



AIDB

Release 7.3.2

EnterpriseDB Corporation

Apr 30, 2026

1	AIDB	3
1.1	Where to start	3
1.2	Documentation map	3
2	AIDB release notes	5
2.1	AIDB 7.x release notes	6
2.1.1	AIDB 7.2.0 release notes	6
2.1.2	AIDB 7.3.0 release notes	7
2.1.3	AIDB 7.3.2 release notes	8
3	AIDB overview	11
3.1	What AIDB does	11
3.2	AIDB as a standalone extension	11
3.3	Key capabilities	12
3.3.1	Known issues	12
4	Compatibility	13
4.1	AIDB	13
4.1.1	Supported platforms	13
4.1.2	Supported Postgres versions	13
4.2	PGFS	13
5	Installing and configuring AIDB	15
5.1	Next steps	15
5.1.1	Installing AIDB	15
5.1.2	Configuring AIDB	16
6	Integrating models	19
6.1	Local models	19
6.1.1	Built-in models	19
6.1.2	Custom local models	20
6.1.3	Using a registered model	22
6.2	Support matrix	23
6.2.1	Local embedded models	23
6.2.2	NIM models	24
6.3	External model connections	26
6.3.1	OpenAI-compatible endpoints	26

6.3.2	NVIDIA NIM	28
6.3.3	Google Gemini	30
6.3.4	OpenRouter	30
7	Configuring external storage	33
7.1	How it works	33
7.2	Step 1: Create a storage location	33
7.2.1	S3-compatible object store	34
7.2.2	Local file system	34
7.3	Step 2: Create a volume	34
7.3.1	Example: PDFs in S3	35
7.3.2	Example: Images in a local directory	35
7.4	Step 3: Use the volume as a pipeline source	35
7.5	Managing volumes	35
7.5.1	List and delete	35
7.5.2	Inspect volume contents	36
8	AI pipelines	37
8.1	Pipelines overview	37
8.1.1	Integrated data lifecycle	38
8.1.2	Pipeline architecture	38
8.1.3	Supported models	38
8.2	Creating pipelines	38
8.2.1	Table sources	39
8.2.2	Volume sources (PGFS)	39
8.3	Pipeline steps	40
8.3.1	ChunkText	40
8.3.2	ParseHtml	41
8.3.3	ParsePdf	41
8.3.4	PerformOcr	42
8.3.5	SummarizeText	43
8.3.6	KnowledgeBase	43
8.4	Orchestration	45
8.4.1	Auto-processing	45
8.4.2	Background workers	46
8.4.3	Observability	48
8.4.4	Pipeline error log	49
8.5	Pipelines API reference	52
8.5.1	Types	52
8.5.2	Domains	55
8.5.3	Views	55
8.5.4	Functions	57
8.5.5	Pipeline step config helpers	59
8.5.6	Vector index config helpers	63
8.5.7	Model config helpers	65
8.6	Examples	66
8.6.1	Creating a pipeline (table source)	66
8.6.2	Creating a pipeline (volume source)	66
8.6.3	Running and updating pipelines	67
8.6.4	Monitoring and deletion	67
8.6.5	End-to-end example: Multi-step pipeline with intermediate storage	68
9	Knowledge bases	71
9.1	Retrieval functions	71

9.2	Flow of retrieval functions	71
9.2.1	aidb.retrieve_text()	72
9.2.2	aidb.retrieve_key()	72
9.3	Advanced querying: Joining intermediate steps	72
9.3.1	Example syntax:	72
10	SQL functions	89
10.1	Text chunking	90
10.1.1	Configuration options	90
10.2	Data parsing	91
10.2.1	Parse HTML	91
10.2.2	Parse PDF	92
10.3	Performing OCR	92
10.4	Text summarization	94
10.4.1	aidb.summarize_text()	95
10.5	Text summarization	95
10.6	Text summarization	95
10.6.1	Summarization use cases	95
10.6.2	Summarization cookbook	99
10.7	AI inference functions	111
10.7.1	aidb.encode_text()	111
10.7.2	aidb.encode_text_batch()	111
10.7.3	aidb.encode_image()	112
10.7.4	aidb.decode_text()	112
10.7.5	aidb.decode_text_batch()	113
10.7.6	aidb.inference_config()	113
10.8	Text reranking	113
10.8.1	Using reranking after knowledge base retrieval	114
11	Reference	117
11.1	Models reference	117
11.1.1	Catalog views	117
11.1.2	Model management functions	118
11.1.3	HCP model functions	120
11.1.4	Inference functions	121
11.1.5	Inference configuration helper	125
11.1.6	Model config helpers	126
11.2	Knowledge bases reference	132
11.2.1	Views	132
11.2.2	Types	133
11.2.3	Knowledge base functions	134
11.2.4	Volume functions	136
11.3	Operations reference	137
11.3.1	Error log functions	137
11.3.2	Error categories	141
11.4	Functions reference	141
11.4.1	aidb.chunk_text	142
11.4.2	aidb.parse_html	143
11.4.3	aidb.parse_pdf	143
11.4.4	aidb.perform_ocr	144
11.4.5	aidb.summarize_text	145
11.4.6	aidb.summarize_text_aggregate	146
11.4.7	Parameter defaults quick reference	146
11.5	Pipelines reference	147

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. ***Pipelines 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 built-in 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. ***Data 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.
- 6. ***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.

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

continues on next page

Table 1 – continued from previous page

Section	What it covers
<i>Configuring external storage</i>	Using S3-compatible object stores and local file systems as pipeline data sources
<i>Reference → Functions</i>	Defining pipelines, pipeline steps, orchestration, and examples
<i>Knowledge bases reference</i>	Semantic search, hybrid search, and vector index options
<i>SQL functions</i>	Standalone SQL functions for chunking, parsing, OCR, summarization, and embedding
<i>Reference → Functions</i>	Full API reference for all AIDB functions and views

AIDB release notes

AIDB is EDB’s PostgreSQL extension for building AI-powered applications. It provides pipelines for knowledge base management, data preparation, and retrieval-augmented generation (RAG), along with integrations for internal and external AI models.

The AIDB documentation describes the latest version of AIDB. These release notes provide information on what was new in each release.

Version	Release date
<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
6.1.0	26 Feb 2026
6.0.0	10 Feb 2026
5.3.0	04 Dec 2025
5.2.0	26 Nov 2025
5.1.0	05 Nov 2025
5.0.0	29 Oct 2025
4.5.0	10 Sep 2025
4.4.0	13 Aug 2025
4.3.0	04 Jul 2025
4.2.0	16 Jun 2025
4.1.1	28 May 2025
4.1.0	19 May 2025
4.0.1	09 May 2025
4.0.0	05 May 2025
3.0.1	03 Apr 2025
2.2.1	13 Mar 2025
2.1.2	24 Feb 2025
2.1.1	03 Feb 2025
2.0.0	13 Jan 2025

continues on next page

Table 1 – continued from previous page

Version	Release date
1.0.7	10 Dec 2024

2.1 AIDB 7.x release notes

Release notes for AIDB major version 7.

Version	Release date
AIDB 7.3.2 release notes	17 Apr 2026
AIDB 7.3.0 release notes	08 Apr 2026
AIDB 7.2.0 release notes	30 Mar 2026

2.1.1 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.

continues on next page

Table 3 – continued from previous page

Type	Description
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.

2.1.2 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.

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.retrieveers</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>

continues on next page

Table 5 – continued from previous page

Deprecated objects	Replacement
<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>

2.1.3 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.

continues on next page

Table 6 – continued from previous page

Type	Description
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.

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 two primary building blocks: **pipelines** and **knowledge bases**.

**** *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.

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 AI Factory or Hybrid Manager. You install it like any other Postgres extension, configure it via `postgresql.conf`, and interact with it through SQL.

!!!note If you're using AI Factory, AIDB is already included as part of that platform, but AI Factory is not 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.

Data source support — Pipelines can read from standard Postgres tables or from external storage via the Postgres File System (PGFS) extension, which supports object store and local file systems. Pipelines 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. VectorChord is also supported as a high-performance alternative, offering faster indexing and better throughput at scale.

3.3.1 Known issues

The following known issues and limitations apply to the current release of AIDB.

`aidb.retry_pipeline_errors()` not yet available

Affects: All versions

The function for retrying failed pipeline records from the error log, `aidb.retry_pipeline_errors()`, is not yet available. It is planned for a future release.

Workaround: Use `aidb.get_error_logs()` to inspect failed records and identify the cause of failure. Once you've resolved the underlying issue (for example, replacing a corrupt source file or correcting a model configuration), use `aidb.clear_error_logs()` to remove the error entries and then re-run the pipeline with `aidb.run_pipeline()` to re-process the affected data.

4.1 AIDB

4.1.1 Supported platforms

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

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*

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 Installing AIDB

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

!!!note If you're deploying through AI Factory, AIDB is bundled and configured as part of that setup. Refer to the AI 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](#).

1. Install VectorChord (optional): for high-performance, low-latency vector retrieval. See [Installing VectorChord](#).
2. Install VectorChord-BM25 (optional): required only if you need hybrid keyword + semantic search. See [Installing VectorChord-BM25](#).

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

5.1.2 Configuring AIDB

To use AIDB, add it to `shared_preload_libraries`, create the extensions in the database, and verify the installation.

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](#) .

Enable VectorChord (optional)

If you installed VectorChord, create the extension in your database:

```
CREATE EXTENSION vectorchord;
```

To also enable hybrid search (BM25 + vector):

```
CREATE EXTENSION vectorchord_bm25;
```

VectorChord knowledge bases can then be configured at the knowledge base level — see [Knowledge bases](#) .

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 Air-gapped environments are not currently supported.

Validate the installation

Confirm that the extensions are installed by running `\dx` in `psql` and looking for `aidb`, `vector`, `pgfs` (if installed), and `vectorchord` and `vectorchord_bm25` (if those extensions were installed) in the output:

```
aidb=# \dx
__OUTPUT__

              List of installed extensions
  Name          | Version | Schema | Description
-----+-----+-----+-----
aidb            | 7.3.1   | aidb   | aidb: makes it easy to build AI_
applications with postgres
pgfs            | 1.0.6   | pgfs   | pgfs: enables access to filesystem-
like storage locations
vector          | 0.8.0   | public | vector data type and ivfflat and hnsw_
access methods
vectorchord     | 0.3.0   | public | vectorchord: fast vector search for_
postgres
vectorchord_bm25 | 0.1.0   | public | vectorchord_bm25: bm25 full-text_
search for postgres
```

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 built-in 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>	Built-in local models (BERT, CLIP, T5, Llama) and how to register custom variants
<i>Support matrix</i>	Pre-vetted HuggingFace model variants confirmed to work with AIDB's local providers
<i>External model connections</i>	Connecting to OpenAI-compatible APIs, NVIDIA NIM, Google Gemini, and Open-Router

6.1 Local models

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

- **Built-in models** — AIDB bundles a default variant of each supported model type and registers them automatically. No configuration is needed to start using them.
- **Custom local 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()`.

