words/100/reduce rf=3	25	72s	2,897	867	PASS
words/200/reduce rf=3	12	44s	3,661	535	PASS
chars/200/append	40	246s	6,154	16,355	PASS
chars/400/append	20	137s	6,835	11,793	PASS
chars/800/append	10	66s	6,584	5,897	PASS
chars/200/reduce rf=2	81	971s	11,988	—	FAIL
chars/400/reduce rf=2	41	368s	8,967	441	PASS
chars/800/reduce rf=2	21	144s	6,842	850	PASS

### Strategy efficiency summary

Strategy	Avg LLM calls	Avg total time	Avg ms/call	Avg output
No chunking	1	12s	11,947	1,443 chars
Append	21	121s	5,868	9,668 chars
Reduce rf=2	34	176s	5,182	616 chars
Reduce rf=3	29	95s	3,318	604 chars

### Projected scale

Scenario	100 notes	500 notes	1,000 notes
No chunking	28s (1 call)	~60s (1 call)	~100s (1 call, partial data*)
words/100/append	~77s (16 calls)	~385s (80 calls)	~770s (160 calls)
words/100/reduce rf=3	~72s (25 calls)	~180s (62 calls)	~240s (125 calls)

- No-chunking at 1,000+ notes may exceed the context window. Use chunking for completeness.

### Recommended configurations

Use case	Config	Why
Fastest possible (< 500 notes)	No chunking	Single LLM call, stays within context window
Fast + detailed	words/200/append	Fewest calls for append, good detail
Balanced detail	words/100/append	More granular, moderate speed
Executive summary	words/100/reduce rf=3	Compressed output, efficient reduce
Maximum compression	words/50/reduce rf=3	Smallest output, most LLM calls

General guidance: start with no chunking; add chunking only if groups exceed ~500 notes. Always prefer word-based chunking over char-based. For reduce, prefer `rf=3` over `rf=2` — fewer total calls with similar output quality.

## Known limitations

### Context window overflow (silent)

When the total input text exceeds the model's context window, the text is silently truncated. No error is raised — the LLM processes whatever fits and ignores the rest.

For example: `granite4:1b` has a 131,072 token context window. A group of 3,248 notes (~806K chars, ~201K tokens) exceeds this by 53%. The result is a valid summary, but only of a fraction of the data.

Safe limits (no chunking):

Avg note size	Safe note count	Total tokens
250 chars	~500 notes	~33K tokens
500 chars	~250 notes	~33K tokens
1,000 chars	~125 notes	~33K tokens

Workaround: use chunking for any group that might exceed these limits. Chunking processes all data regardless of context window size.

```
-- Safe for large groups:
aidb.summarize_text_config(
 my-llm,
 aidb.chunk_text_config(200, 200, 20, words),
 NULL,
 append
)::json
```

### Reduce fails with small chunks

The `reduce` strategy requires each summarization round to produce output shorter than its input. With small chunk sizes, the LLM often generates summaries longer than the input, causing:

```
Text summarization failed to reduce size. The summarized output is larger than or
→equal to the input.
Consider adjusting the chunking parameters.
```

Failure thresholds:

Chunk type	Size	Result
chars/200/reduce	200 chars	Always fails
chars/400/reduce	400 chars	Unreliable
chars/800/reduce	800 chars	Reliable
words/50/reduce	50 words	Reliable
words/100/reduce	100 words	Reliable
words/200/reduce	200 words	Reliable

Workaround: use word-based chunking with `reduce` (100% pass rate at all tested sizes). If you must use char-based, use `>= 800` chars.

## Controlling temperature, max\_tokens, and system prompt

By default, the LLM uses its provider-default temperature (typically 0.7–1.0), which can cause high variance in output across runs. Control these parameters per call using `inference_config`:

```
-- Control temperature and max_tokens for more consistent, bounded output
SELECT p.product_name,
 aidb.summarize_text_aggregate(
 sn.note_text,
 aidb.summarize_text_config(
 my-llm,
 NULL,
 Summarize the key points,
 NULL,
 NULL,
 aidb.inference_config(
 temperature => 0.3,
 max_tokens => 500
)
)::json
) AS summary
FROM sales_notes sn
JOIN sales_orders so ON sn.order_id = so.order_id
JOIN products p ON so.product_id = p.product_id
GROUP BY p.product_name;

-- Override the system prompt for non-summarization tasks
SELECT aidb.summarize_text(
 input => Customer complained about slow response times and billing errors.,
 options => aidb.summarize_text_config(
 my-llm,
 NULL,
 NULL,
 NULL,
 NULL,
 aidb.inference_config(
 system_prompt => You are a customer support analyst. Classify issues by_
↪severity.,
 temperature => 0.2
)
)::json
);
```

!!!Tip For best results, combine `inference_config` with constrained prompts. For example, set `max_tokens => 200` and use a prompt like 'Give exactly 3 bullet points, max 1 sentence each.' — the token limit acts as a hard cap while the prompt guides the format.

## Empty results with no error

In some cases the LLM returns an empty string after processing. AIDB stores this as a successful result with no error. Always check `length(result)` in your application code:

```
SELECT product_name,
 CASE
```

(continues on next page)

(continued from previous page)

```

 WHEN length(summary) = 0 THEN ERROR: Empty result
 WHEN length(summary) < 50 THEN WARNING: Suspiciously short
 ELSE summary
 END AS validated_summary
FROM (
 SELECT p.product_name,
 aidb.summarize_text_aggregate(
 sn.note_text,
 aidb.summarize_text_config(
 my-llm,
 prompt => Your prompt here
)::json
) AS summary
 FROM sales_notes sn
 JOIN sales_orders so ON sn.order_id = so.order_id
 JOIN products p ON so.product_id = p.product_id
 GROUP BY p.product_name
) sub;

```

### Prompt adherence weakens at scale

With large text volumes (1,000+ notes), the LLM tends to default to summarization behavior regardless of the custom prompt. Structured output requests (numbered lists, category formats, sentiment labels) are less reliably followed.

Workarounds: reduce input volume using `LIMIT` or more aggressive filters; use chunking with `append` so each chunk receives the prompt independently against smaller input; use wrapper functions with very explicit format constraints; scrub text ( Recipe 9 ) to reduce noise before summarization.

## Optimization patterns

### 1. Text scrubbing

Strip filler words, greetings, and boilerplate before summarization. See Recipe 9 . Impact: 15–30% fewer tokens, proportionally faster processing.

### 2. Constrained prompts

Telling the LLM to produce specific, short-format output reduces token generation time (the slowest part of each call).

```

-- Slow (open-ended, LLM may generate 2,000+ chars):
Summarize these notes.

-- Fast (constrained, LLM generates ~200 chars):
Give exactly 3 bullet points, max 1 sentence each.

```

### 3. Extractive pre-filtering

For very large datasets, use SQL to pre-filter to the most relevant text before sending to the LLM:

```

-- Only send notes containing relevant keywords
SELECT aidb.summarize_text_aggregate(
 note_text,

```

(continues on next page)

(continued from previous page)

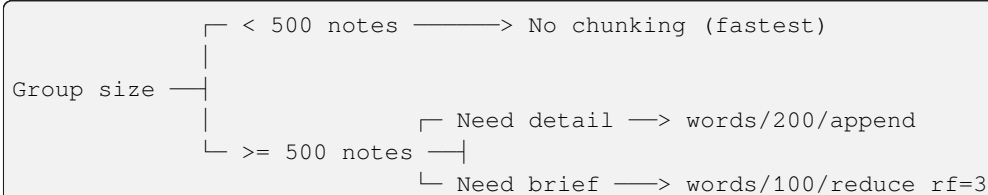
```

aidb.summarize_text_config(
 my-llm,
 prompt => What are the pricing concerns?
)::json
)
FROM sales_notes
WHERE note_text ILIKE %price%
 OR note_text ILIKE %cost%
 OR note_text ILIKE %budget%
 OR note_text ILIKE %expensive%
GROUP BY product_id;

```

Impact: dramatically fewer rows to process. The LLM sees only relevant text, improving both speed and quality.

#### 4. Choosing the right strategy



#### 5. Performance cost model

The dominant cost is LLM call count. Each call takes 3–12 seconds depending on input size.

Step	Time
Network overhead	~50ms
Prompt processing	~1–3s
Token generation	~2–8s
Response parsing	~10ms
<b>Total per call</b>	<b>~3–12s</b>

Prompt processing scales with input length, but token generation scales with output length. Since summaries are roughly fixed-size regardless of input volume, fewer, larger calls are more efficient than many small calls — up to the context window limit. This is why no-chunking is so effective for small groups: it maximizes text per call and minimizes the number of calls. The aggregate already accumulates all rows before processing, so the text is naturally batched.

## 14.8 AI inference functions

AIDB provides a set of SQL functions for performing AI inference directly in queries. These functions let you encode data into vector representations, generate text with language models, and control inference behavior — without setting up a pipeline.

### 14.8.1 aidb.encode\_text()

Encodes a single text string into a vector using a registered embedding model.

```
SELECT aidb.encode_text(model_name => my_embedder, input => What is pgvector?);

__OUTPUT__
 encode_text

[0.021,-0.043,0.117, ...]
(1 row)
```

For the full parameter reference, see [Models reference](#) .

---

### 14.8.2 aidb.encode\_text\_batch()

Encodes multiple text strings into vectors in a single batched request. More efficient than calling `encode_text()` in a loop when processing many inputs.

```
SELECT * FROM aidb.encode_text_batch(
 model_name => my_model_name,
 input => ARRAY[What is pgvector?, How does RAG work?, Explain embeddings.]
);

__OUTPUT__
 encode_text_batch

[0.021,-0.043,0.117,...]
[0.008, 0.091,-0.034,...]
[0.055,-0.012, 0.076,...]
(3 rows)
```

For the full parameter reference, see [Models reference](#) .

---

### 14.8.3 aidb.encode\_image()

Encodes a single image into a vector using a registered multimodal embedding model (for example, a CLIP model).

```
SELECT aidb.encode_image(
 model_name => my_model_name,
 input => pg_read_binary_file(/path/to/image.png)::BYTEA
);

__OUTPUT__
 encode_image

[-0.012, 0.084, 0.031, ...]
(1 row)
```

For the full parameter reference, see [Models reference](#) .

---

### 14.8.4 aidb.generate\_text()

Generates text from a language model (LLM) given a prompt. This is the core function for LLM inference in AIDB.

```
SELECT aidb.generate_text(
 model_name => my_model_name,
 input => Explain what a knowledge base is in two sentences.
);
```

\_\_OUTPUT\_\_

generate\_text

---

```
→-----
A knowledge base is a structured repository of information used to answer queries or...
→...
```

(1 row)

To control generation behavior, pass an `inference_config` object built by `aidb.inference_config()`. Cast its JSONB result to `json` so the call matches the `aidb.generate_text(TEXT, TEXT, json)` signature:

```
SELECT aidb.generate_text(
 model_name => my_model_name,
 input => Summarize the following text: ...,
 inference_config => aidb.inference_config(
 system_prompt => You are a concise technical writer.,
 temperature => 0.2,
 max_tokens => 256
)::json
);
```

For the full parameter reference, see [Models reference](#).

### 14.8.5 aidb.generate\_text\_batch()

Generates text responses for multiple prompts in a single batched request.

```
SELECT aidb.generate_text_batch(
 model_name => my_model_name,
 input => ARRAY[
 What is a vector database?,
 What is retrieval-augmented generation?
]
);
```

\_\_OUTPUT\_\_

generate\_text\_batch

---

```
A vector database stores high-dimensional vectors ...
Retrieval-augmented generation (RAG) combines a ...
```

(2 rows)

For the full parameter reference, see [Models reference](#).

### 14.8.6 aidb.inference\_config()

Builds an inference configuration object to control language model generation behavior. Pass the result as the `inference_config` argument of `aidb.generate_text()` or `aidb.generate_text_batch()`, or as part of `aidb.summarize_text_config()` inside a pipeline step.

```
SELECT aidb.generate_text(
 model_name => my_llm,
 input => Write a haiku about PostgreSQL.,
 inference_config => aidb.inference_config(
 temperature => 0.9,
 max_tokens => 64,
 seed => 42
)::json
);
```

For the full parameter reference, see [Models reference](#).

!!!note To use these functions as steps inside a pipeline, see [Pipeline steps](#).

