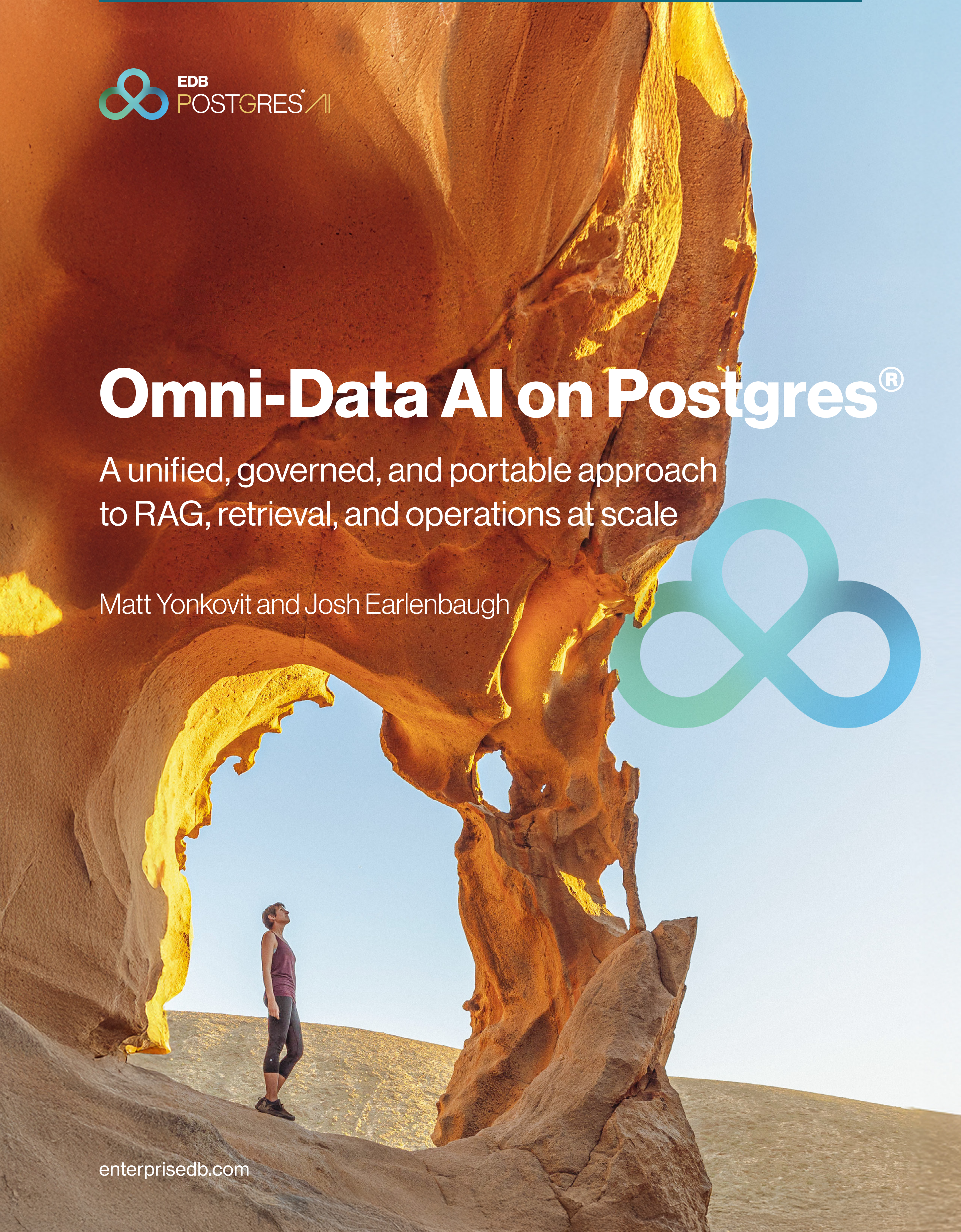

EDB POSTGRES® AI

Omni-Data AI on Postgres®

A unified, governed, and portable approach to RAG, retrieval, and operations at scale

Matt Yonkovit and Josh Earlenbaugh

enterprisedb.com

Table of Contents

Introduction to EDB Postgres AI Factory AI Pipelines and aidb

Overview

When accuracy, governance, and speed decide outcomes, keeping AI data and operations in one place becomes essential. This paper describes how EDB Postgres (PostgreSQL as packaged and supported by EnterpriseDB, or EDB), combined with pgvector and EDB Postgres AI Factory AI Pipelines, provides an omni-data foundation in which relational records, JSON documents, and vector embeddings are stored and queried together. By expressing ingestion, chunking, embedding, and refresh as first-class SQL operations, teams gain a dependable way to prepare data for retrieval-augmented generation (RAG) and to run semantic retrieval with the same reliability and controls they expect from core databases. By consolidating preparation and retrieval in one database, teams shorten time to first feature and gain clearer cost lines (storage, indexing, tokens) without the overhead of extra services and sync jobs.