6.1.1 Built-in models

The following models are available in every AIDB installation without any configuration:

Model name	Provider	Use
<i>bert</i>	<i>bert_local</i>	Text embeddings
<i>clip</i>	<i>clip_local</i>	Multimodal embeddings (text and image)
<i>t5</i>	<i>t5_local</i>	Text-to-text generation
<i>llama</i>	<i>llama_instruct_local</i>	Text-to-text generation
<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 built-in BERT model
SELECT aidb.encode_text(bert, The quick brown fox);

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

- - Text generation with the built-in T5 model
SELECT aidb.decode_text(t5, Translate to French: Hello, world.);
```

6.1.2 Custom local 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

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.

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>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.

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,
```

(continues on next page)

(continued from previous page)

```

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>cache_dir</i>	TEXT	NULL	Local directory for caching model files.
<i>model_path</i>	TEXT	NULL	Explicit local path to model weights (overrides <i>model</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.

!!!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.

6.1.3 Using a registered model

Once registered, a model is referenced by its name in any AIDB function. Custom and built-in 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(
    my_pipeline,
    ...
    step_1_operation => Embed,
    step_1_options   => aidb.knowledge_base_config(my_bert, Text),
    ...
);

```

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

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

6.2 Support matrix

6.2.1 Local embedded models

The following HuggingFace model variants have been tested and verified to work with AIDB's local model providers. You can use any of these identifiers with the corresponding config helper when calling `aidb.create_model()`.

BERT

Model name	HuggingFace identifier	Config string
<i>bert_all-MiniLM-L12-v1</i>	<i>sentence-transformers/all-MiniLM-L12-v1</i>	<code>aidb.bert_config('sentence-transformers/all-MiniLM-L12-v1', revision => 'main')</code>
<i>bert_all-MiniLM-L6-v1</i>	<i>sentence-transformers/all-MiniLM-L6-v1</i>	<code>aidb.bert_config('sentence-transformers/all-MiniLM-L6-v1', revision => 'main')</code>
<i>bert_all-MiniLM-L6-v2</i>	<i>sentence-transformers/all-MiniLM-L6-v2</i>	<code>aidb.bert_config('sentence-transformers/all-MiniLM-L6-v2', revision => 'main')</code>
<i>bert_all-distilroberta-v1</i>	<i>sentence-transformers/all-distilroberta-v1</i>	<code>aidb.bert_config('sentence-transformers/all-distilroberta-v1', revision => 'main')</code>
<i>bert msmarco-bert-base</i>	<i>sentence-transformers/msmarco-bert-base-dot-v5</i>	<code>aidb.bert_config('sentence-transformers/msmarco-bert-base-dot-v5', revision => 'main')</code>
<i>bert_multi-qa-MiniLM-L6-cos-v1</i>	<i>sentence-transformers/multi-qa-MiniLM-L6-cos-v1</i>	<code>aidb.bert_config('sentence-transformers/multi-qa-MiniLM-L6-cos-v1', revision => 'main')</code>
<i>bert_multi-qa-MiniLM-L6-dot-v1</i>	<i>sentence-transformers/multi-qa-MiniLM-L6-dot-v1</i>	<code>aidb.bert_config('sentence-transformers/multi-qa-MiniLM-L6-dot-v1', revision => 'main')</code>
<i>bert_paraphrase-TinyBERT-L6-v2</i>	<i>sentence-transformers/paraphrase-TinyBERT-L6-v2</i>	<code>aidb.bert_config('sentence-transformers/paraphrase-TinyBERT-L6-v2', revision => 'main')</code>
<i>bert_paraphrase-multilingual-MiniLM-L12-v2</i>	<i>sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2</i>	<code>aidb.bert_config('sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2', revision => 'main')</code>
<i>bert_paraphrase-multilingual-mpnet-base-v2</i>	<i>sentence-transformers/paraphrase-multilingual-mpnet-base-v2</i>	<code>aidb.bert_config('sentence-transformers/paraphrase-multilingual-mpnet-base-v2', revision => 'main')</code>

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

CLIP

Model name	HuggingFace identifier	Config string
<i>openai/clip-vit-base-patch32</i>	<i>openai/clip-vit-base-patch32</i>	<code>aidb.clip_config('openai/clip-vit-base-patch32')</code>

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

Llama

Model name	HuggingFace identifier	Config string
<i>Hugging-FaceTB/SmolLM2-1.7B</i>	<i>HuggingFaceTB/SmolLM2-1.7B-Instruct</i>	<code>aidb.llama_config('Hugging-FaceTB/SmolLM2-1.7B-Instruct', revision => 'main')</code>
<i>Hugging-FaceTB/SmolLM2-135M</i>	<i>HuggingFaceTB/SmolLM2-135M-Instruct</i>	<code>aidb.llama_config('Hugging-FaceTB/SmolLM2-135M-Instruct', revision => 'main')</code>
<i>Hugging-FaceTB/SmolLM2-360M</i>	<i>HuggingFaceTB/SmolLM2-360M-Instruct</i>	<code>aidb.llama_config('Hugging-FaceTB/SmolLM2-360M-Instruct', revision => 'main')</code>
<i>TinyLlama/TinyLlama-1.1B-C</i>	<i>TinyLlama/TinyLlama-1.1B-Chat-v1.0</i>	<code>aidb.llama_config('TinyLlama/TinyLlama-1.1B-Chat-v1.0', revision => 'main')</code>

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

T5

Model name	HuggingFace identifier	Config string
<i>t5-small</i>	<i>t5-small</i>	<code>aidb.t5_config('t5-small', revision => 'main')</code>
<i>t5-base</i>	<i>t5-base</i>	<code>aidb.t5_config('t5-base', revision => 'main')</code>
<i>t5-large</i>	<i>t5-large</i>	<code>aidb.t5_config('t5-large', revision => 'main')</code>

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

!!!Note These are the pre-vetted variants confirmed to work with AIDB. You are not 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.

6.2.2 NIM models

AIDB supports NVIDIA NIM microservices via several providers. The tables below list the compatible NIM models for each provider. Models marked as **Default on HM** are shipped with EDB Hybrid Manager's Model Serving and are available out of the box in HM deployments — no separate deployment required.

NIM completions (`nim_completions`)

Supports: `aidb.decode_text()`, `aidb.decode_text_batch()`

NIM model identifier	Default on HM
<i>nvidia/nemotron-3-nano</i>	OK
<i>nvidia/llama-3.3-nemotron-super-49b-v1</i>	OK

continues on next page

Table 12 – continued from previous page

NIM model identifier	Default on HM
<i>ibm/granite-guardian-3.0-8b</i>	—
<i>ibm/granite-3.0-8b-instruct</i>	—
<i>ibm/granite-3.0-3b-a800m-instruct</i>	—
<i>meta/llama-3.3-70b-instruct</i>	—
<i>meta/llama-3.2-3b-instruct</i>	—
<i>meta/llama-3.2-1b-instruct</i>	—
<i>meta/llama-3.1-405b-instruct</i>	—
<i>meta/llama-3.1-70b-instruct</i>	—
<i>meta/llama-3.1-8b-instruct</i>	—
<i>meta/llama3-70b-instruct</i>	—
<i>meta/llama3-8b-instruct</i>	—
<i>nvidia/llama-3.1-nemotron-70b-instruct</i>	—
<i>nvidia/llama-3.1-nemotron-51b-instruct</i>	—
<i>nvidia/nemotron-mini-4b-instruct</i>	—
<i>nvidia/nemotron-4-340b-instruct</i>	—
<i>google/shieldgemma-9b</i>	—
<i>google/gemma-7b</i>	—
<i>google/codegemma-7b</i>	—

NIM embeddings (nim_embeddings)

Supports: `aidb.encode_text()` , `aidb.encode_text_batch()`

NIM model identifier	Default on HM
<i>nvidia/llama-3.2-nemoretriever-300m-embed-v1</i>	OK
<i>nvidia/nv-embedqa-e5-v5</i>	—

NIM CLIP (nim_clip)

Supports: `aidb.encode_text()` , `aidb.encode_text_batch()` , `aidb.encode_image()`

NIM model identifier	Default on HM
<i>nvidia/nvclip</i>	OK

NIM OCR (nim_paddle_ocr)

Supports: `aidb.perform_ocr()`

NIM model identifier	Default on HM
<i>baidu/paddleocr</i>	OK

NIM reranking (nim_reranking)

Supports: `aidb.rerank_text()`

NIM model identifier	Default on HM
<i>nvidia/llama-3.2-nv-rerankqa-1b-v2</i>	OK

!!!Note Models marked as **Default on HM** are pre-deployed by EDB Hybrid Manager’s Model Serving. All other models can be self-hosted using KServe and registered with AIDB using `aidb.create_model()`. For setup instructions, see [NVIDIA NIM](#).

6.3 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.

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

6.3.1 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,
    api_key => sk-...,
    url => https://api.openai.com/v1
  )
);

-- 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.

continues on next page

Table 17 – continued from previous page

Parameter	Type	Default	Description
<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>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,
    api_key     => sk-...,
    temperature => 0.2
  )
);
```

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>).

continues on next page

Table 18 – continued from previous page

Parameter	Type	Default	Description
<i>thinking</i>	BOOLEAN	NULL	Enable extended reasoning (supported models only).
<i>max_concurrent_requests</i>	INTEGER	NULL	Maximum concurrent requests to the endpoint.
<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.

6.3.2 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 the model identifier and your NIM API key:

```
SELECT aidb.create_model(
  my_nim_llm,
  nim_completions,
  {"model": "meta/llama-3.3-70b-instruct"}::JSONB,
  {"api_key": "<NIM API KEY>"}::JSONB
);
```

To use a NIM instance running in your own environment, add a `url` key to the config pointing at your NIM endpoint.

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

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

__OUTPUT__

          decode_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(
    api_key => nvapi-...,
    model   => nvidia/nvclip
```

(continues on next page)

(continued from previous page)

```
)
);
```

`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_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(api_key => nvapi-...)
);
```

`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(
    api_key => nvapi-...,
    model   => nvidia/nv-rerankqa-mistral-4b-v3
  )
);
```

`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.

6.3.3 Google Gemini

Use the `gemini` provider with `aidb.gemini_config()` . Pass your Google API key as the first argument:

```
SELECT aidb.create_model(
  my_gemini,
  gemini,
  config => aidb.gemini_config(
    AIza...,
    model => gemini-2.0-flash
  )
);
```

`aidb.gemini_config()` parameters:

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 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).

6.3.4 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,
    api_key => sk-or-...
  )
);
```

`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 <i>aidb.max_tokens_config()</i>).

Embeddings

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

`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 [Pipelines API reference](#) for full parameter details on all config helper functions.

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.

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** — Connects a PGFS storage location to AIDB. Specifies the data format (Text, Image, Pdf) and an optional sub-path within the storage location. 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 name.

!!!Note The PGFS extension must be installed before creating storage locations. See [Configuring AIDB](#).

7.2 Step 1: Create a storage location

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

7.2.1 S3-compatible object store

```

-- Private S3 bucket with credentials
SELECT pgfs.create_storage_location(
    name => my_s3_location,
    uri  => s3://my-bucket/my-folder,
    options => {"region": "us-east-1", "access_key_id": "<key>", "secret_access_key":
-> "<secret>" }
);

-- Public S3 bucket (no credentials required)
SELECT pgfs.create_storage_location(
    name => my_public_bucket,
    uri  => 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,
    uri  => file:///tmp/pgfs/
);

```

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

7.3 Step 2: Create a volume

Use `aidb.create_volume()` to attach a PGFS storage location to AIDB. The volume is what pipelines and SQL functions reference.

```

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

```

Parameters:

Parameter	Description
<i>name</i>	Unique name for the volume. Used to reference it in pipelines.

continues on next page

Table 1 – continued from previous page

Parameter	Description
<i>storage_location</i>	Name of the PGFS storage location to attach.
<i>sub_path</i>	Optional path within the storage location. Useful for pointing multiple volumes at different folders in the same bucket.
<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 does not filter them. Ensure the volume only contains objects of the declared format.

7.3.1 Example: PDFs in S3

```
SELECT pgfs.create_storage_location(
  pdf_bucket,
  s3://my-docs-bucket,
  options => {"region": "us-east-1", "access_key_id": "<key>", "secret_access_key":
    "<secret>"}
);

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

7.3.2 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: Use 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 List and delete

List all volumes:

```
SELECT aidb.list_volumes();
```

Delete a volume:

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

!!!Note Deleting a PGFS storage location also deletes all volumes created on top of it.