## 14.9 Text reranking

Use the `aidb.rerank_text()` function to reorder a set of candidate text passages by their relevance to a query. Reranking is typically applied as a post-retrieval step — after an initial set of candidates has been retrieved from a knowledge base — to surface the most relevant results at the top.

### Step 1: Register a reranking model

```
-- Remote: NVIDIA NIM
SELECT aidb.create_model(
 my_reranker,
 nim_reranking,
 config => aidb.nim_reranking_config(
 api_key => nvapi-..., -- or use create_models own credentials_env parameter.
 ↳instead, to avoid storing this
 model => nvidia/nv-rerankqa-mistral-4b-v3
)
);

-- Local: llama.cpp (GGUF), no API key required
SELECT aidb.create_model(
 my_reranker,
 llamacpp_reranking,
 config => jsonb_build_object(
 model, gpustack/bge-reranker-v2-m3-GGUF,
 model_file, bge-reranker-v2-m3-Q4_K_M.gguf,
 n_ctx, 2048
)
);
```

### Step 2: Rerank a set of candidates

```
SELECT * FROM aidb.rerank_text(
 model_name => my_reranker,
```

(continues on next page)

(continued from previous page)

```

query => What is retrieval-augmented generation?,
input => ARRAY[
 RAG combines retrieval and generation to produce grounded answers.,
 Vector databases store high-dimensional embeddings for similarity search.,
 RAG systems retrieve relevant documents before generating a response.
]
) ORDER BY logit_score DESC;

__OUTPUT__

```

text	logit_score	id
RAG systems retrieve relevant documents <b>before</b> generating ...	0.97	2
RAG combines retrieval <b>and</b> generation <b>to</b> produce grounded ...	0.91	0
Vector databases store high-dimensional embeddings <b>for</b> ...	0.43	1

(3 rows)

The id column is the zero-based index of each string in the input array, so you can join back to the original data.

For the full parameter reference, see [Models reference](#).

### 14.9.1 Using reranking after knowledge base retrieval

A typical pattern is to retrieve an initial candidate set from a knowledge base and then rerank the results:

```

WITH candidates AS (
 SELECT key, value
 FROM aidb.retrieve_text(my_kb, What is RAG?, topk => 20)
)
SELECT c.key, c.value, r.logit_score
FROM candidates c
JOIN aidb.rerank_text(
 model_name => my_reranker,
 query => What is RAG?,
 input => ARRAY(SELECT value FROM candidates ORDER BY key)
) r ON r.text = c.value
ORDER BY r.logit_score DESC
LIMIT 5;

```

!!!note Reranking models score passages against a query but don't generate text. To register a reranking model, use the appropriate [Model config helpers](#) for your provider. For example, `aidb.nim_reranking_config()` for NVIDIA NIM reranking endpoints. `llamacpp_reranking` has no config helper yet; build config with `jsonb_build_object()` instead, as shown above. See [Local models](#).

!!!note `logit_score`'s scale depends on the provider: some return a normalized score in a fixed range (as in the example above), while `llamacpp_reranking` returns a raw, unbounded model logit. Compare scores within one `aidb.rerank_text()` call and provider, not across different providers or calls.



# CHAPTER 15

## Reference

Complete API reference for the AIDB extension: parameter tables, return types, and signatures for lookup. For usage context and worked examples, see the guide sections linked in the table below.

Page	What it covers	Related guide
<a href="#">Models reference</a>	Catalog views, model CRUD ( <i>create_model</i> , <i>get_model</i> , <i>delete_model</i> ), HCP model functions, inference functions ( <i>encode_text</i> , <i>generate_text</i> , <i>rerank_text</i> ), and all model config helpers	<a href="#">Integrating models</a>
<a href="#">Knowledge bases reference</a>	Knowledge base views and stats, retrieval functions ( <i>retrieve_key</i> , <i>retrieve_text</i> ), volume functions, and auto-processing	<a href="#">Knowledge bases reference</a>
<a href="#">Operations reference</a>	Pipeline error log functions ( <i>get_error_logs</i> , <i>clear_error_logs</i> , <i>requeue_pipeline_errors</i> )	<a href="#">Pipeline error log</a>
<a href="#">Functions reference</a>	Standalone SQL functions: <i>chunk_text</i> , <i>parse_html</i> , <i>parse_pdf</i> , <i>perform_ocr</i> , <i>summarize_text</i>	<a href="#">SQL functions</a>
<a href="#">Pipelines reference</a>	Pipeline types, views, CRUD functions, step config helpers, and vector index config helpers	<a href="#">AI pipelines</a>
<a href="#">Agents reference</a>	Agent CRUD, conversation functions, structured output helpers, and catalog views	<a href="#">Agents reference</a>
<a href="#">Tools reference</a>	Tool catalog view, SQL tool registration, MCP server management, and parameter helpers	<a href="#">Tools reference</a>

## 15.1 Models reference

Reference for all model-related functions and views in AIDB. For guide-style documentation, see [Integrating models](#) .

### 15.1.1 Configuration model

A model's behavior is controlled two ways, and it matters which one you use for a given setting:

- **Creation-time ``config``** — set when you call `aidb.create_model()`, using a provider-specific *Model config helpers* (`aidb.bert_config()`, `aidb.completions_config()`, and so on). This is the model's persistent, stored configuration.
- **Call-time ``inference\_config``** — an optional per-call override, built with `aidb.inference_config()`, passed directly to an inference function. It overrides the model's stored config for that one call only; anything you don't override falls back to the value set at creation time.

Only some inference functions accept a call-time override — the rest are fixed entirely by the model's creation-time config:

Function	Call-time override?
<code>aidb.generate_text()</code> , <code>aidb.generate_text_batch()</code>	Yes — pass <code>aidb.inference_config()</code> (or raw JSON) as the <i>inference_config</i> argument.
<code>aidb.summarize_text()</code> , <code>aidb.summarize_text_aggregate()</code>	Yes — via the <i>inference_config</i> field of <i>AI pipelines</i> .
<code>aidb.encode_text()</code> , <code>aidb.encode_text_batch()</code> , <code>aidb.encode_image()</code>	No. Behavior is fixed by <i>config</i> at creation time.
<code>aidb.rerank_text()</code>	No. Behavior is fixed by <i>config</i> at creation time.
<code>aidb.perform_ocr()</code>	No — its <i>options</i> parameter only selects which registered model to run, not inference settings.

This asymmetry follows from what each capability needs: text generation has runtime knobs worth tuning per call (temperature, system prompt, and so on); embeddings, reranking, and OCR don't, so there's nothing to override.

### 15.1.2 Catalog views

#### `aidb.model_providers`

Lists all available model providers registered in the system.

Column	Type	Description
<code>server_name</code>	name	Name of the model provider
<code>server_description</code>	text	Description of the provider
<code>server_options</code>	text[]	Available configuration options

#### `aidb.models`

Lists all models in the registry, including default and user-created models.

Column	Type	Description
<code>name</code>	text	User-defined name for the model
<code>provider</code>	text	Model provider name
<code>options</code>	text[]	Configured options for the model
<code>functions</code>	text[]	Internal capability identifiers the model's provider supports (for example, <i>aidb-openai-text-embeddings</i> , <i>aidb-openai-text-completion</i> )

### Example

```
SELECT * FROM aidb.models;
```

```
__OUTPUT__
name | provider | options | functions
-----+-----+-----+-----
bert | bert_local | {"config={}} | {aidb-openai-text-embeddings}
clip | clip_local | {"config={}} | {aidb-openai-text-image-embeddings}
t5 | t5_local | {"config={}} | {aidb-openai-text-embeddings,aidb-openai-text-
completion}
(3 rows)
```

## 15.1.3 Model management functions

### aidb.create\_model

Registers a new model in the AIDB model registry.

#### Parameters

Parameter	Type	Default	Description
<i>name</i>	TEXT	Required	Unique name for the model.
<i>provider</i>	TEXT	Required	Provider name (see <i>aidb.model_providers</i> ).
<i>config</i>	JSONB	'{}'	Provider-specific configuration. Build with a <i>Model config helpers</i> .
<i>credentials</i>	JSONB	'{}'	Provider credentials (for example, {"api_key": "..."}).
<i>replace_credentials</i>	BOOLEAN	false	If true, updates stored credentials without re-creating the model.
<i>validate</i>	BOOLEAN	true	If true, probe the model at creation time to confirm it's functional. See <i>Validation</i> .
<i>credentials_env</i>	TEXT	NULL	Name of an environment variable to read credentials from instead of storing them. See <i>Credentials from environment variables</i> . Mutually exclusive with <i>credentials</i> .

#### Credentials from environment variables

!!!note Don't want a provider's API key or password sitting in a database table at all, even one with restricted grants? Pass *credentials\_env* instead of *credentials*, naming a Postgres environment variable to read the credential from. Only the variable's *name* is stored — the value itself is read fresh from the Postgres backend process's environment every time the model is used, and is never persisted in the database.

Pass a variable name instead of a *credentials* object:

```

SELECT aidb.create_model(
 name => my_openai,
 provider => openai_embeddings,
 config => aidb.embeddings_config(model => text-embedding-3-small),
 credentials_env => AIDB_OPENAI_API_KEY
);

```

The referenced variable can hold either a bare secret (used as `api_key`) or a JSON object (for example, `{"basic_auth": "..."}` ) — the same shapes `credentials` itself accepts. `credentials` and `credentials_env` are mutually exclusive; passing both raises an error.

For security, the environment variable's name must start with the prefix configured by *`aidb.env_var_allowed_prefix`* (`AIDB_` by default) — this stops a model or MCP server configuration from naming an arbitrary, unrelated environment variable on the Postgres host and exfiltrating its value. The same mechanism and the same GUC apply to MCP server authentication; see *`headers_env`*.

## TLS configuration

To connect to HTTPS model endpoints, include a `tls_config` field inside `config`:

```

"tls_config": {
 "insecure_skip_verify": true,
 "ca_path": "/etc/aidb/myCA.pem"
}

```

## Validation

By default (`validate => true`), `aidb.create_model()` runs a minimal probe inference before the registration commits, to confirm the model is functional:

- **Local models** are downloaded and loaded, and a test inference is run — so a bad model identifier or unloadable weights fail at creation time rather than on first use.
- **External models** receive a small test request — so an invalid API key, wrong URL, or unreachable endpoint is caught right away.

If the probe fails, `aidb.create_model()` raises an error and registers nothing. Pass `validate => false` to skip the probe — for example, to defer a large download to first use, or to register a model before its endpoint or credentials are available. You can validate a model later with `aidb.validate_model()`.

## Examples

```

-- Minimal registration
SELECT aidb.create_model(my_t5, t5_local);

-- With config and credentials
SELECT aidb.create_model(
 name => my_t5,
 provider => t5_local,
 config => {"param1": "value1"}::JSONB,
 credentials => {"token": "abcd"}::JSONB
);

-- Rotate credentials without re-creating
SELECT aidb.create_model(

```

(continues on next page)

(continued from previous page)

```

name => my_openai,
provider => openai_embeddings,
config => aidb.embeddings_config(model => text-embedding-3-small),
credentials => {"api_key": "<new-key>"}::JSONB,
replace_credentials => true
);

```

## aidb.get\_model

Returns the configuration for a registered model.

### Parameters

Parameter	Type	Description
<i>model_name</i>	TEXT	Name of the model to retrieve.

### Returns

Column	Type	Description
<i>name</i>	text	Model name
<i>provider</i>	text	Provider name
<i>options</i>	text[]	Configured options

### Example

```
SELECT * FROM aidb.get_model(t5);
```

```

__OUTPUT__
name | provider | options
-----+-----+-----
t5 | t5_local | {"config={}" }
(1 row)

```

## aidb.delete\_model

Removes a model from the registry. Doesn't affect pipelines or knowledge bases that reference it until they are next executed.

### Parameters

Parameter	Type	Description
<i>model_name</i>	TEXT	Name of the model to delete.

## Returns

The name, provider, and options of the deleted model.

## Example

```
SELECT aidb.delete_model(t5);

__OUTPUT__
delete_model

(t5,t5_local,{"config={}"})
(1 row)
```

## aidb.validate\_model

Probes a registered model to confirm it's functional, running a minimal test inference. Called automatically by `aidb.create_model()` unless `validate => false`; you can also call it directly to recheck an existing model.

## Parameters

Parameter	Type	Description
<i>model_name</i>	TEXT	Name of the model to validate.