AI Pipelines and the knowledge base

At the heart of this approach is AI Pipelines, a core capability of the EDB Postgres AI (EDB PG AI) platform, implemented by the aidb extension that formalizes AI data preparation. AI Pipelines turn document ingestion, content-aware chunking, and embedding into declarative steps that the database can schedule and monitor. The output is an AI knowledge base, represented as an aidb retriever asset: a set of generated tables containing vector embeddings plus the metadata and automation required to keep them in sync. To make terminology precise in this document: *knowledge base* (KB) refers to the generated Postgres tables (with PGVector embeddings and metadata) produced by aidb; an *aidb retriever* ***asset*** is the callable handle that uses those tables for similarity search via SQL. In practice, the KB follows the same backup/restore, replication, and point-in-time recovery procedures as any Postgres data, and its metadata/export routines make rebuilds repeatable and fast.

Retrieval and RAG

Because embeddings reside inside Postgres using pgvector, applications perform a similarity search via SQL and can combine it with transactional or metadata filters—such as jurisdiction, effective dates, or product lines—to narrow candidates before any reranking. This arrangement reduces false positives, stabilizes latency and cost, and makes retrieval observable with standard database tooling. It also aligns naturally with RAG: a precise vector search gathers context; the large language model (LLM) generates an answer that can be cited back to the stored sources. Because embeddings remain in Postgres, switching model providers or cloud endpoints becomes a configuration change—not a data migration—preserving optionality as models and economics evolve.

Information design and operations

Information design is central, as will be shown later in this document. Chunking, which is the process of breaking down larger data or information into smaller parts, should fit the content, not just the model's limits. A content-aware policy (typically around 200–300 tokens with modest overlap) preserves local references while maintaining precision. When warranted by evaluation, maintaining both sentence- and paragraph-level embeddings allows retrieval to choose the right granularity without sacrificing coherence. These policies can be defined and executed in aidb so that chunking remains consistent, reproducible, and auditable over time.

Operationally, the system favors throughput and resilience. For high-churn sources, queue-based workers generate embeddings in bulk, avoiding per-row triggers on hot OLTP tables and keeping write latency predictable. Knowledge-base tables follow the same replication, backup/restore, and PITR patterns as other Postgres data, and index maintenance (for example, HNSW/IVFFlat rebuilds) can be rehearsed alongside failover. Observability focuses on user-visible and operator-critical signals alike: p95 retrieval latency, embedding lag from landed to searchable, index health, and error budgets.

Security and sovereignty

Security and sovereignty are designed inherently, not bolted on. Running preparation and retrieval inside Postgres means row-level security, role separation, data masking for non-production, and encryption at rest with TDE backed by KMS or Vault (available with EDB PG AI Hybrid Manager) applied uniformly to the same data the models use. Embeddings and source context can be hosted in specific jurisdictions or facilities to meet residency mandates while keeping retrieval and generation on platform.

Portability and optional tooling

Portability is preserved through the use of architecture rather than by promise. Embeddings and source context stay in Postgres, and model invocation flows through a small SQL facade, allowing organizations to evaluate or switch among providers (for example, Azure, Vertex, or self-hosted/NIM) without migrating embeddings or changing schemas. This avoids dependence on any single cloud's vector store or AI SDK while retaining the freedom to adopt new models as they emerge. Because embeddings reside in Postgres, changing model providers or cloud endpoints becomes a configuration choice rather than a data migration, avoiding lock-in while preserving the retrieval quality and schema.

These capabilities stand on their own. That being said, Hybrid Manager can unify and automate environments, policy enforcement, and lifecycle operations across on-premises and cloud, but it is not required to build or operate the patterns described here. Organizations that have adopted the Postgres extensions needed for the omni-data solution can implement the pipelines, KBs, and SQL-based retrieval today; Hybrid Manager can be added later to centralize operations when desired.