7.5.2 Inspect volume contents

Use these functions to verify a volume before attaching it to a pipeline:

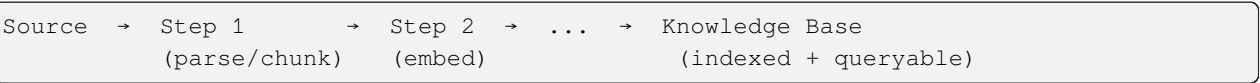
```
-- List all objects in the volume
SELECT * FROM aidb.list_volume_content(my_volume);

-- Read a specific file as BYTEA
SELECT aidb.read_volume_file(my_volume, report.pdf);

-- Read a plain text file as text
SELECT convert_from(
    aidb.read_volume_file(my_volume, notes.txt),
    utf8
);
```

These direct access functions are also useful for building custom SQL queries against external storage, independent of any pipeline.

A pipeline is the core building block of AIDB. It defines how raw data — from a Postgres table or an external volume — flows through a sequence of transformation steps and lands in an AI-ready destination.



Each step handles one transformation: parsing a PDF, chunking text, running OCR, summarizing content, or generating vector embeddings. The output of one step becomes the input for the next. At the end of the pipeline, your data is embedded, indexed, and ready to query with semantic or hybrid search.

Pipelines can run on demand, in batch, or automatically whenever source data changes — keeping your knowledge base in sync without manual intervention.

Page	What it covers
Pipelines overview	Core concepts: sources, steps, destinations, and how pipelines relate to knowledge bases.
Creating pipelines	Defining a pipeline with <code>aidb.create_pipeline()</code> — source, steps, auto-processing, and volume sources.
Pipeline steps	Available step types: <code>ChunkText</code> , <code>ParseHtml</code> , <code>ParsePdf</code> , <code>PerformOcr</code> , <code>SummarizeText</code> , <code>KnowledgeBase</code> .
Orchestration	Auto-processing modes, background workers, observability, and error handling.
Reference → Functions	Full API reference for pipeline types, views, CRUD functions, and config helpers.
Examples	End-to-end worked example.

8.1 Pipelines overview

Pipelines is an EDB Postgres AI component that automates the transformation of raw data into vector-searchable insights. It integrates *pgvector* *iterative index scans* search capabilities with built-in data preparation tools, keeping knowledge bases synchronized with source data without manual intervention.

8.1.1 Integrated data lifecycle

Pipelines automates the journey from raw data to searchable insights. By managing data ingestion, embedding generation, and indexing, it ensures your vector stores stay synchronized with your source data.

- **Flexible model integration:** Choose from supported local models, such as HuggingFace, or connect to OpenAI-compatible external APIs.
- **Preprocessing automation:** Built-in data preparation steps handle data cleaning and transformation directly within AI pipelines.
- **Eliminate stale data:** Auto-processing keeps embeddings current, significantly reducing the hallucinations often caused by outdated vector data.
- **Unified semantic search:** The intelligent knowledge base feature allows you to query text and images using a single function call, regardless of whether the data lives in Postgres tables or S3-compatible object storage.

8.1.2 Pipeline architecture

A pipeline is configured using the `aidb.create_pipeline()` function and consists of three main aspects:

- ***Creating pipelines*** — pipelines read from Postgres tables or from external cloud storage (S3, GCS, Azure) via Postgres File System (PGFS).
- ***Pipeline steps*** — up to 10 sequential operations transform your data. Steps include chunking, parsing, OCR, summarization, and embedding into a knowledge base. Each step is configured using a dedicated helper function.
- ***Orchestration*** — auto-processing modes (`Live`, `Background`, or `Disabled`) keep your knowledge base in sync with source data automatically, without manual intervention.

8.1.3 Supported models

The AIDB extension supports a diverse range of Open Encoder LLMs from HuggingFace (running locally on your Postgres node) and OpenAI Encoders (via cloud API). You can view all available options in the `aidb.model_providers` table.

8.2 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	<code>source</code> , <code>source_key_column</code> , <code>source_data_column</code>	This page
Steps	<code>step_1</code> ... <code>step_10</code> , <code>step_N_options</code>	Pipeline steps
Orchestration	<code>auto_processing</code> , <code>background_sync_interval</code> , <code>batch_size</code>	Orchestration
Destination	<code>destination</code> , <code>destination_key_column</code> , <code>destination_data_column</code>	Set automatically by the <code>KnowledgeBase</code> step, or specify explicitly

8.2.1 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,
  source    => my_table,
  source_key_column => id,
  source_data_column => content,
  step_1     => KnowledgeBase,
  step_1_options => aidb.knowledge_base_config(my_model, Text)
);
```

8.2.2 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,
  uri       => 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](#) .

8.3 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.

8.3.1 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(
```

(continues on next page)

(continued from previous page)

```

        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.

8.3.2 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)
);

```

8.3.3 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,
  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.

8.3.4 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
);

```

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.

8.3.5 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, 'reduce' iteratively compresses.
<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,
    strategy       => reduce,
    reduction_factor => 3
  ),
  step_2        => KnowledgeBase,
  step_2_options => aidb.knowledge_base_config(model => bert, data_format =>
  ↳ Text)
);
```

8.3.6 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 cannot 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>Cosine</i> , or <i>InnerProduct</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)
  )
);

```

To link multiple pipelines to the same knowledge base, use `aidb.knowledge_base_config_from_kb(data_format)` instead. This technique inherits the model and distance operator settings from the existing knowledge base.

Destination table

The KnowledgeBase step automatically creates a destination table named `pipeline_<pipeline_name>` with the following schema:

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.

Multi-pipeline knowledge bases

A single knowledge base can aggregate embeddings from multiple pipelines. The internal `knowledge_base_pipeline` junction table manages these mappings. When retrieving results via `aidb.retrieve_text()`, each row includes a `pipeline_name` column so you can identify which pipeline produced each embedding.

For knowledge base views and statistics, see [Knowledge bases reference](#).

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

8.4 Orchestration

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

- **Auto-processing:** Pipelines can run in the background or stay in sync with your data. You can configure batch sizes, 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.
- **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.
- **Audit & error logs:** It captures exactly why a specific file failed to parse, for example, a corrupted PDF or a timeout, allowing for quick troubleshooting.

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

8.4.1 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
<i>auto_processing</i>	<i>aidb.PipelineAutoProcessingMode</i>	Auto-processing mode: <i>Live</i> , <i>Background</i> , or <i>Disabled</i> .
<i>background_sync_interval</i>	<i>interval</i>	Interval between background executions. Must be between 1 second and 2 days. Only applies when <i>auto_processing</i> is <i>Background</i> .

8.4.2 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.

8.4.3 Observability

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

A good place to get an overview is the statistics table. Use the view `aidb.knowledge_base_stats`, or use its short alias `aidb.kbstat`. The view shows all configured knowledge base pipelines, which processing mode is set, and statistics about the processed records:

```
SELECT * from aidb.kbstat;
__OUTPUT__
 knowledge base      | auto processing | table: unprocessed rows | volume: scans_
->completed | count(source records) | count(embeddings)
-----+-----+-----+-----+
->-----+-----+-----+-----+
 kb_table_text_bg    | Background     | 0 | 15 |
-> | 15 | 15 |
 kb_table_text_manual | Disabled       | 0 | 15 |
-> | 15 | 15 |
```

(continues on next page)

(continued from previous page)

kb_table_image_manual	Disabled		0	
→	3	3		
kb_table_text_live	Live		0	
→	15	15		
kb_table_image_bg	Background		0	
→	3	3		
kb_volume_text_bg	Background			
→	6	7	7	
kb_volume_text_manual	Disabled			
→	0	0	0	
kb_volume_image_bg	Background			
→	4	177	6	
kb_volume_image_manual	Disabled			
→	1	177	6	
(9 rows)				

The *Change detection* mechanism is central to how auto-processing works. It's different for volume and table sources. For this reason, the stats table has different columns for these two source types.

- `table: unprocessed rows` : How many unique rows are listed in the backlog of change events. - If auto-processing is disabled, no new change events are captured.
- `volume: scans completed` : How many full listings of the source have been completed so far.
- `count(source records)` : How many records exist in the source for this pipeline.
 - For table sources, this number is always accurate.
 - For volume sources, this number is updated only after a full scan has completed.
- `count(embeddings)` : How many embeddings exist in the vector destination table for this pipeline.

8.4.4 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.
- Errors get one of four `error_category` values:

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.

continues on next page

Table 12 – continued from previous page

Category	Scope	Blocks pipeline?	Description
<i>PipelinePermanent</i>	Pipeline	Yes	Permanent failure that stopped the entire step (for example, invalid model config).

`RecordTemporary` and `RecordPermanent` errors don't block the pipeline — other records keep processing. `PipelineTemporary` and `PipelinePermanent` errors block the entire step. Temporary errors may resolve on retry; permanent errors point to bad data or misconfiguration.

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	error_category	failed_at	retry_count	last_retry_at
3	doc_99	{0,2}	2	ChunkText	text segment exceeded	RecordPermanent	2026-04-10 14:32:01.123+00	0	
2	doc_42	{0}	1	ParsePdf	failed to extract text	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();
```

```
__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)
```

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.

Retrying failed records

!!!Note `aidb.retry_pipeline_errors()` is not yet available. It will be included in a future release. See [Known issues](#).

When available, `aidb.retry_pipeline_errors()` will re-process specific error log entries from the step where they failed:

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

__OUTPUT__

```
error_id | status | error_message
-----+-----+-----
      1 | resolved |
      2 | failed  | failed to extract text from page 0
(2 rows)
```

Resolved entries are removed from the error log. Failed retries stay in the log with an incremented `retry_count` and updated `last_retry_at` timestamp.

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.

8.5 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](#).

8.5.1 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 <code>aidb.run_pipeline()</code> to trigger manually.

`aidb.PipelineDataFormat`

Specifies the format of source data the pipeline processes.

```
CREATE TYPE PipelineDataFormat AS ENUM (
    Text,
    Image,
```

(continues on next page)

(continued from previous page)

```
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,
```

(continues on next page)

(continued from previous page)

```

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>PartialErrors</i>	Some records failed to process; others succeeded.
<i>BlockingErrors</i>	The pipeline fails to run.
<i>Unknown</i>	Status cannot be determined.

aidb.DistanceOperator

Specifies the distance metric used for vector similarity search.

```

CREATE TYPE DistanceOperator AS ENUM (
L2,
InnerProduct,
Cosine,

```

(continues on next page)

(continued from previous page)

```

    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.

8.5.2 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).

8.5.3 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.

continues on next page

Table 20 – continued from previous page

Column	Type	Description
<i>steps</i>	jsonb	Ordered array of pipeline step definitions.
<i>auto_processing</i>	<i>aidb.PipelineAutoProcessing-Mode</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	Total number of records the failed processing.
<i>count(blocking errors)</i>	bigint	Total number of errors that prevent the pipeline from running.