## Returns

TEXT — the model capability that was probed (for example, `text embedding`, `language`, `reranking`, `image embedding`, or `OCR`). Raises an error if the probe fails.

## Example

```
SELECT aidb.validate_model(my_bert);

__OUTPUT__
validate_model

text embedding
(1 row)
```

## 15.1.4 HCP model functions

These functions manage models running on EDB Hybrid Manager (HM).

### aidb.list\_hcp\_models

Lists models currently running on HM.

## Returns

Column	Type	Description
<i>name</i>	text	Model instance name on HCP
<i>url</i>	text	API endpoint URL
<i>model</i>	text	Model identifier

## Example

```
SELECT * FROM aidb.list_hcp_models();
```

__OUTPUT__		
name		url
model		
-----+-----		
llama-3-1-8b-instruct-1xgpu		http://llama-3-1-8b-predictor.default.svc.local
meta/llama-3.1-8b-instruct		
(1 row)		

## aidb.create\_hcp\_model

Registers an HCP-hosted model by referencing its running instance name.

## Parameters

Parameter	Type	Description
<i>name</i>	TEXT	User-defined name for the model in AIDB.
<i>hcp_model_name</i>	TEXT	Name of the model instance running on HCP.

## aidb.sync\_hcp\_models

Synchronizes the AIDB model registry with models currently running on HCP. Creates entries for new HCP models and deletes entries for models no longer running there.

## Returns

Column	Type	Description
<i>status</i>	text	<i>created</i> , <i>deleted</i> , <i>unchanged</i> , or <i>skipped</i>
<i>model</i>	text	Name of the synchronized model

### 15.1.5 Inference functions

!!!note `aidb.generate_text()` and `aidb.generate_text_batch()` were previously named `aidb.decode_text()` and `aidb.decode_text_batch()`. The old names still work but are deprecated and will be removed in a future version — use the new names in new code.

#### `aidb.encode_text`

Encodes a single text string into a vector using the specified model.

##### Parameters

Parameter	Type	Description
<i>model_name</i>	TEXT	Name of the registered model.
<i>input</i>	TEXT	Text to encode.

##### Returns

`real[]` — the embedding for the input text. Cast to `vector` to use with `pgvector` operators.

##### Example

```
SELECT aidb.encode_text(bert_local, The quick brown fox);
```

#### `aidb.encode_text_batch`

Encodes an array of text strings into vectors in a single call.

##### Parameters

Parameter	Type	Description
<i>model_name</i>	TEXT	Name of the registered model.
<i>input</i>	TEXT[]	Array of strings to encode.

##### Returns

SETOF `real[]` — one row per input, in input order. The result column is named `encode_text_batch`.

##### Example

```
SELECT * FROM aidb.encode_text_batch(bert_local, ARRAY[The quick brown fox, Lorem
↳ ipsum dolor sit amet]);
```

#### `aidb.encode_image`

Encodes a binary image into a vector using a multimodal model (for example, CLIP).

## Parameters

Parameter	Type	Description
<i>model_name</i>	TEXT	Name of the registered model.
<i>input</i>	BYTEA	Raw image bytes.

## Returns

`real[]` — the embedding for the input image. Cast to `vector` to use with `pgvector` operators.

## Example

```
SELECT aidb.encode_image(clip_local, pg_read_binary_file('/tmp/photo.jpg')::BYTEA);
```

## aidb.generate\_text

Generates a text response from a prompt using the specified model.

## Parameters

Parameter	Type	Default	Description
<i>model_name</i>	TEXT	Required	Name of the registered model.
<i>input</i>	TEXT	Required	Text prompt or input.
<i>inference_config</i>	JSON	NULL	Runtime inference settings. Use <code>:ref:``aidb.inference_config()``</code> <code>&lt;#aidbinference_config&gt;</code> . Cast the result to <code>json</code> .

## Returns

TEXT — the generated response.

## Examples

```
-- Basic usage
SELECT aidb.generate_text(t5_local, translate to French: Hello, world.);

-- With inference configuration
SELECT aidb.generate_text(
 my_llama,
 Explain quantum computing in one sentence.,
 aidb.inference_config(
 system_prompt => Be concise and factual.,
 temperature => 0.7,
 max_tokens => 100
)::json
);
```

**aidb.generate\_text\_batch**

Generates text responses for an array of prompts in a single call.

**Parameters**

Parameter	Type	Default	Description
<i>model_name</i>	TEXT	Required	Name of the registered model.
<i>input</i>	TEXT[]	Required	Array of text prompts.
<i>inference_config</i>	JSON	NULL	Runtime inference settings. Use <code>:ref:``aidb.inference_config()``</code> <code>&lt;#aidbinference_config&gt;</code> . Cast the result to <i>json</i> .

**Returns**

SETOF TEXT — one row per input prompt, in input order. The result column is named `generate_text_batch`.

**Example**

```
SELECT aidb.generate_text_batch(t5_local, ARRAY[
 translate to German: hello,
 translate to German: goodbye
]);
```

**aidb.rerank\_text**

Scores and ranks a set of text inputs against a query using a reranking model.

**Parameters**

Parameter	Type	Default	Description
<i>model_name</i>	TEXT	Required	Name of a registered reranking model.
<i>query</i>	TEXT	Required	The query to rank inputs against.
<i>input</i>	TEXT[]	[]	Array of candidate texts to rank.

**Returns**

Column	Type	Description
<i>text</i>	text	The candidate text
<i>logit_score</i>	double precision	Relevance score (higher = more relevant)
<i>id</i>	int	Original index of the text in the input array

## Example

```
SELECT * FROM aidb.rerank_text(
 my_reranker,
 How do I configure AIDB?,
 ARRAY[
 AIDB requires shared_preload_libraries.,
 Postgres supports JSON natively.,
 Run CREATE EXTENSION aidb CASCADE to install.
]
) ORDER BY logit_score DESC;
```

## 15.1.6 Inference configuration helper

### aidb.inference\_config

Builds a JSONB configuration object for runtime inference settings. Pass the result (cast to `json`) to `aidb.generate_text()`, `aidb.generate_text_batch()`, or use it with `aidb.summarize_text_config()`.

All parameters are optional — omit any you don't need.

### Parameters

Parameter	Type	Default	Description
<i>system_prompt</i>	TEXT	NULL	System prompt prepended to the request. Not supported by T5 models.
<i>temperature</i>	DOUBLE PRECISION	NULL	Sampling temperature. <i>0.0</i> is deterministic; higher values increase variety.
<i>max_tokens</i>	INTEGER	NULL	Maximum tokens to generate.
<i>top_p</i>	DOUBLE PRECISION	NULL	Nucleus sampling threshold.
<i>seed</i>	BIGINT	NULL	Random seed for reproducible outputs.
<i>repeat_penalty</i>	REAL	NULL	Penalty for repeating tokens. <i>1.0</i> = no penalty. Not supported by NIM or OpenAI.
<i>repeat_last_n</i>	INTEGER	NULL	Number of recent tokens considered for <i>repeat_penalty</i> . Not supported by NIM or OpenAI.
<i>thinking</i>	BOOLEAN	NULL	Controls <code>&lt;think&gt;</code> tag handling. <i>true/NULL</i> retains tags; <i>false</i> strips them.
<i>extra_args</i>	JSONB	NULL	Additional provider-specific arguments passed directly to the API.

### Returns

JSONB — cast to `::json` before passing to inference functions.

## Examples

```

-- With temperature and token limit
SELECT aidb.generate_text(
 my_llama,
 What is machine learning?,
 aidb.inference_config(temperature => 0.5, max_tokens => 150)::json
);

-- Full configuration
SELECT aidb.generate_text(
 my_llama,
 Explain relativity.,
 aidb.inference_config(
 system_prompt => You are a physics professor. Be precise.,
 temperature => 0.3,
 max_tokens => 200,
 top_p => 0.9,
 seed => 42,
 repeat_penalty => 1.1
)::json
);

-- Hide reasoning tags from a thinking model
SELECT aidb.generate_text(
 my_reasoning_model,
 Solve: 2x + 5 = 15,
 aidb.inference_config(thinking => false)::json
);

```

## Tool calling and structured output

Three more inference settings exist but aren't exposed as named parameters of `aidb.inference_config()` — build them as raw JSON instead, merged with the helper's output using `||` if you also need other settings. **Only** ```llamacpp_generate```, ```openai_responses``` (+ ```openai_responses_azure```), and ```anthropic_messages``` (+ ```anthropic_messages_azure``` / ```anthropic_messages_bedrock```) honor these; every other provider rejects them.

Field	Type	Description
<i>tools</i>	JSONB array	OpenAI-compatible <i>tools</i> array — function definitions the model can call.
<i>tool_choice</i>	JSONB	OpenAI-compatible <i>tool_choice</i> : the string <code>"auto"</code> , <code>"required"</code> , or <code>"none"</code> , or <code>{"type": "function", "function": {"name": "..."} }</code> to force a specific tool.
<i>response_format</i>	JSONB	Structured output. <code>{"type": "json_schema", "json_schema": {"schema": {...}}}</code> constrains the response to a JSON schema. Mutually exclusive with <i>tools/tool_choice</i> on <i>openai_responses</i> and <i>anthropic_messages</i> (and their variants).

!!!note `openai_responses` and `anthropic_messages` (and their Azure/Bedrock variants) map these to each API's native *tools/tool\_choice* request fields and parse real tool-call responses back — see [OpenAI Responses API](#) and [Anthropic Messages API](#). Every other remote provider (`openai_completions`, `gemini`, `nim_completions`, and

so on) rejects `tools /tool_choice /response_format` outright — this rejection only applies to direct calls like `aidb.generate_text()`. Any of those providers (except `t5_local`) can still back an *Agents reference*; the agent framework falls back to simulating tool calls by describing them in the prompt and parsing the model's reply, rather than using these three fields.

```
-- Tool calling
SELECT aidb.generate_text(
 my_llamacpp_chat,
 What is the weather in Paris?,
 (aidb.inference_config(temperature => 0.2) || jsonb_build_object(
 tools, jsonb_build_array(
 jsonb_build_object(
 type, function,
 function, jsonb_build_object(
 name, get_weather,
 description, Get current weather for a city,
 parameters, jsonb_build_object(
 type, object,
 properties, jsonb_build_object(city, jsonb_build_object(type, string)),
 required, jsonb_build_array(city)
)
)
)
),
 tool_choice, auto
))::json
);

-- Structured output
SELECT aidb.generate_text(
 my_llamacpp_chat,
 Extract the city from: I live in Paris.,
 jsonb_build_object(
 response_format, jsonb_build_object(
 type, json_schema,
 json_schema, jsonb_build_object(
 schema, jsonb_build_object(
 type, object,
 properties, jsonb_build_object(city, jsonb_build_object(type, string)),
 required, jsonb_build_array(city)
)
)
)
)::json
);
```

### 15.1.7 Model config helpers

These functions return a JSONB configuration object for the `config` parameter of `aidb.create_model()`. Each helper corresponds to a specific model provider.

#### `aidb.embeddings_config`

For `openai_embeddings` and any OpenAI-compatible embeddings endpoint.

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	Model identifier.
<i>api_key</i>	TEXT	NULL	API key.
<i>url</i>	TEXT	NULL	Endpoint URL override.
<i>basic_auth</i>	TEXT	NULL	Basic auth ( <i>user:password</i> ).
<i>max_concurrent_requests</i>	INTEGER	NULL	Max concurrent requests.
<i>max_batch_size</i>	INTEGER	NULL	Max inputs per batch.
<i>input_type</i>	TEXT	NULL	Input type hint (provider-specific).
<i>input_type_query</i>	TEXT	NULL	Query input type hint (provider-specific).
<i>tls_config</i>	JSONB	NULL	TLS configuration.
<i>is_hcp_model</i>	BOOLEAN	NULL	<i>true</i> if model runs on HCP.

#### Example

```
SELECT aidb.create_model(
 my_embedder,
 openai_embeddings,
 config => aidb.embeddings_config(
 model => text-embedding-3-small,
 api_key => sk-...,
 url => https://api.openai.com/v1
)
);
```

#### `aidb.completions_config`

For `openai_completions` and any OpenAI-compatible completions endpoint.