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)
  -----+-----+-----+-----+
  ↳ -----+-----+-----+-----+
  ↳ -----+-----+-----+-----+
  pipeline__7471a | Background     | 0 | 5 | UpToDate |
  ↳ | 5 |
  ↳ 0 | 0
  pipeline__7471b | Background     | 0 | 5 | UpToDate |
  ↳ | 5 |
  ↳ 0 | 0
```

(continues on next page)

(continued from previous page)

animal_facts_kb	Disabled		0			
→		99			372	UpToDate
→	0		0			
animal_facts_kb_bert	Disabled		0			
→		99			372	UpToDate
→	0		0			
mpkb_pipe_int	Disabled		0			
→		2			4	UpToDate
→	0		0			
mpkb_pipe_text	Disabled		0			
→		2			4	UpToDate
→	0		0			
(6 rows)						

8.5.4 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 50 characters.
<i>source</i>	TEXT	Required	Name of the source table or volume.
<i>step_1</i>	aidb.PipelineStep-Operation	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.
<i>destination</i>	TEXT	NULL	Name of the destination table or volume.
<i>auto_processing</i>	aidb.PipelineAuto-ProcessingMode	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 Pipeline step config helpers).
<i>step_2 ... step_10</i>	aidb.PipelineStep-Operation	NULL	Operation for steps 2–10.

continues on next page

Table 22 – continued from previous page

Parameter	Type	Default	Description
<i>step_2_options</i> <i>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_col- umn</i>	text	Key column in the destination.
<i>destination_data_col- umn</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
);

-- 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);
```

8.5.5 Pipeline step config helpers

These functions return a JSONB configuration object for use in `step_N_options` parameters of `aidb.create_pipeline`.

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: 'chars' (default) or 'words'.

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 cannot 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,
  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).

continues on next page

Table 34 – continued from previous page

Parameter	Type	Default	Description
<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
);
```

8.5.6 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();
```

8.5.7 Model config helpers

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

8.6 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.

8.6.1 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
);
```

8.6.2 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
  )
);

```

8.6.3 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
);

```

8.6.4 Monitoring and deletion

To view your existing pipelines and their configurations, query the `aidb.pipelines` view:

```

SELECT * FROM aidb.pipelines;

```

name	source_type	source_schema	source	source_key_column	source_data_column	destination_type	destination_schema	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 aidb.delete_pipeline(kb_pipeline_table);
```

8.6.5 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.

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
    )
  ),
);

```

(continues on next page)

(continued from previous page)

```
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 do not 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(
    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;
```

Knowledge bases

A knowledge base is a vector-indexed store of embeddings. It is created automatically when a pipeline includes a *KnowledgeBase* — the pipeline handles embedding generation and indexing, and the knowledge base is the resulting queryable store.

Page	What it covers
<i>Hybrid search</i>	Combining semantic search with relational filters and BM25 keyword search
<i>Vector extensions</i>	VectorChord and VectorChord-BM25 for high-performance dense and sparse vector search
<i>Examples</i>	End-to-end worked examples for table and volume sources

9.1 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.

9.2 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.

9.2.1 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 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 example, `ChunkText`.

9.2.2 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.

9.3 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.

9.3.1 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
ON int_step.source_id = r.key
AND int_step.part_ids = (r.part_ids)[:1];
```

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'`.
- **BM25 full-text search** — rank results by keyword relevance using a BM25 index, 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.

Prerequisites

pgvector iterative index scans

Requires pgvector 0.8.0 or later (EDB Postgres AI defaults to 0.8.1).

Iterative index scans are the key enabler for combining vector `ORDER BY` with `WHERE` filters efficiently. Without them, the vector index pre-fetches a fixed number of candidates before filtering, which means a highly selective filter may return fewer results than your `LIMIT` requests. With iterative scans enabled, the index keeps producing candidates until your `LIMIT` is satisfied, even after filtering.

Enable iterative scans for HNSW indexes:

```
SET hnsw.iterative_scan = relaxed_order;
```

For IVFFlat indexes:

```
SET ivfflat.iterative_scan = on;
```

For very selective filters (few rows match), set a safety limit to prevent the scan from traversing the entire index:

```
SET hnsw.iterative_scan_max_tuples = 20000;
```

Important

The vector `ORDER BY` and the `WHERE` filter must be in the same query level — not separated by a CTE boundary — for iterative scans to take effect. The query patterns below are structured accordingly.

BM25 (optional)

BM25 full-text filtering requires the `vchord_bm25` and `pg_tokenizer` extensions. These are user-installed and managed. See [VectorChord-BM25](#) for setup details.

Helper functions and metadata

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, test query);
```

Use this in place of `aidb.encode_text('<model>', '<query>')` when constructing custom vector queries.

`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>	<code>aidb.distanceoperator</code>	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;
```

`aidb.rerank_text()`

Optional: 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.

Query patterns

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) AS_
↪distance
FROM my_kb_vector kb
JOIN my_source_table s ON s.id::TEXT = kb.source_id
```

(continues on next page)

(continued from previous page)

```
WHERE s.category_id = 2 -- relational filter on source table
ORDER BY kb.value <=> aidb.kb_query_encode(public.my_kb_vector, summer fashion)
LIMIT 10;
```

Vector + BM25 as a filter

Use a BM25 score threshold as a WHERE condition, then order results by vector distance. This returns the semantically nearest results that also satisfy the keyword match.

```
SET hnsw.iterative_scan = relaxed_order;

SELECT s.id, s.description,
       kb.value <=> aidb.kb_query_encode(public.my_kb_vector, summer fashion) AS
--distance
FROM my_kb_vector kb
JOIN my_source_table s ON s.id::TEXT = kb.source_id
WHERE s.bm25_col <&> to_bm25query(my_bm25_idx, tokenize(shoes, bert)) < -1.0 -- BM25
--filter
ORDER BY kb.value <=> aidb.kb_query_encode(public.my_kb_vector, summer fashion)
LIMIT 10;
```

Vector + BM25 via Reciprocal Rank Fusion

Reciprocal Rank Fusion (RRF) runs semantic and keyword searches independently, then merges and scores the result sets. This avoids over-filtering and surfaces results that rank well in either dimension.

```
WITH semantic AS (
  SELECT source_id AS id,
         ROW_NUMBER() OVER (ORDER BY value <=> aidb.kb_query_encode(public.my_kb_
--vector, summer fashion) AS rank
FROM my_kb_vector
ORDER BY value <=> aidb.kb_query_encode(public.my_kb_vector, summer fashion)
LIMIT 50
),
fulltext AS (
  SELECT id::TEXT AS id,
         ROW_NUMBER() OVER (ORDER BY bm25_col <&> to_bm25query(my_bm25_idx,
--tokenize(shoes, bert))) AS rank
FROM my_source_table
ORDER BY bm25_col <&> to_bm25query(my_bm25_idx, tokenize(shoes, bert))
LIMIT 50
)
SELECT COALESCE(s.id, f.id) AS id,
       COALESCE(1.0 / (60 + s.rank), 0) + COALESCE(1.0 / (60 + f.rank), 0) AS rrf_
--score
FROM semantic s
FULL OUTER JOIN fulltext f ON s.id = f.id
ORDER BY rrf_score DESC
LIMIT 10;
```

The constant 60 in the RRF formula is the standard rank smoothing factor. Adjust it to control the relative weight of high-rank vs lower-rank results.

Vector + relational + BM25 (all three)

Combine all three filtering mechanisms in a single query:

```
SET hnsw.iterative_scan = relaxed_order;

SELECT s.id, s.description,
       kb.value <=> aidb.kb_query_encode(public.my_kb_vector, summer fashion) AS distance,
       s.bm25_col <&> to_bm25query(my_bm25_idx, tokenize(shoes, bert)) AS bm25_score
FROM my_kb_vector kb
JOIN my_source_table s ON s.id::TEXT = kb.source_id
WHERE s.category_id = 2
      -- relational filter
      AND s.bm25_col <&> to_bm25query(my_bm25_idx, tokenize(shoes, bert)) < -1.0 -- BM25 filter
ORDER BY kb.value <=> aidb.kb_query_encode(public.my_kb_vector, summer fashion)
LIMIT 10;
```

Worked examples

Example 1: 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.

Set up the source table and pipeline

Create the source table:

```
CREATE TABLE products (
  id          SERIAL PRIMARY KEY,
  product_name TEXT NOT NULL,
  category    TEXT NOT NULL
);
```

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),
(Sandwich,     meat),
(Veggie Burger, vegetarian),
(Kebab,        meat);
```

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
                      )
);

```

Run the pipeline to generate embeddings:

```

SELECT aidb.run_pipeline(food_products);

```

Standard semantic query (no filter)

```

SELECT * FROM aidb.retrieve_text(public.pipeline_food_products, fast food, topk => 4);

```

__OUTPUT__	key	value	distance	part_ids	intermediate_steps
8	Veggie Burger	0.9955745765192575	{0}	[]	
6	Pizza	1.0280680840689795	{0}	[]	
4	Fries	1.0630385212203115	{0}	[]	
7	Sandwich	1.0903594734917037	{0}	[]	

Custom query with relational filter

```

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)
LIMIT 10;

```

__OUTPUT__	source_id	product_name	category
8	Veggie Burger	vegetarian	
4	Fries	vegetarian	

Only vegetarian items are returned, ordered by semantic relevance to “fast food”.

Example 2: BM25 full-text search with RRF

This example shows how to combine semantic vector search with BM25 keyword search using Reciprocal Rank Fusion (RRF) to merge the two result sets. RRF re-ranks results by combining their positions in the semantic and keyword rankings, surfacing items that score well in both. This example builds on the same `products` table and `food_products` pipeline from Example 1.

Add a BM25 index to the source table

Enable the `pg_tokenizer` and `vchord_bm25` extensions:

```
CREATE EXTENSION IF NOT EXISTS pg_tokenizer;
CREATE EXTENSION IF NOT EXISTS vchord_bm25;
```

Create a tokenizer using the BERT model:

```
SELECT tokenizer_catalog.create_tokenizer(bert, $$
    model = "bert_base_uncased"
$$);
```

Add a BM25 vector column to the source table and populate it with tokenized values:

```
ALTER TABLE products ADD COLUMN embedding_bm25 bm25_catalog.bm25vector;
UPDATE products SET embedding_bm25 = tokenizer_catalog.tokenize(product_name, bert);
```

Create a BM25 index on the new column:

```
CREATE INDEX products_bm25_idx ON products USING bm25 (embedding_bm25 bm25_catalog.
↳bm25_ops);
```

Hybrid query with RRF

```
SET hnsw.iterative_scan = relaxed_order;
SET search_path TO public, tokenizer_catalog, bm25_catalog;

WITH semantic AS (
    SELECT source.id,
           source.product_name,
           source.category,
           ROW_NUMBER() OVER (ORDER BY vector.value <=> aidb.kb_query_encode(public.
↳pipeline_food_products, fast food) AS rank
    FROM public.pipeline_food_products AS vector
    JOIN products AS source ON source.id::TEXT = vector.source_id
    ORDER BY vector.value <=> aidb.kb_query_encode(public.pipeline_food_products,
↳fast food)
    LIMIT 20
),
keyword AS (
    SELECT id,
           product_name,
           ROW_NUMBER() OVER (ORDER BY embedding_bm25 <&> to_bm25query(products_bm25_
↳idx, tokenize(fast food, bert))) AS rank
    FROM products
    ORDER BY embedding_bm25 <&> to_bm25query(products_bm25_idx, tokenize(fast food,
↳bert))
    LIMIT 20
)
SELECT
    COALESCE(s.id, k.id) AS
↳id,
    COALESCE(s.product_name, k.product_name) AS
↳product_name,
```

(continues on next page)

(continued from previous page)

```

    s.category,
    s.rank
    AS_
  ↳ semantic_rank,
    k.rank
    AS_
  ↳ keyword_rank,
    COALESCE(1.0 / (60 + s.rank), 0) + COALESCE(1.0 / (60 + k.rank), 0)
    AS rrf_
  ↳ score
FROM semantic s
FULL OUTER JOIN keyword k ON s.id = k.id
ORDER BY rrf_score DESC
LIMIT 10;
__OUTPUT__
id | product_name | category | semantic_rank | keyword_rank | rrf_score
-----+-----+-----+-----+-----+-----
  ↳ -----
  1 | Hamburger   | meat     | 6 | 1 | 0.
  ↳ 03154495777446597119
  4 | Fries       | vegetarian | 3 | 4 | 0.
  ↳ 03149801587301587302
  6 | Pizza       | meat     | 2 | 6 | 0.
  ↳ 03128054740957966765
  8 | Veggie Burger | vegetarian | 1 | 8 | 0.
  ↳ 03109932497589199614
  3 | Fish n Chips | fish     | 7 | 3 | 0.
  ↳ 03079838900734423123
  5 | Burrito     | meat     | 5 | 5 | 0.
  ↳ 03076923076923076924
  2 | Cheeseburger | meat     | 9 | 2 | 0.
  ↳ 03062178588125292193
  7 | Sandwich    | meat     | 4 | 7 | 0.
  ↳ 03055037313432835821
  9 | Kebab       | meat     | 8 | 9 | 0.
  ↳ 02919863597612958227

```

The RRF score combines both rankings: items that rank highly in either semantic or keyword search surface at the top, with items that rank well in both receiving the highest scores.

Setting up user-managed BM25

BM25 is not managed by AI pipelines. You set it up independently on your source table, then reference it in your hybrid queries.

1. Install extensions

```

CREATE EXTENSION IF NOT EXISTS pg_tokenizer;
CREATE EXTENSION IF NOT EXISTS vchord_bm25;

```

2. Create a tokenizer

```

SELECT create_tokenizer(bert, $$
    model = "bert_base_uncased"
  $$);

```

3. Add a ``bm25vector`` column and index to your source table

```
ALTER TABLE my_source ADD COLUMN bm25_tokens bm25vector;
UPDATE my_source SET bm25_tokens = tokenize(description, bert);

CREATE INDEX my_source_bm25 ON my_source USING bm25 (bm25_tokens bm25_ops);
```

4. Keep the index current

New rows must be tokenized before they will appear in BM25 results. Use a trigger or update the column as part of your data ingestion process:

```
UPDATE my_source SET bm25_tokens = tokenize(description, bert) WHERE bm25_tokens IS_
→ NULL;
```

For full tokenizer configuration options, see [VectorChord-BM25](#) and [VectorChord-BM25](#) in `pg_extensions` .

Volume source limitation

Hybrid search with relational predicate filtering or BM25 is **not supported for volume-sourced knowledge bases**. Volume sources have no backing Postgres table to join against, so there is 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()` .

Vector extensions

AI pipelines integrates with two high-performance PostgreSQL extensions for vector search: **VectorChord** for dense semantic (embedding-based) search, and **VectorChord-BM25** for sparse keyword (BM25) search. Both are configured as part of the pipeline's embedding (knowledge base) step and complement the standard `pgvector` capabilities built into EDB Postgres AI.

Extension	Search type	Best for
<i>Enable VectorChord (optional)</i>	Dense vector (ANN)	Semantic similarity at billion-scale; lower latency than HNSW/IVFFlat at high dimensions.
<i>VectorChord-BM25</i>	Sparse keyword (BM25)	Exact keyword relevance ranking; complements dense search in hybrid retrieval.

Together, these extensions form the foundation for hybrid search pipelines — combining semantic meaning (VectorChord) with keyword precision (VectorChord-BM25) to produce higher-quality retrieval results.

For full extension documentation including installation and release notes, see:

- [Enable VectorChord \(optional\)](#)
- [VectorChord-BM25](#)

VectorChord-BM25

VectorChord-BM25 brings Best Matching 25 (BM25) keyword search into Postgres using the same high-performance framework as VectorChord. Where VectorChord handles dense semantic search on AI embeddings, VectorChord-BM25 handles sparse keyword search — finding documents that contain the right words, ranked by statistical relevance.

For installation, configuration, and full technical reference, see [VectorChord-BM25 documentation](#) .

When to use VectorChord-BM25

BM25 keyword search complements dense vector search in knowledge base pipelines. Use VectorChord-BM25 when:

- Users search with precise terminology, product names, or technical identifiers that semantic models may not rank highly.
- You want to build a hybrid retrieval pipeline that combines keyword relevance with semantic similarity.
- Your corpus is large and you need fast, indexed keyword lookup rather than sequential text scanning.

How it works

VectorChord-BM25 introduces a `bm25vector` data type and a `bm25` index. Text is tokenized into a sparse vector representation using a tokenizer (such as BERT), stored as a `bm25vector`, indexed with a BM25 index, and queried using the `<&` distance operator.

Unlike dense vector search, which requires an embedding model at query time, BM25 scoring is entirely statistical — no model inference is needed at retrieval time.

Using VectorChord-BM25 alongside pipelines

VectorChord-BM25 is not a pipeline step operation itself. Instead, it operates as a **parallel search layer** alongside your dense knowledge base. The typical pattern is:

1. Your pipeline populates a dense knowledge base (embedding-indexed table) via the `KnowledgeBase` step.
2. A separate BM25-indexed table stores the same or related content, tokenized with `tokenize()`.
3. At query time, results from both indexes are merged and re-ranked.

Step 1: Set up the tokenizer

Initialize a tokenizer before creating your BM25 table. BERT is the most common choice:

```
SELECT create_tokenizer(bert, $$
model = "bert_base_uncased"
$$);
```

Step 2: Create the BM25 table

Create a table with a `bm25vector` column alongside your pipeline's source or destination table:

```
CREATE TABLE documents_bm25 (
  id      serial PRIMARY KEY,
  source_id bigint,          -- references the pipeline source key
  passage  text,
  embedding bm25vector
);
```

Step 3: Populate and index

Load content, tokenize it, and build the BM25 index:

```
-- Tokenize text into bm25vector
UPDATE documents_bm25
```

(continues on next page)

(continued from previous page)

```

SET embedding = tokenize(passage, bert);

-- Create the BM25 index
CREATE INDEX documents_embedding_bm25
ON documents_bm25
USING bm25 (embedding bm25_ops);

```

Step 4: Query with BM25

Use the `<&>` operator and `to_bm25query()` to retrieve keyword-ranked results:

```

SELECT
    id,
    source_id,
    passage,
    embedding <&> to_bm25query(
        documents_embedding_bm25,
        tokenize(PostgreSQL full-text search, bert)
    ) AS bm25_score
FROM documents_bm25
ORDER BY bm25_score
LIMIT 10;

```

Combining BM25 with a dense knowledge base

The most effective retrieval pattern is hybrid search: running both a dense (semantic) query and a BM25 (keyword) query, then merging the result sets. This example runs both searches and unions the top results for downstream re-ranking:

```

-- Dense semantic results from the pipeline knowledge base
WITH dense_results AS (
    SELECT key, distance AS score, dense AS source
    FROM aidb.retrieve_key(my_knowledge_base, PostgreSQL full-text search, 10)
),

-- BM25 keyword results from the BM25 table
bm25_results AS (
    SELECT source_id::text AS key,
        embedding <&> to_bm25query(
            documents_embedding_bm25,
            tokenize(PostgreSQL full-text search, bert)
        ) AS score,
        bm25 AS source
    FROM documents_bm25
    ORDER BY score
    LIMIT 10
)

SELECT key, score, source
FROM dense_results
UNION ALL

```

(continues on next page)

(continued from previous page)

```
SELECT key, score, source
FROM bm25_results
ORDER BY score;
```

The merged results can then be passed to a re-ranking model (via `aidb.rerank_text()`) for a final relevance-ordered response.

Relationship to hybrid search

VectorChord-BM25 is one of the building blocks for the hybrid search pattern described in [Hybrid search](#). Dense search with VectorChord provides semantic breadth; BM25 provides keyword precision. Together they reduce both false positives (off-topic semantic matches) and false negatives (missed exact-match results).

Further reading

- VectorChord-BM25 documentation — installation, tokenizer configuration, reference, and release notes
- [Enable VectorChord \(optional\)](#) — dense vector indexing with pipelines
- [Hybrid search](#) — combining dense and sparse search
- [Pipeline step config helpers](#) — KnowledgeBase step configuration reference

VectorChord

VectorChord is a high-performance Postgres extension for dense vector search. It extends pgvector with index types that are faster and more memory-efficient at scale, making it well-suited for knowledge bases containing millions or billions of embeddings.

For installation and full technical reference, see VectorChord in `pg_extensions`.

When to use VectorChord

Standard pgvector HNSW and IVFFlat indexes work well for most workloads. VectorChord becomes the better choice when:

- Your knowledge base contains millions or billions of embeddings and HNSW indexes are consuming too much RAM.
- You need lower query latency at high throughput.
- Your embeddings have high dimensionality (above the 2000-dimension limit supported by pgvector's HNSW index).

Index types

VectorChord provides two index types, both configurable through the [Vector index config helpers](#).

chord_hnsw

A VectorChord implementation of HNSW with granular control over graph connectivity and search depth. Use this as a drop-in upgrade from pgvector's HNSW when you need more tuning control or better performance at scale.

Configure with `aidb.vector_index_chord_hnsw_config()`:

Parameter	Description
<i>m</i>	Maximum connections per node in the graph (default: 16).
<i>ef_construction</i>	Search depth during index build (default: 64).
<i>max_connections</i>	Hard cap on graph connections.
<i>ml</i>	Level multiplier controlling the graph layer structure.
<i>vector_data_type</i>	Vector storage type (e.g., <i>float32</i>).

chord_vchordq

A quantization-based index (IVF-RaBitQ) designed for maximum throughput at very large scale. It trades a small amount of recall accuracy for significantly faster searches and lower memory usage than HNSW.

Configure with `aidb.vector_index_chord_vchordq_config()` :

Parameter	Description
<i>lists</i>	Number of IVF clusters. Higher values increase accuracy at scale.
<i>spherical_centroids</i>	Use normalized centroids — recommended for cosine similarity.
<i>vector_data_type</i>	Vector storage type (e.g., <i>float32</i>).

Using VectorChord with pipelines

VectorChord index types are configured through the `vector_index` parameter of `aidb.knowledge_base_config()`, which is passed as `step_N_options` when creating a pipeline.

Configuring a pipeline with chord_hnsw

```
SELECT aidb.create_pipeline(
  name          => my_semantic_pipeline,
  source        => source_docs,
  source_key_column => id,
  source_data_column => content,
  destination   => my_knowledge_base,
  step_1        => KnowledgeBase,
  step_1_options => aidb.knowledge_base_config(
    my_embedding_model,
    Text,
    distance_operator => Cosine,
    vector_index      => aidb.vector_index_chord_hnsw_
    ↪config(
                                          m          => 16,
                                          ef_construction => 64,
                                          max_connections => 32
                                          )
    ),
  auto_processing => Live
);
```

Configuring a pipeline with chord_vchordq

```
SELECT aidb.create_pipeline(
  name           => my_large_scale_pipeline,
  source         => billion_scale_docs,
  source_key_column => id,
  source_data_column => content,
  destination    => my_large_kb,
  step_1         => KnowledgeBase,
  step_1_options => aidb.knowledge_base_config(
    my_embedding_model,
    Text,
    distance_operator => Cosine,
    vector_index      => aidb.vector_index_chord_vchordq_
    ↪config(
                                            lists           => 1000,
                                            spherical_centroids => true
                                            )
    ),
  auto_processing    => Background,
  background_sync_interval => 60 seconds
);
```

Multi-step pipeline with chunking and VectorChord

For document pipelines, chunk text first and then embed into a VectorChord-indexed knowledge base:

```
SELECT aidb.create_pipeline(
  name           => doc_pipeline_vchord,
  source         => raw_documents,
  source_key_column => id,
  source_data_column => body,
  destination    => doc_knowledge_base,
  step_1         => ChunkText,
  step_1_options => aidb.chunk_text_config(200, 250, 25, words),
  step_2         => KnowledgeBase,
  step_2_options => aidb.knowledge_base_config(
    my_embedding_model,
    Text,
    vector_index => aidb.vector_index_chord_hnsw_config(
      m => 32,
      ef_construction => 128
    )
    ),
  auto_processing    => Live
);
```

Choosing between index types

Scenario	Recommended index
Up to ~10M vectors, general use	<code>chord_hnsw</code>
100M+ vectors, memory-constrained	<code>chord_vchordq</code>
High-dimensional embeddings (>2000 dims)	<code>chord_vchordq</code>
Cosine similarity with normalized vectors	<code>chord_vchordq</code> with <code>spherical_centroids => true</code>
Drop-in pgvector HNSW replacement	<code>chord_hnsw</code>

Further reading

- [VectorChord documentation](#) — installation, index reference, and release notes
- [Vector index config helpers](#) — full parameter reference for all index config functions
- [Hybrid search](#) — combining VectorChord dense search with BM25 keyword search

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.

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)
```

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

Verify the files available for processing:

```
SELECT * FROM aidb.list_volume_content(volume__6545);
```

file_name	size	last_modified
Bilge Ince.html	8878	2025-02-19 09:58:29 UTC
all_tags.html	11170	2025-02-19 09:58:29 UTC
hello_world.html	108	2025-02-19 09:58:30 UTC
simple.html	662	2025-02-19 09:58:29 UTC

(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;
```

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)

CHAPTER 10

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`, `decode_text`, `rerank_text`) are in [Reference → Models](#).

Page	Functions	What it covers
Text chunking	<code>aidb.chunk_text()</code>	Splitting text into segments for LLM context windows
Data parsing	<code>aidb.parse_html()</code> , <code>aidb.parse_pdf()</code>	Extracting readable text from HTML and PDF inputs
Performing OCR	<code>aidb.perform_ocr()</code>	Extracting text from images using an OCR-capable model
Summarization cookbook	<code>aidb.summarize_text()</code> , <code>aidb.summarize_text_aggregate()</code>	Generating and aggregating AI-powered text summaries, use cases, and cookbook
AI inference functions	<code>aidb.encode_text()</code> , <code>aidb.decode_text()</code> , and variants	Embedding text and images into vectors; generating completions
Text reranking	<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](#).

10.1 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)

10.1.1 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.

10.2 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.

10.2.1 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 <strong>bold text</strong>, some <em>italic test</em>,
      and a <a href="https://google.com" target="_blank">link</a>.
    </p>

    <ol>
      <li>List item</li>
      <li>List item</li>
      <li>List item</li>
    </ol>,
  options => {"method": "StructuredPlaintext"} -- Default
);
```

__OUTPUT__

parse_html	

Hello, world!	+
	+
This is my first web page.	+
	+
It contains some bold text, some italic test, and 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.

10.2.2 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 (255044462d312e340a25b89a929d0a312030206f626a3c3c2f547970652f436174616c6f672f506167657320332
  ↳ hex),
  options => {"method": "Structured", "allow_partial_parsing": true} -- Default
);

__OUTPUT__
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.

10.3 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

segment. Some providers have format restrictions; for example, the NVIDIA NIM provider currently supports only png and jpeg formats.

Step 1: Register an OCR-capable model

```
SELECT aidb.create_model(
  my_paddle_ocr_model,
  nim_paddle_ocr,
  credentials => {"api_key": "__NVIDIA_NIM_API_KEY__"}::JSONB
);
```

Step 2: Extract text from images

```
SELECT * FROM aidb.perform_ocr(
  decode (/9j/4AAQSkZJRgABAQAAQABAAAD/2wCEAAUFBUFBQUFBGUICAcIC
    AsKCQkKCxEMDQwNDBEaEBMQEBMQGhcbFhUWGxcpIBwcICkvJyUnLzkzMzlhRE
    ddXX0BBQUFBQUFBQUYGBQgIBwgICwoJCQoLEQwNDA0MERoQExAQExAaFxsWFRYb
    FykgHBWgKS8nJScvOTMzOUdER11df/CABEIAWYDKgMBIgACEQEDEQH/xAAvAA
    EAAQUBAQEAAAAAAAAAAAAAAAAQIDBAcIBgkFAQEAAAAAAAAAAAAAAAAAA/9oA
    DAMBAAIQAxAAAADsuKLhCRRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCR
    CRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCRCRE2rpTVT
    UAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAWrtq6U1U1AAAA
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAf7aulNVNQAAAAAAAA
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABau2rpTVTUAAAAAAAA
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAWrtq6U1U1AAAAAAAAAAAAAAAA
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAf7aulNVNQAAAAAAAAAAAAAAAA
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABau2rpTVTUAAAAAAAAAAAAAAAA
    AAAAAAAAAAAAAAAAAAAWrtq6U1U1AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    AAAAAAAAAAAAAAAAAAAf7aulNVNQAAAAAAAAAAAAAeS/INiNdjYj8z9MAAAAA
    FJUiQAAAAAAAAA8l60AAAAAAAAAAAAAtXbV0pqpqAAAAAAAAAESNE617AwD5kd
    Jap+gp5L2PC/459AJ5c/ROko4DxD6EU+I4rPoY+enVJuJzfpI79ngXrU2LT88Pf
    naEcVfhHek8K9jno6fnrmH0A5c/S5Y089icId3FSOODsiOC847meN4qPoPPzx6i
    N0zw9hHeD5yeq08Y+cfeZ6ev5uepO+nzz71OD/AKDfPnsc2BHz7xj6HPBe9AAAA
    AAAAAALV21dKaqaGAAAAAAAAABgZ+AcB9U8rddHGO7vc9XHz43/me6NMbI3Jxwd
    YcV9C7HNYbf17pY1f0lo3uI4n3V+3+oeE745D6qNBYHPvdp89ulde78PWb15730as
    1P+h+cbw2VrDZw4F7757Nd+52Vwsd76d8b6I8H6LZ2sjWvf8AwN9Bjh7qvlbrY0P7
    bxfuDkP6Q/M/6UnPuDbzZRP6f5npD9GxuT9Y557i576EAAAAAAAAAALV21dKaqaG
    AAAAAAAAAB+f8AoDkPrmscZeZ7zHN3ot3jgnI7uHhuRe8xbw0NuwaL5774HEXUXu
    xzt0HeHCE93DgLq7Z44Rx+9hqWxuEcQ9vRJHMHUA40/Y7ZGpOWfoCOJMTuYci9cVD
    mPf8A+6NIei2bQfMjZn5vf5wr2z+kOReivYDgivvMeI9wAAAAAAAAAAf7aulNVNQ
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAiRh5gAAAAAAAAAAAAAAAAWrtq6U1U1AAAA
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAf7aulNVNQAAAAAAAAAAAAAAAA
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABau2rpTVTUAAAAAAAAAAAAAAAA
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAWrtq6U1U1AAAAAAAAAAAAAAAA
    AAAAAAAAAAAAAAAAAAAf7aulNVNQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    AAAAAAAWrtq6U1U1AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAf7
    7aulNVu2ZDHGQxxkMcZDHGQxxkMcZDHGQxxkMcZDHGQxxkMcZDHGQxxkMcZDHGQxxkMc
    cZDHGQxxkMcZDHGQxxkMcZDHGQxxkMcZDHGQxxkMcZDHGQxxkMcZDHGQxxkMcZDHGQ
    xxxkMcZDHGQxxkMcZDHGQxxkMcZDHGQxxkMcZDHfD2xfIgAAAAAAAAAAAAAAAAAAAA
    AAAAAAAAAAAAAAAAAAAAAAAIrD//EADIQAAAFBAEEAQMDAgcBAAAAAABAwQFAg
    YHERQIEDIOYBiHQBYxNxUXExgiMzZBUTX/2gAIAQEAAQgAM/2IaMaMaMaMaMaMaMa
    aMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMa
    aMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMa
    aMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMa
```

(continues on next page)

(continued from previous page)

```

aMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMaMGejIjCn+9R8kr80+1HsJfJK/
NPtR7CXySvzT7Uewl8kr80+1HsJfJK/NPtR7CXySvzT7Uewl8kr80+1HsJfJK/NPtR7CXy
SvzT7Uewl8kr80+1HsJfJK/NPtR7CXySvzT7Uewl8kr80+1HsJfJK/NPtR7CXySvzT7Uewl
8kr80+1HsJfJK/NPtR7CXySvzT7Uewl+PWJq+bPtt3Q0mv7s40H92caA8uY2IvtEzEXOsUZCK
/FMGYI/w6vsQjr6tGWlFIljTv/v8qvzT7Uewl+PV+32yLgyLyLO0y7qS6V4JjHP3dQLLBaZCu
v8Aobk+k6AFiWehYtuM4Fv/AKgRgxVVsEezBjf3BmewRgz0NmCP7gwdX/h1aLY2exv/AMI/uDF
VWwZiqrY6nZyZhIi0zjMSvXL7G9ouXdPYx9QIwZ6BfuDq+w+obH1D6h9Q+ob0Nj6hsGMOFzu9H
iPqBH+TX5p9qPYS/IOLiPcFOjplL68nBdZJuiqsreWX76v+4lLesZ/jvPVutq5YsIZrf308/TNy
Z3y5I2OTSfhY7Hud7uaJTJxuRsoYkuNCPuS5L4joCyFrtDGYzJmSQd1xU3B5xxeknLr4ayVXkS3
FFHubsyubKqRgYJpYGe7sbUzNdr5ZyDjC4k4a8MgTSieOLgmIiAzpdUPa0qzPEUNl13e8RO3F1D
Mt0zN0r2jZr4kzyuib6vFOWrzgLOZWrdFz3JHWlBSM3Jr35ljME8sxxgZayc8WEzrmDw9lx9kSM
mIaRyVCZPim8VXe2LbfzKudmvWlAqPQzNma40blVs+1KMUZ7WQ/qJ42zFeFrXY3te8r7vBnY1tSE
46j5TNmY3rpeMuC182Y1bFMr4OyO8yFbjs5LM+QbptfKRos6Es5Zg/x5dkyvNJ2ILppYzNUNmjND
p28g5STy/h+XZFI2fciV22nDzidWYb7gLunq6GdjZ5vZrTMqxeSsoYluGiPuGCmWdww7CUZYc/nZ6M
jX0yx/bjiVcsl83ZiXcPWD93mrD7xs6kMb36wyBbbeUQ/Hr80+1HsJfk3Ee4KdHTL/ACcM9SriIxj
OKIYeynb+M05ZV5/mxt0K3SweZQZXFCDr2PZ2adR91RNn9TEtAs20XcJ3zhLLTqPSnb+sei4McvrYh
rMv28sLSD+NcodRWPLqj14q48cwOP4ti5fWYtdbOjKbm5Zz/NfbtJERZgytb2TG0WbS2ZNaT6ZJm1
uma02U7c0tLvf+vs6h8K4muKidcSnVLaTYq6Y5tcLy6svQU296q5RZC2bYjKOnSAaxWN457Qommqn
WnXaOfRqsqfonYnqxMv6PZowyZHiqySMVHovvmmxrqtW/nV3xtu9VWkk0biJHuD8qz9EkD9Wnbl4Q
psrijMq4cxdG1QUHkbqGQu+3ZSChukv1r2GfUKHeYTQrYMGsYyaMWvVikmU9ai90No5tH2LaTzr1V
oUfo6AVPAf8VQXx5ANbjzXWzdU0kRERds1uspGw/wCrH0uy67yy5Rirhn+eXY6pZZda7ouMFsdSNqW
xARcO1vDqLtS7rblIRx0pyaqc9cUaZfj1+afaj2EvybiPcFOjpl/k4ZphlZvGlxtkenJGyJn+sQk//
↪bT
↪H5Ee17xxknfQnqxV75zj7Euv90O5CybEu9qi5eZ1xRbthpxsnCWdlF3a+E4K4ZOyrstTMMCu8Xv7AtgPo
eUkGHTBKvm15ykrSSzibQzc5bXCjJRHLpFNZvk55ivGqEcS8k6jpDAs6/
↪jekoqDZ3mZz7xWMgZyQRxHbLXK
GQXn6lj7CseCR+prPS8ZOZ7YvYvqohl3Vq29LJ9N11sZexW8IcjJM4pg6fvsU5ou/
↪IN81xVfVgW4izRhgojx, base64),
  options => {"model": "my_paddle_ocr_model"}
);

__OUTPUT__
create_model
-----
my_paddle_ocr_model
(1 row)

part_id |      text
-----+-----
0 | Tesseract sample
(1 row)