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	Model identifier.
<i>api_key</i>	TEXT	NULL	API key.
<i>url</i>	TEXT	NULL	Endpoint URL override.
<i>basic_auth</i>	TEXT	NULL	Basic auth ( <i>user:password</i> ).
<i>system_prompt</i>	TEXT	NULL	Default system prompt.
<i>temperature</i>	DOUBLE PRECISION	NULL	Sampling temperature.
<i>top_p</i>	DOUBLE PRECISION	NULL	Nucleus sampling threshold.
<i>seed</i>	BIGINT	NULL	Random seed.

continues on next page

Table 23 – continued from previous page

Parameter	Type	Default	Description
<i>thinking</i>	BOOLEAN	NULL	<think> tag handling.
<i>max_tokens</i>	JSONB	NULL	Max tokens config (use :ref:`aidb.max_tokens_config()` <#aidbmax_tokens_config>` ).
<i>max_concurrent_requests</i>	INTEGER	NULL	Max concurrent requests.
<i>extra_args</i>	JSONB	NULL	Additional provider-specific arguments.
<i>is_hcp_model</i>	BOOLEAN	NULL	<i>true</i> if model runs on HCP.

### Example

```
SELECT aidb.create_model(
 my_llm,
 openai_completions,
 config => aidb.completions_config(
 model => gpt-4o,
 api_key => sk-...,
 temperature => 0.2
)
);
```

!!!note For agentic use with reliable, native tool calling on OpenAI models, use [aidb.openai\\_responses\\_config](#) instead — see [Tool calling and structured output](#) for how the two providers differ, and [Agents reference](#) for building an agent around either one.

### aidb.max\_tokens\_config

Builds the `max_tokens` object for use with `aidb.completions_config()` .

Parameter	Type	Default	Description
<i>size</i>	INTEGER	Required	Maximum tokens to generate.
<i>format</i>	TEXT	NULL	Format: 'default', 'legacy', or 'both'.

### Example

```
SELECT aidb.completions_config(
 model => gpt-4o,
 max_tokens => aidb.max_tokens_config(size => 1024, format => default)
);
```

### aidb.bert\_config

For the `bert_local` provider.

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	HuggingFace model identifier.
<i>revision</i>	TEXT	NULL	Model revision or branch.
<i>cache_dir</i>	TEXT	NULL	Local cache directory.

### Example

```
SELECT aidb.create_model(my_bert, bert_local,
 config => aidb.bert_config(sentence-transformers/all-MiniLM-L6-v2));
```

### aidb.clip\_config

For the `clip_local` provider.

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	HuggingFace model identifier.
<i>revision</i>	TEXT	NULL	Model revision or branch.
<i>cache_dir</i>	TEXT	NULL	Local cache directory.
<i>image_size</i>	INTEGER	NULL	Input image size in pixels.

### Example

```
SELECT aidb.create_model(my_clip, clip_local,
 config => aidb.clip_config(openai/clip-vit-base-patch32));
```

### aidb.llama\_config

For the `llama_instruct_local` provider.

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	Model identifier or HuggingFace path.
<i>revision</i>	TEXT	NULL	Model revision.
<i>cache_dir</i>	TEXT	NULL	Local cache directory.
<i>model_path</i>	TEXT	NULL	Explicit local path to model weights (overrides <i>model</i> ).
<i>system_prompt</i>	TEXT	NULL	Default system prompt.
<i>temperature</i>	DOUBLE	PRECISION	Sampling temperature.
<i>top_p</i>	DOUBLE	PRECISION	Nucleus sampling threshold.
<i>seed</i>	BIGINT	NULL	Random seed.
<i>sample_len</i>	INTEGER	NULL	Max tokens to generate.
<i>repeat_penalty</i>	REAL	NULL	Repetition penalty.

continues on next page

Table 27 – continued from previous page

Parameter	Type	Default	Description
<i>repeat_last_n</i>	INTEGER	NULL	Tokens considered for repetition penalty.
<i>use_flash_attention</i>	BOOLEAN	NULL	Enable flash attention.
<i>use_kv_cache</i>	BOOLEAN	NULL	Enable KV cache.

**Example**

```
SELECT aidb.create_model(my_llama, llama_instruct_local,
 config => aidb.llama_config(
 meta-llama/Llama-3.2-3B-Instruct,
 temperature => 0.5
)
);
```

**aidb.t5\_config**

For the `t5_local` provider.

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	Model identifier or HuggingFace path.
<i>revision</i>	TEXT	NULL	Model revision.
<i>model_path</i>	TEXT	NULL	Explicit local path to model weights.
<i>cache_dir</i>	TEXT	NULL	Local cache directory.
<i>temperature</i>	DOUBLE PRECISION	NULL	Sampling temperature.
<i>top_p</i>	DOUBLE PRECISION	NULL	Nucleus sampling threshold.
<i>seed</i>	BIGINT	NULL	Random seed.
<i>max_tokens</i>	INTEGER	NULL	Max tokens to generate.
<i>repeat_penalty</i>	REAL	NULL	Repetition penalty.
<i>repeat_last_n</i>	INTEGER	NULL	Tokens considered for repetition penalty.

**Example**

```
SELECT aidb.create_model(my_t5, t5_local,
 config => aidb.t5_config(google/flan-t5-base));
```

**aidb.gemini\_config**

For the `gemini` provider.

Parameter	Type	Default	Description
<i>api_key</i>	TEXT	Required	Google API key.
<i>model</i>	TEXT	NULL	Gemini model identifier (for example, <i>gemini-2.0-flash</i> ).
<i>url</i>	TEXT	NULL	API endpoint override.
<i>max_concurrent_requests</i>	INTEGER	NULL	Max concurrent requests.
<i>thinking_budget</i>	INTEGER	NULL	Extended thinking token budget (Gemini 2.x only).

### Example

```
SELECT aidb.create_model(my_gemini, gemini,
 config => aidb.gemini_config(AIza..., model => gemini-2.0-flash));
```

!!!note Unlike other remote providers, Gemini's `api_key` is a plain field of `config` rather than being passed through `aidb.create_model()`'s separate `credentials` parameter. It's stored as part of the model's config rather than the more restricted credentials store — keep that in mind if `aidb.models.options` is visible to more roles than you want to see the key.

### aidb.nim\_clip\_config

For the `nim_clip` provider (NVIDIA NIM multimodal embeddings).

Parameter	Type	Default	Description
<i>api_key</i>	TEXT	NULL	NIM API key.
<i>model</i>	TEXT	NULL	NIM CLIP model identifier.
<i>url</i>	TEXT	NULL	NIM endpoint URL override.
<i>basic_auth</i>	TEXT	NULL	Basic auth credentials.
<i>is_hcp_model</i>	BOOLEAN	NULL	<i>true</i> if model runs on HCP.

### Example

```
SELECT aidb.create_model(my_nim_clip, nim_clip,
 config => aidb.nim_clip_config(api_key => nvapi-..., model => nvidia/nvclip));
```

### aidb.nim\_ocr\_config

For the `nim_ocr` provider (NVIDIA NIM OCR).

Parameter	Type	Default	Description
<i>api_key</i>	TEXT	NULL	NIM API key.
<i>model</i>	TEXT	NULL	NIM OCR model identifier.
<i>url</i>	TEXT	NULL	NIM endpoint URL override.
<i>basic_auth</i>	TEXT	NULL	Basic auth credentials.
<i>is_hcp_model</i>	BOOLEAN	NULL	<i>true</i> if model runs on HCP.

### Example

```
SELECT aidb.create_model(my_nim_ocr, nim_ocr,
 config => aidb.nim_ocr_config(api_key => nvapi-...));
```

### aidb.nim\_reranking\_config

For the `nim_reranking` provider.

Parameter	Type	Default	Description
<i>api_key</i>	TEXT	NULL	NIM API key.
<i>model</i>	TEXT	NULL	NIM reranking model identifier.
<i>url</i>	TEXT	NULL	NIM endpoint URL override.
<i>basic_auth</i>	TEXT	NULL	Basic auth credentials.
<i>is_hcp_model</i>	BOOLEAN	NULL	<i>true</i> if model runs on HCP.

### Example

```
SELECT aidb.create_model(my_reranker, nim_reranking,
 config => aidb.nim_reranking_config(
 api_key => nvapi-...,
 model => nvidia/nv-rerankqa-mistral-4b-v3
)
);
```

### aidb.openrouter\_chat\_config

For the `openrouter_chat` provider.

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	OpenRouter model identifier.
<i>api_key</i>	TEXT	NULL	OpenRouter API key.
<i>url</i>	TEXT	NULL	API endpoint override.
<i>max_concurrent_requests</i>	INTEGER	NULL	Max concurrent requests.
<i>max_tokens</i>	JSONB	NULL	Max tokens config (use <code>:ref:``aidb.max_tokens_config()``` &lt;#aidbmax_tokens_config&gt;` ).</code>

### Example

```
SELECT aidb.create_model(my_or_chat, openrouter_chat,
 config => aidb.openrouter_chat_config(
 anthropic/claude-3-5-haiku,
 api_key => sk-or-...
)
);
```

**aidb.openrouter\_embeddings\_config**

For the `openrouter_embeddings` provider.

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	OpenRouter embeddings model identifier.
<i>api_key</i>	TEXT	NULL	OpenRouter API key.
<i>url</i>	TEXT	NULL	API endpoint override.
<i>max_concurrent_requests</i>	INTEGER	NULL	Max concurrent requests.
<i>max_batch_size</i>	INTEGER	NULL	Max inputs per batch.

**Example**

```
SELECT aidb.create_model(my_or_embedder, openrouter_embeddings,
 config => aidb.openrouter_embeddings_config(
 mistral/mistral-embed,
 api_key => sk-or-...
)
);
```

**aidb.openai\_responses\_config**

For the `openai_responses` and `openai_responses_azure` providers.

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	Model identifier (for example, <i>gpt-5.1</i> ).
<i>api_key</i>	TEXT	NULL	API key for authentication.
<i>basic_auth</i>	TEXT	NULL	Basic auth credentials ( <i>user:password</i> ).
<i>url</i>	TEXT	NULL	API endpoint URL. Defaults to OpenAI's Responses endpoint for <i>openai_responses</i> ; required for <i>openai_responses_azure</i> (no universal default — use your resource's endpoint).
<i>max_concurrent_requests</i>	INTEGER	25	Maximum concurrent requests to the endpoint.
<i>system_prompt</i>	TEXT	NULL	Default system prompt, sent via the Responses API's native <i>instructions</i> field.
<i>temperature</i>	DOUBLE PRECISION	NULL	Sampling temperature.
<i>max_output_tokens</i>	INTEGER	NULL	Maximum tokens to generate.
<i>top_p</i>	DOUBLE PRECISION	NULL	Nucleus sampling threshold.

continues on next page

Table 35 – continued from previous page

Parameter	Type	Default	Description
<i>extra_args</i>	JSONB	NULL	Additional provider-specific arguments passed directly to the API.

## Examples

```
-- OpenAI
SELECT aidb.create_model(
 my_gpt,
 openai_responses,
 config => aidb.openai_responses_config(model => gpt-5.1),
 credentials => {"api_key": "sk-..."}::JSONB
);

-- Azure AI Foundry
SELECT aidb.create_model(
 my_gpt_azure,
 openai_responses_azure,
 config => aidb.openai_responses_config(
 model => gpt-5.1,
 url => https://<resource>.openai.azure.com/openai/v1/responses
),
 credentials => {"api_key": "..."}::JSONB
);
```

This provider natively supports the `tools`, `tool_choice`, and `response_format` fields described in [Tool calling and structured output](#).

## aidb.anthropic\_messages\_config

For the `anthropic_messages`, `anthropic_messages_azure`, and `anthropic_messages_bedrock` providers.

Parameter	Type	Default	Description
<i>model</i>	TEXT	Required	Model identifier (for example, <i>claude-opus-4-6</i> , or a Bedrock model ID for the Bedrock variant).
<i>api_key</i>	TEXT	NULL	API key for authentication. For <i>anthropic_messages_bedrock</i> , this is a Bedrock API key (bearer token), not an AWS SigV4 credential.
<i>basic_auth</i>	TEXT	NULL	Basic auth credentials ( <i>user:password</i> ).

continues on next page

Table 36 – continued from previous page

Parameter	Type	Default	Description
<i>url</i>	TEXT	NULL	API endpoint URL. Defaults to <i>api.anthropic.com</i> for <i>anthropic_messages</i> . Required for <i>anthropic_messages_azure</i> (your resource's Messages endpoint) and <i>anthropic_messages_bedrock</i> (the regional <i>bedrock-runtime</i> base endpoint — the model ID is appended automatically).
<i>max_concurrent_requests</i>	INTEGER	25	Maximum concurrent requests to the endpoint.
<i>system_prompt</i>	TEXT	NULL	Default system prompt, sent via the Messages API's native <i>system</i> field.
<i>temperature</i>	DOUBLE PRECISION	NULL	Sampling temperature.
<i>max_tokens</i>	INTEGER	4096	Maximum tokens to generate. Anthropic's Messages API requires this field; AIDB defaults it when omitted.
<i>top_p</i>	DOUBLE PRECISION	NULL	Nucleus sampling threshold.
<i>extra_args</i>	JSONB	NULL	Additional provider-specific arguments passed directly to the API.