```

10.4 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](#).

10.4.1 aidb.summarize_text()

Step 1: Register a model

```
SELECT aidb.create_model(my_t5_model, t5_local);
```

Step 2: Summarize a text string

```
edb_notranlate_1 sql SELECT aidb.summarize_text( input => 'Long article text...', options => aidb.summarize_text_config( 'my_t5_model', prompt => 'Summarize the key points for a non-technical audience' )::json );
```

```
edb_notranlate_2 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 );
```

10.5 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 ``

10.6 Text summarization

<i>Provider</i>	<i>'bert_local'</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:</i>	<i>'desired_length' must be greater than 0</i>	Invalid chunk config	Use a positive integer for <i>desired_length</i>
<i>Invalid parameters:</i>	<i>'max_length' must be >= 'desired_length'</i>	<i>max_length</i> is smaller than <i>desired_length</i>	Set <i>max_length</i> >= <i>desired_length</i>
<i>Invalid parameters:</i>	<i>'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</i>	<i>reduce strategy produced output longer than input failed to reduce size</i>		Increase chunk size (use >= 50 words or >= 800 chars)

10.6.1 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

10.6.2 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
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,
                    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
```

(continues on next page)

(continued from previous page)

```
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,
                       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
```

(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;

```

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,

```

(continues on next page)

(continued from previous page)

```

    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)
       ) 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
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

```

(continues on next page)

(continued from previous page)

```

--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
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
    )
  )
```

(continues on next page)

(continued from previous page)

```

    )
  )::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
         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.
```

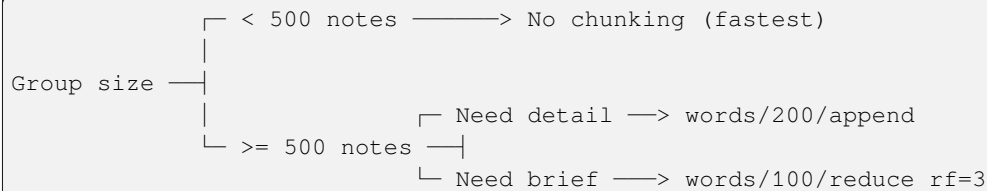
3. Extractive pre-filtering

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,
  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
Total per call	~3–12s

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.

10.7 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.

10.7.1 `aidb.encode_text()`

Encodes a single text string into a vector using a registered embedding model.

```
SELECT aidb.encode_text(
  input    => What is pgvector?,
  options  => {"model": "my_embedder"}
);

__OUTPUT__
          encode_text
-----
[0.021,-0.043,0.117, ...]
(1 row)
```

For the full parameter reference, see [Models reference](#).

10.7.2 `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(
  my_model_name,
  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](#).

10.7.3 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(  
  my_model_name,  
  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](#).

10.7.4 aidb.decode_text()

Generates text from a language model (LLM) given a prompt. This is the core function for LLM inference in AIDB.

```
SELECT aidb.decode_text(  
  model_name => my_model_name,  
  input      => Explain what a knowledge base is in two sentences.,  
  inference_config => NULL  
);  
  
__OUTPUT__  
                                decode_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 in options:

```
SELECT aidb.decode_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  
  )  
);
```

For the full parameter reference, see [Models reference](#).

10.7.5 aidb.decode_text_batch()

Generates text responses for multiple prompts in a single batched request.

```
SELECT * FROM aidb.decode_text_batch(
  model_name => my_model_name,
  input      => ARRAY[
    What is a vector database?,
    What is retrieval-augmented generation?
  ]
);
```

__OUTPUT__

id	value
0	A vector database stores high-dimensional vectors ...
1	Retrieval-augmented generation (RAG) combines a ...

(2 rows)

For the full parameter reference, see [Models reference](#).

10.7.6 aidb.inference_config()

Builds an inference configuration object to control language model generation behavior. Pass the result as the "inference_config" key inside the options JSONB of `decode_text()`, `decode_text_batch()`, or `summarize_text()`.

```
SELECT aidb.decode_text(
  input      => Write a haiku about PostgreSQL.,
  options    => jsonb_build_object(
    model,          my_llm,
    inference_config, aidb.inference_config(
      temperature => 0.9,
      max_tokens  => 64,
      seed        => 42
    )
  )
);
```

For the full parameter reference, see [Models reference](#).

!!!Note To use these functions as steps inside a pipeline, see [Pipeline steps](#).

10.8 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

```

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

```

Step 2: Rerank a set of candidates

```

SELECT * FROM aidb.rerank_text(
  query  => What is retrieval-augmented generation?,
  inputs => 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.
  ],
  options => [{"model": "my_reranker"}]
)
ORDER BY relevance_score DESC;

__OUTPUT__
id | relevance_score
--|-----
 2 |           0.97
 0 |           0.91
 1 |           0.43
(3 rows)

```

The `id` column corresponds to the zero-based index of each string in the `inputs` array, allowing you to join back to the original data.

For the full parameter reference, see [Models reference](#).

10.8.1 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
    source_id,
    content
  FROM aidb.retrieve_text(my_kb, What is RAG?, 20)
  JOIN my_source_table ON source_id = id::TEXT
)
SELECT
  c.source_id,
  c.content,
  r.relevance_score
FROM candidates c
JOIN aidb.rerank_text(
  query  => What is RAG?,
  inputs => ARRAY(SELECT content FROM candidates),

```

(continues on next page)

(continued from previous page)

```
options => {"model": "my_reranker"}
) r ON r.id = (
  SELECT array_position(ARRAY(SELECT content FROM candidates), c.content) - 1
)
ORDER BY r.relevance_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.

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
Models reference	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>decode_text</i> , <i>rerank_text</i>), and all model config helpers	Integrating models
Knowledge bases reference	Knowledge base views and stats, retrieval functions (<i>retrieve_key</i> , <i>retrieve_text</i>), volume functions, and auto-processing	Knowledge bases reference
Operations reference	Pipeline error log functions (<i>get_error_logs</i> , <i>clear_error_logs</i> , <i>retry_pipeline_errors</i>)	Pipeline error log
Functions reference	Standalone SQL functions: <i>chunk_text</i> , <i>parse_html</i> , <i>parse_pdf</i> , <i>perform_ocr</i> , <i>summarize_text</i>	SQL functions
Pipelines reference	Pipeline types, views, CRUD functions, step config helpers, and vector index config helpers	Reference → Functions

11.1 Models reference

Reference for all model-related functions and views in AIDB. For guide-style documentation, see [Integrating models](#).

11.1.1 Catalog views

`aidb.model_providers`

Lists all available model providers registered in the system.

Column	Type	Description
<i>server_name</i>	name	Name of the model provider
<i>server_description</i>	text	Description of the provider
<i>server_options</i>	text[]	Available configuration options

aidb.models

Lists all models in the registry, including built-in and user-created models.

Column	Type	Description
<i>name</i>	text	User-defined name for the model
<i>provider</i>	text	Model provider name
<i>options</i>	jsonb	Configured options for the model

Example

```
SELECT * FROM aidb.models;
```

```
__OUTPUT__
name | provider | options
-----+-----+-----
bert | bert_local | {"config={}}"
clip | clip_local | {"config={}}"
t5   | t5_local  | {"config={}}"
(3 rows)
```

11.1.2 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 aidb.model_providers).
<i>config</i>	JSONB	'{}'	Provider-specific configuration. Build with a Model config helpers .
<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.

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"
}
```

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(
  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>	jsonb	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. Does not 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 as JSONB.

Example

```
SELECT aidb.delete_model(t5);

__OUTPUT__
delete_model
-----+-----+-----
(t5,t5_local,{"config={}"})
```

(1 row)

11.1.3 HCP model functions

These functions manage models running on EDB Hybrid Manager (HCP).

aidb.list_hcp_models

Lists models currently running on the Hybrid Manager.

Returns

Column	Type	Description
<i>name</i>	text	Model instance name on HCP

continues on next page

Table 8 – continued from previous page

Column	Type	Description
<i>url</i>	text	API endpoint URL
<i>model</i>	text	Model identifier

Example

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

```
__OUTPUT__
      name                |                url
-----+-----
 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, deleted, unchanged, or skipped</i>
<i>model</i>	text	Name of the synchronized model

11.1.4 Inference functions**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>text</i>	TEXT	Text to encode.

Returns

VECTOR — the embedding for the input text.

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>text</i>	TEXT[]	Array of strings to encode.

Returns

TABLE(id INT, value VECTOR) — one row per input, preserving input order via id.

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>image</i>	BYTEA	Raw image bytes.

Returns

VECTOR — the embedding for the input image.

Example

```
SELECT aidb.encode_image(clip_local, pg_read_binary_file('/tmp/photo.jpg')::BYTEA);
```

aidb.encode_image_batch

Encodes an array of binary images into vectors in a single call.

Parameters

Parameter	Type	Description
<i>model_name</i>	TEXT	Name of the registered model.
<i>images</i>	BYTEA[]	Array of raw image bytes.

Returns

TABLE(id INT, value VECTOR) — one row per input image, preserving input order via id.

aidb.decode_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>text</i>	TEXT	Required	Text prompt or input.
<i>inference_config</i>	JSON	NULL	Runtime inference settings. Use <code>:ref:``aidb.inference_config()``<#aidbinference_config>`</code> .

Returns

TEXT — the generated response.

Examples

```
-- Basic usage
SELECT aidb.decode_text(t5_local, translate to French: Hello, world.);

-- With inference configuration
SELECT aidb.decode_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.decode_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><#aidbinference_config>`</code> .

Returns

TABLE(id INT, value TEXT) — one row per input prompt.

Example

```
SELECT * FROM aidb.decode_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;

```

11.1.5 Inference configuration helper

aidb.inference_config

Builds a JSONB configuration object for runtime inference settings. Pass the result (cast to `json`) to `aidb.decode_text()`, `aidb.decode_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><think></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.decode_text(
  my_llama,
  What is machine learning?,
  aidb.inference_config(temperature => 0.5, max_tokens => 150)::json
);

- - Full configuration
SELECT aidb.decode_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.decode_text(
  my_reasoning_model,
  Solve: 2x + 5 = 15,
  aidb.inference_config(thinking => false)::json
);

```

11.1.6 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.

continues on next page

Table 20 – continued from previous page

Parameter	Type	Default	Description
<i>is_hcp_model</i>	BOOLEAN	NULL	<i>true</i> if model runs on HCP.

```

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.
<i>thinking</i>	BOOLEAN	NULL	<think> tag handling.
<i>max_tokens</i>	JSONB	NULL	Max tokens config (use <i>aidb.max_tokens_config()</i>).
<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.

```

SELECT aidb.create_model(
  my_llm,
  openai_completions,
  config => aidb.completions_config(
    model      => gpt-4o,
    api_key    => sk-...,
    temperature => 0.2
  )
);

```

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'.

```
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.

```
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.

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

aidb.llama_config

For the `llama_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	NULL	Sampling temperature.
<i>top_p</i>	DOUBLE PRECISION	NULL	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.
<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.

```
SELECT aidb.create_model(my_llama, llama_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.

```
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).

```
SELECT aidb.create_model(my_gemini, gemini,
    config => aidb.gemini_config(AIza..., model => gemini-2.0-flash));
```

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.

```
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.

```
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.

```
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 <i>aidb.max_tokens_config()</i>).

```
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.

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

11.2 Knowledge bases reference

Reference for knowledge base functions and views. For guide-style documentation, see [Knowledge bases reference](#).

11.2.1 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>	<i>aidb.DistanceOperator</i>	Vector distance function used for retrieval
<i>distance_operator_sql</i>	text	The distance operator in SQL-operator syntax (e.g. <code><-></code> , <code><=></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_stats

Also accessible as `aidb.kbstat`. Provides current processing statistics for all knowledge base pipelines.

raw-latex: `aidb.knowledge_base_stats` View “aidb.knowledge_base_stats” Column | Type | Collation | Nullable | Default
 name | text || pipelines | integer || <- number of pipelines attached embeddings | bigint || <- this is the old “count(embeddings)” status | text || <- the same status field we have in “aidb.pipeline_metrics” but we collect the “worst” status of all the connected pipelines

Example

```
SELECT * from aidb.kbm;
```

```
__OUTPUT__
      name | pipelines | embeddings | status
-----+-----+-----+-----
public.pipeline_pipeline__7471a | 1 | 5 | UpToDate
public.pipeline_pipeline__7471b | 1 | 5 | UpToDate
public.pipeline_animal_facts_kb | 1 | 372 | UpToDate
public.pipeline_animal_facts_kb_bert | 1 | 372 | UpToDate
public.mpkb_shared_vectors | 2 | 4 | UpToDate
(5 rows)
```

11.2.2 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.bulk_embedding()</i>

11.2.3 Knowledge base functions**aidb.set_auto_knowledge_base**

Updates the auto-processing mode for an existing knowledge base.

Parameters

Parameter	Type	Default	Description
<i>knowledge_base_name</i>	TEXT	Required	Name of the knowledge base.
<i>mode</i>	<i>aidb.PipelineAuto-ProcessingMode</i>	Required	New auto-processing mode.
<i>batch_size</i>	INTEGER	NULL	Records processed per batch (Background and Disabled modes).
<i>background_sync_interval</i>	INTERVAL	NULL	Polling interval (Background mode).

Examples

```
SELECT aidb.set_auto_knowledge_base(my_kb, Live);
SELECT aidb.set_auto_knowledge_base(my_kb, Background, background_sync_interval => 1
↪minute);
SELECT aidb.set_auto_knowledge_base(my_kb, Disabled, batch_size => 200);
```

aidb.retrieve_key

Returns the source record keys and distances for the top matching embeddings, without fetching source data. In *part_ids*, the call also returns, for each step in a multi-step pipeline, the IDs of the parts that match the query. For example if a single source record is split into 10 parts by a “ChunkText” step, then *part_ids* will indicate which chunk matched.

Parameters

Parameter	Type	Default	Description
<i>knowledge_base_name</i>	TEXT	Required	Name of the knowledge base.
<i>query</i>	TEXT	Required	Query text to search with.
<i>topk</i>	INTEGER	1	Number of results to return.

continues on next page

Table 38 – continued from previous page

Parameter	Type	Default	Description
<i>deduplicate</i>	BOOLEAN	<i>true</i>	Return each source result only once even if multiple "parts" (e.g. chunks/pages) match

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 i.e. pipeline step results match (chunk or page)
<i>pipeline_name</i>	text	The name of the pipeline processing this source record

Example

```
SELECT * FROM aidb.retrieve_key(public.pipeline_animal_facts_kb, birds, topk=>3);
```

```
__OUTPUT__
key | distance | part_ids | pipeline_name
-----+-----+-----+-----
5 | 1.1931772046185758 | {0,0} | animal_facts_kb
93 | 1.1980633963685476 | {2,0} | animal_facts_kb
98 | 1.2150919866080878 | {2,0} | animal_facts_kb
(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*, the call also returns, for each step in a multi-step pipeline, the IDs of the parts that match the query. For example if a single source record is split into 10 parts by a “ChunkText” step, then *part_ids* will indicate which chunk matched. The return column *intermediate_steps* will contain the actual result text of the intermediate step results.

Parameters

Parameter	Type	Default	Description
<i>knowledge_base_name</i>	TEXT	Required	Name of the knowledge base.
<i>query_string</i>	TEXT	Required	Query text to search with.
<i>number_of_results</i>	INTEGER	0	Number of results to return. Uses <i>topk</i> default if 0.
<i>knowledge_base_name</i>	TEXT	Required	Name of the knowledge base.
<i>query</i>	TEXT	Required	Query text to search with.
<i>topk</i>	INTEGER	1	Number of results to return.

continues on next page

Table 40 – continued from previous page

Parameter	Type	Default	Description
<i>deduplicate</i>	BOOLEAN	<i>true</i>	Return each source result only once even if multiple "parts" (e.g. chunks/pages) match

Returns

Column	Type	Description
<i>key</i>	text	Source record key
<i>value</i>	text	Source text content
<i>distance</i>	double precision	Vector distance from the query
<i>part_ids</i>	bigint[]	Which parts i.e. pipeline step results match (chunk or page)
<i>pipeline_name</i>	text	The name of the pipeline processing this source record
<i>intermediate_steps</i>	jsonb	The matching results from the intermediate steps; i.e. the values belonging to "part_ids"

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

Removes a knowledge base; i.e. its destination vector table and its configuration. Also removes all attached Pipelines. Does not delete the underlying source table.

Parameters

Parameter	Type	Description
<i>knowledge_base_name</i>	TEXT	Name of the knowledge base to delete.

Example

11.2.4 Volume functions

aidb.create_volume

Creates an AIDB volume from a PGFS storage location for use as a knowledge base data source. See [Configuring external storage](#) for setting up the PGFS storage location first.

Parameters

Parameter	Type	Description
<i>name</i>	TEXT	Name for the volume.
<i>server_name</i>	TEXT	Name of the PGFS storage location.
<i>path</i>	TEXT	Sub-path within the storage location.
<i>mime_type</i>	TEXT	Data type: <i>Text</i> , <i>Image</i> , or <i>Pdf</i> .

Example

```
SELECT aidb.create_volume(pdf_volume, my_s3_location, /, Pdf);
```

aidb.list_volumes

Lists all AIDB volumes in the database.

```
SELECT * FROM aidb.list_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.

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

11.3 Operations reference

Reference for pipeline error log functions. For guide-style documentation, see [Pipeline error log](#).

11.3.1 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>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>	TEXT	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>	text	Operation name (for example, <i>ParsePdf</i> , <i>ChunkText</i>)
<i>error_message</i>	text	Error description
<i>error_category</i>	aidb.ErrorBlocking	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

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>pipeline_name</i>	TEXT	Name of the pipeline.

Returns

Column	Type	Description
<i>pipeline_step</i>	smallint	Step number
<i>step_operation</i>	text	Operation name
<i>error_category</i>	text	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
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>	text	Operation name
<i>error_category</i>	text	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>pipeline_name</i>	TEXT	Name of the pipeline.
<i>error_ids</i>	BIGINT[]	Array of error log entry IDs to delete.

Returns

INTEGER — the number of rows deleted.

Example

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

__OUTPUT__
clear_error_logs
-----
2
(1 row)
```

aidb.retry_pipeline_errors

!!!Note `aidb.retry_pipeline_errors()` is not yet available. It will be included in a future release. See [Known issues](#).

Re-processes specific error log entries from the step where they originally failed.

Parameters

Parameter	Type	Description
<i>pipeline_name</i>	TEXT	Name of the pipeline.
<i>error_ids</i>	BIGINT[]	Array of error log entry IDs to retry.

Returns

Column	Type	Description
<i>error_id</i>	bigint	Error log entry ID
<i>status</i>	text	<i>resolved</i> or <i>failed</i>
<i>error_message</i>	text	Failure reason (empty if resolved)

Resolved entries are removed from the error log. Failed retries remain with an incremented `retry_count` and updated `last_retry_at`.

Example

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

__OUTPUT__

```
error_id | status | error_message
-----+-----+-----
      1 | resolved |
      2 | failed  | failed to extract text from page 0
(2 rows)
```

11.3.2 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).

11.4 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`, `decode_text`, `rerank_text`, and related), see [Inference functions](#).

11.4.1 `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 (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>	integer	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}
);
```

11.4.2 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"}
);
```

11.4.3 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).

continues on next page

Table 60 – continued from previous page

Key	Type	Default	Description
<i>allow_partial_parsing</i>	boolean	<i>true</i>	When <i>true</i> , continues parsing when errors are encountered on individual pages, returning as much data as possible.

Returns

Column	Type	Description
<i>part_id</i>	integer	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}
);
```

11.4.4 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_ocr</i> provider).

Returns

Column	Type	Description
<i>part_id</i>	integer	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"}
);
```

11.4.5 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>decode_text</i> .
<i>prompt</i>	text	(standard prompt)	Custom instruction to guide the summary style, for example <i>'Summarize for a 5th grader'</i> .
<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	<i>'append'</i>	Summarization strategy: <i>'append'</i> (summarize each chunk independently and concatenate) or <i>'reduce'</i> (iteratively summarize until the desired length is reached).
<i>reduction_factor</i>	integer	3	Used with the <i>'reduce'</i> 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"}
);
```

11.4.6 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). Note that `options` is **required** for the aggregate, not optional.

Returns

TEXT — the summary for the group.

Usage

```
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;
```

11.4.7 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

continues on next page

Table 68 – continued from previous page

Function	Parameter	SQL default	Runtime default
<i>summarize_text_config</i>	<i>strategy</i>	<i>NULL</i>	<i>'append'</i>
<i>summarize_text_config</i>	<i>reduction_factor</i>	<i>NULL</i>	<i>3</i>
<i>summarize_text_config</i>	<i>inference_config</i>	<i>NULL</i>	Provider defaults
<i>chunk_text_config</i>	<i>desired_length</i>	— (required)	—
<i>chunk_text_config</i>	<i>max_length</i>	<i>NULL</i>	Same as <i>desired_length</i>
<i>chunk_text_config</i>	<i>overlap_length</i>	<i>NULL</i>	<i>0</i>
<i>chunk_text_config</i>	<i>strategy</i>	<i>NULL</i>	<i>'chars'</i>

11.5 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 step config 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 config 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>