## Examples

```
-- Anthropic direct
SELECT aidb.create_model(
 my_claude,
 anthropic_messages,
 config => aidb.anthropic_messages_config(model => claude-opus-4-6),
 credentials => {"api_key": "sk-ant-..."}::JSONB
);

-- Azure AI Foundry
SELECT aidb.create_model(
 my_claude_azure,
 anthropic_messages_azure,
 config => aidb.anthropic_messages_config(
 model => claude-opus-4-6,
 url => https://<resource>.services.ai.azure.com/anthropic/v1/messages
),
 credentials => {"api_key": "..."}::JSONB
);

-- AWS Bedrock
SELECT aidb.create_model(
 my_claude_bedrock,
 anthropic_messages_bedrock,
 config => aidb.anthropic_messages_config(
 model => anthropic.claude-opus-4-6-v1:0,
```

(continues on next page)

(continued from previous page)

```

url => https://bedrock-runtime.us-east-1.amazonaws.com
),
credentials => {"api_key": "..."}::JSONB
);

```

This provider natively supports the `tools`, `tool_choice`, and `response_format` fields described in [Tool calling and structured output](#). Anthropic’s Messages API has no native structured-output field, so `response_format` is implemented as a forced call to a synthetic tool — functionally equivalent, but it means `response_format` and `tools/tool_choice` can’t be used in the same call.

## 15.2 Knowledge bases reference

Reference for knowledge base (KB) functions and views. For guide-style documentation, see [Knowledge bases reference](#).

### 15.2.1 Catalog views

#### `aidb.knowledge_bases`

Also accessible as `aidb.kbs`. Lists all knowledge bases in the database with their full configuration.

Column	Type	Description
<i>id</i>	integer	Internal identifier
<i>name</i>	text	Knowledge base name
<i>vector_schema</i>	text	Schema of the embeddings table
<i>vector_table</i>	text	Embeddings storage table
<i>model_name</i>	text	Embedding model used
<i>distance_operator</i>	<a href="#"><code>aidb.DistanceOperator</code></a>	Vector distance function used for retrieval
<i>distance_operator_sql</i>	text	The distance operator in SQL-operator syntax (for example, <code>&lt;-&gt;</code> , <code>&lt;=&gt;</code> )
<i>vector_data_column</i>	text	Embeddings column
<i>vector_key_column</i>	text	Key column in the embeddings table
<i>vector_index</i>	jsonb	Vector index configuration
<i>pipeline_ids</i>	integer[]	IDs of all pipelines attached to this knowledge base
<i>pipeline_names</i>	text[]	Names of all pipelines attached to this knowledge base

#### `aidb.knowledge_base_metrics`

Also accessible as `aidb.kbm`. Aggregated stats for each knowledge base — pipeline count, embedding row count, and the worst pipeline status. For per-pipeline progress (backlog, errors, source / destination counts), see [aidb.pipeline\\_metrics](#).

Column	Type	Description
<i>name</i>	text	Knowledge base name ( <i>schema.vector_table</i> ).
<i>pipelines</i>	integer	Number of pipelines attached to this KB.
<i>embeddings</i>	bigint	Row count of the vector table.

continues on next page

Table 38 – continued from previous page

Column	Type	Description
<i>status</i>	text	Worst status across all attached pipelines (see <i>AI pipelines</i> ).

**Example**

```
SELECT * FROM aidb.kbm;
```

```
__OUTPUT__
```

name	pipelines	embeddings	status
<i>public</i> .product_catalog_kb	1	1842	UpToDate
<i>public</i> .customer_feedback_kb	2	915	UpToDate

(2 rows)

The legacy view names `aidb.knowledge_base_stats` and `aidb.kbstat` are kept as aliases of `aidb.knowledge_base_metrics` for backward compatibility.

**15.2.2 Types****aidb.DistanceOperator**

Vector distance function used during retrieval.

Value	Description
<i>L2</i>	Euclidean distance
<i>InnerProduct</i>	Inner product
<i>Cosine</i>	Cosine similarity
<i>L1</i>	L1 distance
<i>Hamming</i>	Hamming distance
<i>Jaccard</i>	Jaccard distance

**aidb.PipelineDataFormat**

Format of data in a pipeline source or volume.

Value	Description
<i>Text</i>	Plain text
<i>Image</i>	Binary image data
<i>Pdf</i>	PDF documents

**aidb.PipelineAutoProcessingMode**

Auto-processing mode for a knowledge base pipeline.

Value	Description
<i>Live</i>	Immediate processing via Postgres triggers on each data change
<i>Background</i>	Periodic processing via a Postgres background worker
<i>Disabled</i>	No automatic processing; run manually with <i>aidb.run_pipeline()</i>

**15.2.3 Knowledge base functions**

To change a pipeline's auto-processing mode, use *aidb.update\_pipeline* .

**aidb.retrieve\_key**

Returns the source record keys and distances for the top matching embeddings, without fetching source data. In *part\_ids* , also returns the IDs of the matching parts for each step in a multi-step pipeline. For example, if a single source record is split into 10 parts by a *ChunkText* step, then *part\_ids* indicates which chunk matched.

**Parameters**

Parameter	Type	Default	Description
<i>knowledge_base_name</i>	TEXT	Required	Name of the knowledge base ( <i>schema.vector_table</i> ).
<i>query</i>	TEXT or BYTEA	Required	Query text or image bytes. Must match the data format of the embedding model.
<i>topk</i>	INTEGER	<i>1</i>	Number of results to return. Must be a positive integer — <i>0</i> or a negative value raises an error.
<i>deduplicate</i>	BOOLEAN	<i>true</i>	Return each source result only once even if multiple "parts" (for example, chunks/pages) match. See <i>Deduplication and intermediate results</i> .

**Returns**

Column	Type	Description
<i>key</i>	text	Source record key
<i>distance</i>	double precision	Vector distance from the query
<i>part_ids</i>	bigint[]	Which parts (for example, chunk or page) of the source record produced the matching vector
<i>pipeline_name</i>	text	Name of the pipeline that supplied this row. Useful when the KB has multiple attached pipelines.

Example

```
SELECT * FROM aidb.retrieve_key(public.customer_feedback_kb, late delivery, topk=>3);

__OUTPUT__
key | distance | part_ids | pipeline_name
-----+-----+-----+-----
1042 | 1.1931772046185758 | {0,0} | feedback_eu_pipe
318 | 1.1980633963685476 | {2,0} | feedback_na_pipe
1057 | 1.2150919866080878 | {2,0} | feedback_eu_pipe
(3 rows)
```

aidb.retrieve\_text

Returns source text and distances for the top matching embeddings by joining the embeddings table with the source table. In `part_ids`, also returns the IDs of the matching parts for each step in a multi-step pipeline. For example, if a single source record is split into 10 parts by a `ChunkText` step, then `part_ids` indicates which chunk matched. The return column `intermediate_steps` contains the actual intermediate step output.

Parameters

Parameter	Type	Default	Description
<i>knowledge_base_name</i>	TEXT	Required	Name of the knowledge base ( <i>schema.vector table</i> ).
<i>query</i>	TEXT or BYTEA	Required	Query text or image bytes. Must match the data format of the embedding model.
<i>topk</i>	INTEGER	<i>1</i>	Number of results to return. Must be a positive integer — <i>0</i> or a negative value raises an error.
<i>deduplicate</i>	BOOLEAN	<i>true</i>	Return each source result only once even if multiple "parts" (for example, chunks/pages) match. See <i>Deduplication and intermediate results</i> .

Returns

Column	Type	Description
<i>key</i>	text	Source record key
<i>value</i>	text	Source text content. <i>NULL</i> when the original source content is non-text (for example, PDF or Image); other columns still surface match metadata.
<i>distance</i>	double precision	Vector distance from the query
<i>part_ids</i>	bigint[]	Which part of the source record matched (for example, a chunk or page)
<i>pipeline_name</i>	text	The name of the pipeline processing this source record

continues on next page

Table 45 – continued from previous page

Column	Type	Description
<i>intermediate_steps</i>	jsonb	The matching results from any step configured with <i>intermediate_destination</i> — that is, the values belonging to <i>part_ids</i> . Empty ( <code>[]</code> ) if no step in the pipeline has intermediate storage enabled. See <i>End-to-end example: Multi-step pipeline with intermediate storage</i> .

*intermediate\_steps* is a JSON array with one entry per step that has intermediate storage enabled, each shaped like:

```
{ "value": "...", "operation": "ChunkText", "step_order": 1, "destination_table":
 ↳ "public.pipeline_my_pipeline_step_1" }
```

### Example

```
SELECT * FROM aidb.retrieve_text(my_kb, waterproof jacket, 3);
```

```
__OUTPUT__
key | value | distance
-----+-----+-----
19337 | Men Stripes Waterproof Shell Jacket | 0.2994317672742334
55018 | Women All-Weather Hiking Anorak | 0.3804609668507203
(2 rows)
```

### aidb.delete\_knowledge\_base

Cascades: deletes all pipelines attached to the knowledge base, drops the vector table, and removes the registry entry. Doesn't delete underlying source tables.

To remove a single pipeline from a multi-pipeline KB while keeping the KB and the other pipelines' vectors intact, use *aidb.delete\_pipeline* instead. Deleting the last attached pipeline also drops the KB.

### Parameters

Parameter	Type	Description
<i>knowledge_base_name</i>	TEXT	Knowledge base identifier ( <i>schema.vector_table</i> ).

### Example

```
SELECT aidb.delete_knowledge_base(public.customer_feedback_kb);
```

## 15.2.4 Volume management functions

### aidb.create\_volume

Creates an AIDB volume from a PGFS storage location. The volume is a foreign table that exposes the storage location's contents as rows and can be used as a knowledge base data source or queried directly with SQL. See [Configuring external storage](#) for setting up the PGFS storage location first.

#### Parameters

Parameter	Type	Description
<i>name</i>	TEXT	Name for the volume. Must be a valid unquoted PostgreSQL identifier.
<i>server_name</i>	TEXT	Name of the PGFS storage location.
<i>path</i>	TEXT	Optional sub-path within the storage location. Canonicalized to a single trailing slash (for example, <i>foo</i> , <i>/foo,foo/</i> , <i>/foo/</i> all store as <i>foo/</i> ; <i>NULL</i> or <i>/</i> store as <i>/</i> ).
<i>data_format</i>	TEXT	Data format: <i>Text</i> , <i>Image</i> , or <i>Pdf</i> .

#### Example

```
SELECT aidb.create_volume(pdf_volume, my_s3_location, /, Pdf);
```

### aidb.volumes

View that lists all AIDB volumes in the database. Returns *schema*, *volume*, *storage\_location*, and *path*.

#### Example

```
SELECT * FROM aidb.volumes;
```

### aidb.delete\_volume

Deletes an AIDB volume.

!!!note Deleting the underlying PGFS storage location also deletes all volumes built on top of it.

#### Parameters

Parameter	Type	Description
<i>volume_name</i>	TEXT	Name of the volume to delete.

#### Example

```
SELECT aidb.delete_volume(pdf_volume);
```

## 15.3 Operations reference

Reference for pipeline error log functions. For guide-style documentation, see *Pipeline error log*.

### 15.3.1 Error log functions

Each pipeline maintains an error log table at `{source_schema}.pipeline_{pipeline_name}_errors`. These functions provide a consistent interface for querying and managing that log.

#### **aidb.get\_error\_logs**

Returns error log entries for a pipeline, with optional filters.

#### Parameters

Parameter	Type	Default	Description
<i>p_pipeline_name</i>	TEXT	Required	Name of the pipeline.
<i>p_source_id</i>	TEXT	NULL	Filter to a specific source record ID.
<i>p_pipeline_step</i>	SMALLINT	NULL	Filter to a specific pipeline step number.
<i>p_error_category</i>	<i>aidb.ErrorBlocking</i>	NULL	Filter by category: <i>RecordTemporary</i> , <i>RecordPermanent</i> , <i>PipelineTemporary</i> , or <i>PipelinePermanent</i> .
<i>p_limit</i>	INTEGER	NULL	Maximum number of rows to return.
<i>p_offset</i>	INTEGER	NULL	Number of rows to skip (for pagination).

#### Returns

Column	Type	Description
<i>id</i>	bigint	Error log entry ID
<i>source_id</i>	text	Source record that failed
<i>part_ids</i>	bigint[]	Part identifiers (for chunked records)
<i>pipeline_step</i>	smallint	Step number where the failure occurred
<i>step_operation</i>	<i>aidb.PipelineStepOperation</i>	Operation name (for example, <i>ParsePdf</i> , <i>ChunkText</i> )
<i>error_message</i>	text	Error description
<i>error_category</i>	<i>aidb.ErrorBlocking</i>	Error category
<i>failed_at</i>	timestampz	Timestamp of the failure
<i>retry_count</i>	integer	Number of retry attempts
<i>last_retry_at</i>	timestampz	Timestamp of the last retry
<i>last_seen_at</i>	timestampz	Timestamp of the most recent occurrence of this error. Updates on every repeat; <i>failed_at</i> stays fixed at the first occurrence.

Examples

```
- - All errors for a pipeline
SELECT * FROM aidb.get_error_logs(my_pipeline);

- - Filter by source record
SELECT * FROM aidb.get_error_logs(my_pipeline, p_source_id => doc_42);

- - Filter by step and category
SELECT * FROM aidb.get_error_logs(
 my_pipeline,
 p_pipeline_step => 1::SMALLINT,
 p_error_category => RecordPermanent
);

- - Paginate
SELECT * FROM aidb.get_error_logs(my_pipeline, p_limit => 20, p_offset => 40);
```

aidb.get\_error\_log\_summary

Returns aggregated error counts for a single pipeline, grouped by step, operation, and category.

Parameters

Parameter	Type	Description
<i>p_pipeline_name</i>	TEXT	Name of the pipeline.

Returns

Column	Type	Description
<i>pipeline_step</i>	smallint	Step number
<i>step_operation</i>	<i>aidb.PipelineStepOperation</i>	Operation name
<i>error_category</i>	<i>aidb.ErrorBlocking</i>	Error category
<i>error_count</i>	bigint	Number of errors in this group
<i>latest_failed_at</i>	timestampz	Most recent failure timestamp

Example

```
SELECT * FROM aidb.get_error_log_summary(my_pipeline);

__OUTPUT__
pipeline_step | step_operation | error_category | error_count | latest_failed_at
-----+-----+-----+-----+-----
1 | ParsePdf | RecordPermanent | 2 | 2026-04-10 14:31:58.
456+00
```

(continues on next page)

(continued from previous page)

```

2 | ChunkText | RecordPermanent | 1 | 2026-04-10 14:32:01.
↪123+00
(2 rows)

```

## aidb.get\_all\_error\_summaries

Returns the same rollup as `aidb.get_error_log_summary()` but across all pipelines at once.

### Returns

Column	Type	Description
<i>pipeline_name</i>	text	Pipeline name
<i>pipeline_step</i>	smallint	Step number
<i>step_operation</i>	<i>aidb.PipelineStepOperation</i>	Operation name
<i>error_category</i>	<i>aidb.ErrorBlocking</i>	Error category
<i>error_count</i>	bigint	Number of errors in this group
<i>latest_failed_at</i>	timestampz	Most recent failure timestamp

### Example

```
SELECT * FROM aidb.get_all_error_summaries();
```

```

__OUTPUT__
pipeline_name | pipeline_step | step_operation | error_category | error_count |
↪latest_failed_at
-----+-----+-----+-----+-----+
↪-----
my_pipeline | 1 | ParsePdf | RecordPermanent | 2 |
↪2026-04-10 14:31:58.456+00
pdf_ingester | 1 | ParsePdf | PipelinePermanent | 1 |
↪2026-04-10 15:00:12.000+00
(2 rows)

```

## aidb.clear\_error\_logs

Deletes specific error log entries by ID. Review entries with `aidb.get_error_logs()` before clearing.

### Parameters

Parameter	Type	Description
<i>p_pipeline_name</i>	TEXT	Name of the pipeline.
<i>p_error_ids</i>	BIGINT[]	Array of error log entry IDs to delete.

## Returns

BIGINT — the number of rows deleted.

## Example

```
SELECT aidb.clear_error_logs(my_pipeline, ARRAY[1, 3]);
```

```
__OUTPUT__
clear_error_logs

 2
(1 row)
```

## aidb.requeue\_pipeline\_errors

Introduced in 7.5.0. Deletes the targeted record-level error log entries and marks their source records dirty on the pipeline's state surface. The next normal run re-processes them; the call itself does no processing. See [Requeuing failed records](#) for the workflow.

## Parameters

Parameter	Type	Description
<i>p_pipeline_name</i>	TEXT	Name of the pipeline.
<i>p_error_ids</i>	BIGINT[]	Array of error log entry IDs to requeue.

## Returns

One row per input ID, ordered by ascending `error_id`.

Column	Type	Description
<i>error_id</i>	bigint	Error log entry ID from the input array.
<i>action</i>	text	<i>requeued</i> , <i>skipped_pipeline_level</i> , or <i>not_found</i> (see below).

- *requeued* : record-level row deleted and source marked dirty. Table sources enqueue a state event; volume sources set a one-shot `retry_requested_at` marker.
- *skipped\_pipeline\_level* : entry has a NULL `source_id` and is left in place. Fix the underlying issue and re-run, or dismiss with `aidb.clear_error_logs`.
- *not\_found* : ID not present in the error log, or the pipeline has no error log table.

Concurrent requeues on overlapping IDs serialize on per-row `FOR UPDATE` locks; non-overlapping ID sets proceed in parallel. Background pipelines drain requeued rows automatically on the next worker tick. Disabled or Live pipelines need a follow-up `aidb.run_pipeline` call.

## Example

```
SELECT * FROM aidb.requeue_pipeline_errors(my_pipeline, ARRAY[1, 2, 99]);
```

\_\_OUTPUT\_\_

```
error_id | action
-----+-----
 1 | requeued
 2 | requeued
 99 | not_found
(3 rows)
```

### 15.3.2 Error categories

Category	Scope	Blocks pipeline?	Description
<i>RecordTemporary</i>	Single record	No	Transient failure for one record. May succeed on retry.
<i>RecordPermanent</i>	Single record	No	Permanent failure for one record (for example, corrupt data).
<i>PipelineTemporary</i>	Entire step	Yes	Transient failure that stopped the step (for example, service outage). May resolve on retry.
<i>PipelinePermanent</i>	Entire step	Yes	Permanent failure that stopped the step (for example, invalid model config).

## 15.4 Functions reference

Reference for AIDB standalone SQL functions that transform data directly in queries, without requiring a pipeline. For guide-style documentation and usage examples, see [SQL functions](#).

For AI inference functions (`encode_text`, `generate_text`, `rerank_text`, and related), see [Inference functions](#).

### 15.4.1 `aidb.chunk_text`

Divides a text string into smaller, semantically coherent segments.

#### Parameters

Parameter	Type	Description
<i>input</i>	TEXT	The text to chunk.
<i>options</i>	JSONB	Chunking configuration, built with <code>aidb.chunk_text_config()</code> (see below).

#### Options

Key	Type	Default	Description
<i>desired_length</i>	integer	Required	Target segment size. Acts as a strict upper limit if <i>max_length</i> is omitted.
<i>max_length</i>	integer	NULL	Upper bound for chunk size. Chunks extend past <i>desired_length</i> only to preserve semantic boundaries.
<i>overlap_length</i>	integer	0	Amount of content to repeat between consecutive chunks, to preserve cross-boundary context.
<i>strategy</i>	text	'chars'	Chunking unit: 'chars' (character-based) or 'words' (word-based). Determines the unit for <i>desired_length</i> , <i>max_length</i> , and <i>overlap_length</i> .

## Returns

Column	Type	Description
<i>part_id</i>	bigint	Zero-based segment index.
<i>chunk</i>	text	The text segment.

## Example

```
SELECT * FROM aidb.chunk_text (
 input => Long text here...,
 options => {"desired_length": 120, "max_length": 150}
);
```

## 15.4.2 aidb.parse\_html

Extracts readable text from an HTML string, stripping tags while preserving structure.

### Parameters

Parameter	Type	Description
<i>html</i>	TEXT	The HTML string to parse.
<i>options</i>	JSONB	Parsing configuration (see below).

### Options

Key	Type	Default	Description
<i>method</i>	text	'StructuredPlaintext'	Parsing method: 'StructuredPlaintext' (plain text extraction) or 'StructuredMarkdown' (Markdown-like output that retains headers and lists).

## Returns

TEXT — the extracted text content.

## Example

```
SELECT aidb.parse_html (
 html => <h1>Hello</h1><p>World</p>,
 options => {"method": "StructuredPlaintext"}
);
```

### 15.4.3 aidb.parse\_pdf

Extracts text from binary PDF data. Returns one row per page.

## Parameters

Parameter	Type	Description
<i>bytes</i>	BYTEA	Raw PDF binary data.
<i>options</i>	JSONB	Parsing configuration (see below).

## Options

Key	Type	Default	Description
<i>method</i>	text	'Structured'	Parsing method. Currently 'Structured' (spec-based text block extraction).
<i>allow_partial_parsing</i>	boolean	true	When true, continues parsing when errors are encountered on individual pages, returning as much data as possible.

## Returns

Column	Type	Description
<i>part_id</i>	bigint	Page index (zero-based) from which the text was extracted.
<i>text</i>	text	Extracted text for that page.

### Example

```
SELECT * FROM aidb.parse_pdf (
 bytes => pg_read_binary_file (/path/to/doc.pdf) ::BYTEA,
 options => {"allow_partial_parsing": true}
);
```

## 15.4.4 aidb.perform\_ocr

Extracts text from image data using a registered OCR-capable model.

### Parameters

Parameter	Type	Description
<i>input</i>	BYTEA	Raw binary image data.
<i>options</i>	JSONB	OCR configuration (see below).

### Options

Key	Type	Description
<i>model</i>	text	Name of a registered OCR-capable model (for example, one using the <i>nim_paddle_ocr</i> or <i>llamacpp_ocr</i> provider).

### Returns

Column	Type	Description
<i>part_id</i>	bigint	Text block index. A single image may produce multiple rows if the provider returns multiple text segments.
<i>text</i>	text	Extracted text for that block.

### Example

```
SELECT * FROM aidb.perform_ocr (
 input => pg_read_binary_file (/path/to/image.png) ::BYTEA,
 options => {"model": "my_ocr_model"}
);
```

## 15.4.5 aidb.summarize\_text

Generates a concise summary of a text string using a registered language model.

## Parameters

Parameter	Type	Description
<i>input</i>	TEXT	The text to summarize.
<i>options</i>	JSONB	Summarization configuration (see below).

## Options

Key	Type	Default	Description
<i>model</i>	text	Required	Name of a registered model that supports <i>generate_text</i> .
<i>prompt</i>	text	(standard prompt)	Custom instruction to guide the summary style, for example, 'Summarize for a 5th grader'.
<i>chunk_config</i>	JSONB	NULL	Chunking configuration to apply before summarization when input exceeds the model's context window. Accepts the same keys as <i>aidb.chunk_text</i> options.
<i>strategy</i>	text	'append'	Summarization strategy: 'append' (summarize each chunk independently and concatenate) or 'reduce' (iteratively summarize until the desired length is reached).
<i>reduction_factor</i>	integer	3	Used with the 'reduce' strategy. Controls how aggressively each iteration reduces the text.

## Returns

TEXT — the generated summary.

## Example

```
SELECT aidb.summarize_text(
 input => Long article text here...,
 options => {"model": "my_t5_model"}
);
```

### 15.4.6 aidb.summarize\_text\_aggregate

Aggregate version of `aidb.summarize_text`. Accumulates text from all rows in each group, then sends the combined result to the LLM for summarization. Returns one summary per `GROUP BY` group. Empty and NULL rows are skipped.

## Parameters

Parameter	Type	Required	Description
<i>input</i>	TEXT	Yes	Text column from each row. Empty/NULL rows are skipped.
<i>options</i>	JSON	Yes	Configuration object built with <i>aidb.summarize_text_config()</i> . Must contain <i>model</i> at minimum.

## Options

Accepts the same options as `aidb.summarize_text` (see above). `options` is **required** for the aggregate, not optional.

## Returns

TEXT — the summary for the group.

## Example

```
SELECT category,
 aidb.summarize_text_aggregate(
 text_column,
 aidb.summarize_text_config(my_t5_model)::json ORDER BY id
) AS summary
FROM my_table
GROUP BY category;
```

## 15.4.7 Parameter defaults quick reference

Function	Parameter	SQL default	Runtime default
<i>summarize_text</i>	<i>options</i>	'{'	Must include <i>model</i>
<i>summarize_text_aggregate</i>	<i>input</i>	— (required)	—
<i>summarize_text_aggregate</i>	<i>options</i>	— (required)	—
<i>summarize_text_config</i>	<i>model</i>	— (required)	—
<i>summarize_text_config</i>	<i>chunk_config</i>	NULL	No chunking
<i>summarize_text_config</i>	<i>prompt</i>	NULL	Standard summarize prompt
<i>summarize_text_config</i>	<i>strategy</i>	NULL	'append'
<i>summarize_text_config</i>	<i>reduction_factor</i>	NULL	3
<i>summarize_text_config</i>	<i>inference_config</i>	NULL	Provider defaults
<i>chunk_text_config</i>	<i>desired_length</i>	— (required)	—
<i>chunk_text_config</i>	<i>max_length</i>	NULL	Same as <i>desired_length</i>
<i>chunk_text_config</i>	<i>overlap_length</i>	NULL	0
<i>chunk_text_config</i>	<i>strategy</i>	NULL	'chars'

## 15.5 Pipelines reference

The full pipeline API reference lives alongside the pipeline documentation. See [Pipelines API reference](#).

Category	Items
Types and enums	<i>aidb.PipelineAutoProcessingMode</i> , <i>aidb.PipelineDataFormat</i> , <i>aidb.PipelineSourceType</i> , <i>aidb.PipelineDestinationType</i> , <i>aidb.PipelineStepOperation</i> , <i>aidb.PipelineStatus</i> , <i>aidb.DistanceOperator</i>
Domains	<i>aidb.pipeline_name_50</i> , <i>aidb.background_sync_interval</i>
Views	<i>aidb.pipelines</i> , <i>aidb.pipeline_metrics</i>
Functions	<i>aidb.create_pipeline</i> , <i>aidb.update_pipeline</i> , <i>aidb.delete_pipeline</i> , <i>aidb.run_pipeline</i> , <i>aidb.get_pipeline_metrics</i>
Pipeline helpers	<i>aidb.chunk_text_config</i> , <i>aidb.summarize_text_config</i> , <i>aidb.ocr_config</i> , <i>aidb.html_parse_config</i> , <i>aidb.pdf_parse_config</i> , <i>aidb.knowledge_base_config</i> , <i>aidb.knowledge_base_config_from_kb</i> , <i>aidb.inference_config</i>
Vector index helpers	<i>aidb.vector_index_hnsw_config</i> , <i>aidb.vector_index_ivfflat_config</i> , <i>aidb.vector_index_chord_hnsw_config</i> , <i>aidb.vector_index_chord_vchordq_config</i> , <i>aidb.vector_index_hsphere_optimized_config</i> , <i>aidb.vector_index_disabled_config</i>

## 15.6 Agents reference

Reference for agent-related functions, views, and types. For guide-style documentation, see [Agents reference](#).

### 15.6.1 Catalog views

#### **aidb.agents**

Agent configuration. There's no dedicated "list agents" function — query this table directly.

Column	Type	Description
<i>id</i>	uuid	Internal identifier.
<i>name</i>	text	Unique agent name.
<i>instructions</i>	text	The agent's instructions.
<i>model</i>	text	The model backing the agent.
<i>role</i>	text	Postgres role the agent's tool calls execute as, if configured.
<i>delegates</i>	text[]	Names of agents this agent may delegate to.
<i>tools</i>	text[]	Names of tools this agent may call.
<i>output_type</i>	jsonb	Structured output schema, built with <i>aidb.output_type()</i> .
<i>input_token_budget</i>	integer	Input token budget per call.
<i>output_token_budget</i>	integer	Output token budget per call.
<i>max_iterations</i>	integer	Maximum reasoning iterations per call.
<i>timeout_seconds</i>	integer	Maximum wall-clock time per call.
<i>budget_strategy</i>	<i>aidb.budget_strategy</i>	Behavior when a budget is exceeded. See <a href="#">Types</a> below.
<i>preset</i>	text	Stored, but not currently resolved to any behavior.
<i>created_at</i>	timestampz	Creation time.
<i>updated_at</i>	timestampz	Last update time.

**aidb.agent\_tasks**

One row per `agent_converse` call (a “task”), including its lifecycle status.

Column	Type	Description
<i>task_id</i>	uuid	Internal task identifier.
<i>agent_id</i>	uuid	The agent that ran this task ( <i>aidb.agents.id</i> ).
<i>conversation_id</i>	uuid	The conversation this task belongs to.
<i>request_message_id</i>	uuid	The triggering message's id in <i>aidb.conversation_log</i> .
<i>response_message_id</i>	uuid	The resulting answer's id in <i>aidb.conversation_log</i> , once complete.
<i>caller_role</i>	text	Postgres role that invoked <i>agent_converse</i> .
<i>status</i>	<i>aidb.task_status</i>	Current lifecycle status. See <i>Types</i> below.
<i>blocking</i>	boolean	Always <i>true</i> today — every task runs synchronously.
<i>error</i>	text	Error message, if the task failed.
<i>created_at</i>	timestampz	When the task started.
<i>completed_at</i>	timestampz	When the task finished, if it has.

**aidb.conversation\_log**

Every user prompt and agent answer, across every conversation — the same rows `aidb.get_conversation()` returns, unfiltered by `conversation_id`. Tool calls, model requests, and other internal activity aren’t included; query `aidb_internal.action_log` directly for the full activity history.

Column	Type	Description
<i>id</i>	uuid	Message id.
<i>conversation_id</i>	uuid	The conversation this message belongs to.
<i>agent_id</i>	uuid	The agent involved.
<i>task_id</i>	uuid	The task this message was part of.
<i>action_type</i>	<i>aidb.action_type</i>	<i>user_prompt</i> or <i>answer</i> .
<i>payload</i>	jsonb	The message's raw payload (a <i>user_message</i> or <i>llm_response</i> struct).
<i>created_at</i>	timestampz	When the message was created.
<i>completed_at</i>	timestampz	When it was completed.

**aidb.conversations**

One row per conversation, aggregated from `aidb.conversation_log`.

Column	Type	Description
<i>conversation_id</i>	uuid	The conversation's id.
<i>agent_id</i>	uuid	The agent involved.
<i>parent_conversation_id</i>	uuid	Always <i>NULL</i> — conversation forking isn't implemented.
<i>forked_from_message_id</i>	uuid	Always <i>NULL</i> — conversation forking isn't implemented.
<i>status</i>	<i>aidb.task_status</i>	Status of the most recent task in this conversation.
<i>message_count</i>	bigint	Number of messages in the conversation.
<i>last_message_id</i>	uuid	Id of the most recent message.
<i>created_at</i>	timestampz	When the conversation started.

continues on next page

Table 77 – continued from previous page

Column	Type	Description
<i>updated_at</i>	timestampz	When the conversation was last updated.

## 15.6.2 Agent management functions

### aidb.create\_agent

Registers a new agent.

#### Parameters

Parameter	Type	Default	Description
<i>name</i>	TEXT	Required	Unique name for the agent.
<i>instructions</i>	TEXT	Required	Instructions that define the agent's behavior.
<i>model</i>	TEXT	Required	Name of the model backing the agent. See <a href="#">Choosing a model</a> .
<i>tools</i>	TEXT[]	NULL	Names of tools (from <i>aidb.tools</i> ) the agent may call.
<i>delegates</i>	TEXT[]	NULL	Names of other agents this agent may delegate to.
<i>role</i>	TEXT	NULL	Postgres role the agent's tool calls execute as. Caller must be a member of this role.
<i>output_type</i>	JSONB	NULL	Structured output schema, built with <code>:ref:``aidb.output_type() &lt;#aidboutput_type&gt;`</code> .
<i>input_token_budget</i>	INTEGER	NULL (no limit)	Input token budget per <i>agent_converse</i> call.
<i>output_token_budget</i>	INTEGER	NULL (no limit)	Output token budget per call.
<i>max_iterations</i>	INTEGER	NULL	Maximum reasoning iterations per call. A hard ceiling of 25 always applies regardless.
<i>timeout</i>	INTEGER	NULL (300 applied at runtime)	Maximum wall-clock seconds per call.
<i>budget_strategy</i>	TEXT	'attempt_complete'	One of 'ignore', 'error', 'summarize', 'attempt_complete'. See <a href="#">Budgets and limits</a> .
<i>preset</i>	TEXT	NULL	Stored, but not currently resolved to any behavior.

#### Returns

TABLE(error TEXT) — no rows on success; a single row with `error` set on failure (never raises for a validation/name-collision failure).

### Example

```
SELECT aidb.create_agent(
 name => db_helper,
 instructions => You are a helpful assistant that answers questions about_
→this database.,
 model => my_gpt,
 tools => ARRAY[run_sql_query], -- optional; tool names from aidb.
→tools. Default: NULL (no tools)
 delegates => NULL, -- optional; names of other_
→agents to delegate to. Default: NULL (no delegation)
 role => NULL, -- optional; Postgres role tool_
→calls run as. Default: NULL (run as the calling role)
 output_type => NULL, -- optional; structured output_
→schema from aidb.output_type(). Default: NULL (plain-text answer)
 input_token_budget => NULL, -- optional; input token cap per_
→agent_converse call. Default: NULL (no limit)
 output_token_budget => NULL, -- optional; output token cap_
→per agent_converse call. Default: NULL (no limit)
 max_iterations => NULL, -- optional; reasoning-round cap_
→per call. Default: NULL (a hard ceiling of 25 always applies)
 timeout => NULL, -- optional; wall-clock seconds_
→cap per call. Default: NULL (300 seconds applied at runtime)
 budget_strategy => attempt_complete, -- optional; behavior when a_
→budget is exceeded. Default: attempt_complete
 preset => NULL -- optional; reserved for future_
→use, has no effect today. Default: NULL
);
```

### aidb.update\_agent

Updates an existing agent. Every parameter except `name` is optional and defaults to `NULL`, meaning “leave unchanged” — including `budget_strategy`, whose default here is `NULL`, not `'attempt_complete'`.

#### Parameters

Same parameters as `create_agent`, all optional (defaulting to `NULL`) except `name`.

#### Returns

TABLE(error TEXT) — no rows on success; a single row with `error` set on failure (agent not found, invalid field value).

### Example

```
SELECT aidb.update_agent(db_helper, model => my_new_gpt, max_iterations => 15);
```

**aidb.delete\_agent**

Deletes an agent by name.

**Parameters**

Parameter	Type	Default	Description
<i>name</i>	TEXT	Required	Name of the agent to delete.
<i>force</i>	BOOLEAN	NULL (false)	If <i>true</i> , also deletes the agent's internal task/action-queue records so deletion can proceed even if it has conversation history. The conversation transcript itself ( <i>aidb.conversation_log</i> ) is preserved either way.

**Returns**

TABLE(error TEXT) — no rows on success; a single row with `error` set on failure, including when the agent has conversation history and `force` wasn't passed.

**Example**

```
SELECT aidb.delete_agent(db_helper, force => true);
```

**15.6.3 Conversation functions****aidb.agent\_converse**

Runs an agent's reasoning loop against a prompt and returns its answer.

**Parameters**

Parameter	Type	Default	Description
<i>agent_name</i>	TEXT	Required	Name of the agent to converse with.
<i>prompt</i>	TEXT	Required	The prompt to send.
<i>conversation_id</i>	TEXT	NULL	Continue an existing conversation. Omit to start a new one (an id is generated automatically).
<i>read_only</i>	BOOLEAN	NULL	Force <i>Read-only mode</i> on or off. Auto-enabled on a read replica when omitted.
<i>output_type</i>	JSONB	NULL	Overrides the agent's configured <i>Structured output</i> schema for this call only.
<i>debug</i>	BOOLEAN	NULL	If <i>true</i> , also emit every action as a <i>NOTICE</i> . See <i>Debug mode</i> .

## Returns

TABLE(message TEXT, conversation\_id TEXT, error TEXT) — one row. Never raises for an agent-logic failure; error is populated instead, and message / conversation\_id are NULL. conversation\_id is also NULL for any read-only run, since nothing is persisted to attach an id to.

## Example

```
SELECT * FROM aidb.agent_converse(
 agent_name => db_helper,
 prompt => How many rows are in the orders table?,
 conversation_id => NULL, -- optional; continue an existing conversation by id.
 ↳Default: NULL (starts a new conversation)
 read_only => NULL, -- optional; force read-only mode on/off. Default: NULL
 ↳(auto: on for a read replica, off otherwise)
 output_type => NULL, -- optional; override the agents structured output
 ↳schema for this call. Default: NULL (use the agents own configuration)
 debug => NULL -- optional; also emit every action as a NOTICE.
 ↳Default: NULL (false)
);
```

## aidb.start\_agent\_session

Mints a fresh conversation id for a named agent, ahead of the first agent\_converse call that will use it.

## Parameters

Parameter	Type	Description
agent_name	TEXT	Name of the agent to converse with.

## Returns

TABLE(conversation\_id TEXT) . Unlike agent\_converse, this function **raises** (rather than returning an error column) if agent\_name is empty or unknown.

## Example

```
SELECT * FROM aidb.start_agent_session(db_helper);
```

## aidb.get\_conversation

Retrieves a conversation's messages — every user prompt and final answer, in chronological order. Excludes tool calls and other internal activity; see aidb\_internal.action\_log for the full activity history.

## Parameters

Parameter	Type	Description
<i>conversation_id</i>	TEXT	The conversation's id.

### Returns

TABLE(message\_id TEXT, task\_id TEXT, action\_type TEXT, role TEXT, sender\_id TEXT, contents TEXT) — one row per message. role is "user" or "agent" ; sender\_id is the specific sender (the prompt's own user field, or the agent's id). **Raises** (rather than returning an error column) if conversation\_id is empty.

### Example

```
SELECT * FROM aidb.get_conversation(8f14e45f-ceea-467e-9575-a3d1a2c1a123);
```

### aidb.get\_message

Retrieves one message's plain-text contents by id.

### Parameters

Parameter	Type	Description
<i>message_id</i>	TEXT	The message's id.

### Returns

TABLE(message TEXT, error TEXT) — one row. error (not message ) is populated when no such message exists; doesn't raise.

### Example

```
SELECT * FROM aidb.get_message(3fa85f64-5717-4562-b3fc-2c963f66afa6);
```

## 15.6.4 Structured output helpers

### aidb.output\_field

Builds one field of an agent's structured output schema.

### Parameters

Parameter	Type	Default	Description
<i>name</i>	TEXT	Required	The field's name.
<i>field_type</i>	TEXT	Required	The field's type (for example, 'TEXT', 'FLOAT', 'BOOLEAN').
<i>description</i>	TEXT	NULL	Description shown to the model.

Returns

JSONB — pass one or more of these to `aidb.output_type()` .

aidb.output\_type

Builds a complete structured output schema from one or more fields.

Parameters

Parameter	Type	Description
<i>fields</i>	VARIADIC JSONB[]	One or more <i>aidb.output_field()</i> results.

Returns

JSONB — pass to `create_agent` /`update_agent` 's `output_type` parameter, or `agent_converse` 's.

Example

```
SELECT aidb.create_agent(
 name => sentiment_tagger,
 instructions => Classify the sentiment of the given text.,
 model => my_gpt,
 output_type => aidb.output_type(
 aidb.output_field(sentiment, TEXT, One of: positive, negative, neutral),
 aidb.output_field(confidence, FLOAT)
)
);
```

15.6.5 Types

aidb.budget\_strategy

```
CREATE TYPE aidb.budget_strategy AS ENUM (
 ignore,
 error,
 summarize,
 attempt_complete
);
```

See *Budgets and limits* for what each value does.

aidb.task\_status

```
CREATE TYPE aidb.task_status AS ENUM (
 PENDING,
 IN_PROGRESS,
 RECOVERY,
```

(continues on next page)

(continued from previous page)

```

 AWAIT_APPROVAL,
 DENIED,
 ERROR,
 TIMEOUT,
 CANCELED,
 SUCCESS,
 EVALUATING,
 COMPLETE
);

```

Surfaced through `aidb.agent_tasks.status` and `aidb.conversations.status`. Since every task today runs synchronously to completion within one `agent_converse` call, a task's terminal status is normally `SUCCESS` or `ERROR`; the remaining values are reserved for asynchronous/approval workflows that aren't implemented yet.

## 15.7 Tools reference

Reference for tool-related functions, views, and types. For guide-style documentation, see [Tools reference](#). For the full native tool catalog, see [Native tools](#).

### 15.7.1 Catalog views

#### `aidb.tools`

The unified, read-only catalog of every tool AIDB knows about — the union of native tools, registered SQL tools, and cached MCP tools.

Column	Type	Description
<i>name</i>	text	The tool's name.
<i>description</i>	text	Shown to the model as part of the tool's definition.
<i>tool_type</i>	text	' <i>native_tool</i> ', ' <i>sql_tool</i> ', or ' <i>mcp_tool</i> '.
<i>params</i>	jsonb	For <i>native_tool/sql_tool</i> : an array of <code>{name, type, description}</code> objects. For <i>mcp_tool</i> : the server's raw JSON Schema <i>inputSchema</i> .
<i>read_only</i>	boolean	Whether the tool is known to never write. Always <i>false</i> for <i>mcp_tool</i> rows — see <a href="#">Read-only mode</a> .

#### `aidb.sql_tool_registry`

Backs registered SQL tools. `aidb.tools` is the union view most queries should use instead; query this directly to see a SQL tool's stored query text.

Column	Type	Description
<i>name</i>	text	Unique tool name (primary key).
<i>description</i>	text	Shown to the model.
<i>query_text</i>	text	The stored SQL statement, with <code>_\${name}</code> placeholders.
<i>read_only</i>	boolean	Whether the tool is enforced as read-only.
<i>return_type_hint</i>	<code>aidb.ToolParam[]</code>	Documented return shape. Not currently surfaced to the model.

continues on next page

Table 87 – continued from previous page

Column	Type	Description
<i>params</i>	<i>aidb.ToolParam[]</i>	The tool's declared parameters.
<i>created_at</i>	timestamp	Creation time.
<i>updated_at</i>	timestamp	Last update time.

**aidb.mcp\_registry**

Registered external MCP servers.

Column	Type	Description
<i>name</i>	text	Unique server name (primary key).
<i>url</i>	text	The server's MCP endpoint.
<i>transport</i>	text	' <i>streamable_http</i> ' or ' <i>sse</i> '.
<i>headers</i>	jsonb	Stored request headers. Not readable by <i>aidb_users</i> (may hold secrets).
<i>headers_env</i>	text	Environment variable name to read headers from instead. Mutually exclusive with <i>headers</i> .
<i>tool_filter</i>	text[]	Tool names imported from this server, if filtered.
<i>created_at</i>	timestamp	Registration time.
<i>updated_at</i>	timestamp	Last update time.

**15.7.2 Tool management functions****aidb.create\_sql\_tool**

Registers a parameterized SQL query as a tool.

**Parameters**

Parameter	Type	Default	Description
<i>name</i>	TEXT	Required	Unique name for the tool.
<i>description</i>	TEXT	Required	Shown to the model as part of the tool's definition.
<i>sql_statement</i>	TEXT	Required	The query to run, with <i>\${name}</i> placeholders for each declared parameter.
<i>params</i>	JSONB	Required	The tool's parameters, built with <i>aidb.tool_params()</i> .
<i>read_only</i>	BOOLEAN	<i>true</i>	If <i>true</i> , reject the statement at registration time unless it's provably read-only.
<i>return_type_hint</i>	JSONB	NULL	Documents the query's return shape. Not currently surfaced to the model.

## Returns

TEXT — the tool’s name, on success. Raises if the name is already taken (by any tool type) or the statement isn’t a single, invocable, read-only-compliant (if `read_only => true`) statement.

## Example

```
SELECT aidb.create_sql_tool(
 name => orders_by_customer,
 description => Look up recent orders for a customer by id.,
 sql_statement => SELECT id, status, total FROM orders WHERE customer_id =_
 ↳ ${customer_id} ORDER BY created_at DESC LIMIT ${limit},
 params => aidb.tool_params(
 aidb.tool_param(customer_id, INT, The customers id.),
 aidb.tool_param(limit, INT, Maximum number of orders to return.)
),
 read_only => true, -- optional; enforced, not just descriptive. Default:_
 ↳ true
 return_type_hint => NULL -- optional; documents the returned row shape, same_
 ↳ aidb.tool_params() shape as `params`. Default: NULL
);
```

## aidb.delete\_tool

Deletes a SQL tool or an MCP server registration by name (removing every tool the server advertised along with it). Built-in native tools can’t be deleted.

## Parameters

Parameter	Type	Description
<i>name</i>	TEXT	The tool or server name.

## Returns

TEXT — the deleted name, on success. Raises if no such tool or server exists, or if `name` is a native tool.

## aidb.import\_mcp\_tools

Registers an external MCP server and imports the tools it advertises.

## Parameters

Parameter	Type	Default	Description
<i>name</i>	TEXT	Required	Unique name for the server registration.
<i>url</i>	TEXT	Required	The server's MCP endpoint URL.
<i>transport</i>	TEXT	'streamable_http'	'streamable_http' or 'sse' — only 'streamable_http' is fully supported today.
<i>headers</i>	JSONB	NULL	Request headers to send. Mutually exclusive with <i>headers_env</i> .
<i>tool_filter</i>	TEXT[]	NULL	Import only these tool names. Omit to import everything the server advertises.
<i>headers_env</i>	TEXT	NULL	Environment variable holding the headers JSON, read fresh on each use instead of storing it. Mutually exclusive with <i>headers</i> . Name must start with the <i>AI pipelines</i> prefix.

## Returns

TEXT — the server's name, on success. Raises if the name is taken, the URL is unreachable, or no advertised tool matches *tool\_filter*.

## Example

```
SELECT aidb.import_mcp_tools(
 name => weather,
 url => https://weather.example.com/mcp,
 transport => streamable_http, -- optional; streamable_http or sse. Default:
↳streamable_http
 headers => NULL, -- optional; request headers to send, for
↳example, an auth token. Mutually exclusive with headers_env. Default: NULL
 tool_filter => NULL, -- optional; import only these tool names.
↳Default: NULL (import everything the server advertises)
 headers_env => NULL -- optional; env var holding the headers JSON
↳instead of storing it. Mutually exclusive with headers. Default: NULL
);
```

## aidb.refresh\_mcp\_tools

Re-fetches an already-registered MCP server's advertised tools and refreshes the cache.

## Parameters

Parameter	Type	Description
<i>name</i>	TEXT	The registered server's name.

**Returns**

INTEGER — the number of tools now cached (after `tool_filter` applies).

**aidb.get\_mcp\_tools**

Converts every currently registered tool (native, SQL, and MCP alike) into an MCP `tools/list` -shaped descriptor.

**Returns**

TABLE(name TEXT, description TEXT, input\_schema JSONB) — one row per tool in `aidb.tools`.

**15.7.3 Parameter helpers****aidb.tool\_param**

Builds one SQL tool parameter.

**Parameters**

Parameter	Type	Default	Description
<i>name</i>	TEXT	Required	The parameter's name.
<i>type</i>	TEXT	Required	The parameter's SQL type.
<i>description</i>	TEXT	Required	Description shown to the model.

**Returns**

JSONB — pass one or more of these to `aidb.tool_params()`.

**aidb.tool\_params**

Combines one or more `aidb.tool_param()` results, or (called with no arguments) declares an empty parameter list.

**Parameters**

Parameter	Type	Description
<i>params</i>	VARIADIC JSONB[]	Zero or more <code>aidb.tool_param()</code> results.

**Returns**

JSONB — pass to `aidb.create_sql_tool()`'s `params` or `return_type_hint`.

**aidb.param / aidb.params**

Generic parameter-spec constructors, used by native tools that accept an enum-constrained argument. `aidb.param()` additionally accepts `enum_values TEXT[]`. Not typically needed directly — `aidb.tool_param()` / `aidb.tool_params()` are the constructors for `create_sql_tool()`.