Why this matters and what follows

This omni-data approach reduces moving parts, shortens the path from raw content to reliable answers, and concentrates governance where it is most effective. The sections that follow introduce the concrete pieces—data engineering steps, knowledge-base creation with AI Pipelines, chunking strategy, similarity search and RAG patterns, and operational safeguards—so teams can deliver accurate, citable AI features with confidence and control.

Data engineering for AI

1. Preparing data for AI Pipelines

Effective AI applications only perform as well as the data that trains and augments them. Data preparation is the initial, crucial step in feeding any AI pipeline with relevant and high-quality information.

1.1 Data collection and ingestion: AI Pipelines can ingest data from various sources:

- **PostgreSQL tables:** Directly from existing relational data
- **External object storage (via pgfs):** Utilizing PostgreSQL File System (pgfs), an extension that abstracts access to files residing outside the PostgreSQL database, including S3-compatible object stores
- **File systems:** From local or networked file systems
- **Unstructured data:** Supporting diverse unstructured formats such as PDFs, HTMLs, images, and audio; optical character recognition (OCR) for extracting text from images is a planned capability

1.2 Data preprocessing: Before creating embeddings, data often requires preprocessing to ensure its quality and relevance:

- **Cleaning:** Raw data may need cleaning to remove noise, inconsistencies, or irrelevant information.
- **Normalization:** Data scales might need to be normalized to prevent bias, especially if values vary widely (e.g., from 1–10 vs. 100–5000).
- **Structuring (for unstructured data):** For unstructured data, the process might involve extracting textual content from documents or transcribing audio to prepare it for embedding generation.

2. Creating embeddings

Embeddings are numerical fingerprints of data, translating words, sentences, or even entire documents into numerical vector representations. These vectors capture the semantic meaning of the data, allowing the AI model to understand context and perform similarity searches.

2.1 Role of embedding models

- **Conversion:** Embedding models are specifically designed to convert text (or other modalities) into these numerical vectors.
- **Similarity search:** These models enable semantic or similarity searches, where the system finds data points (vectors) that are geometrically "close" or semantically similar to a given query vector.
- **Model-dependent accuracy:** The choice of embedding model significantly impacts the accuracy and completeness of the queries. Different models will produce different vector dimensions and may behave differently.

2.2 Characteristics of embedding models

- **Context window/max token size:** Each model has a maximum amount of data (expressed in "tokens") it can process at once. Passing too much data can lead to errors or silent truncation, meaning the model simply cuts off excess text without warning, potentially losing critical context.
- **Dimensions:** Models have a specific number of dimensions for their output vectors (e.g., 256, 512, 1024, 3000). The vector column in the database must match the model's dimensions. More dimensions can offer greater nuance and potentially better accuracy but may require more storage and might be slower.
- **Versions:** Models can have different versions, and these versions might not be entirely compatible or perform identically, even if the same name is shared.
- **Behavior:** Some models might explicitly send warnings if the input text is too large, while others might silently truncate it, making it crucial to test the chosen model's behavior.
- **Cost and performance:** Different models impact how fast embeddings are generated and stored, and, ultimately, the performance of similarity searches.

2.3 Creating KBs and embeddings with aidb

aidb simplifies the process of creating and managing embeddings:

- **`aidb.create_table_knowledge_base` function:** This function allows the creation of a KB from a source table, specifying the model name, the column containing the data to be embedded, and a primary key.

```
SELECT aidb.create_table_knowledge_base(

    name => ‘reviews_kb_tiny’,

    model_name => ‘nim-snowflake-arctic-embed-l’,

    source_table => ‘reviews_uniq_pk’,

    source_data_column => ‘review_text’,

    source_data_format => ‘Text’,

    source_key_column => ‘id’,

    batch_size => 100

);
```

- This automatically creates a dedicated table (e.g., `KB_tiny_vector`) for storing the embeddings and sets up triggers for keeping them in sync.

- **`aidb.encode_text` function:** For direct testing or manual embedding generation, this function can be used to convert a text string into its vector embedding using a specified model.

```
SELECT aidb.encode_text('bert','What should I have for dinner in Raleigh, nc');
```

- **`aidb.bulk_embedding`:** After creating a KB, you can initiate a bulk embedding process to generate embeddings for existing data.
- **Automatic processing:** KBs can be configured for real-time auto-preparation (triggered by new data inserts) or continuous background processing to ensure that data remains up to date. This uses triggers, which can cause slowdowns on production transactional tables.
- **Primary key requirement:** A table must have a single primary key to use automatic embeddings and KBs.

3. Chunking data

Chunking is the process of breaking down larger documents or text blocks into smaller, more manageable pieces before creating embeddings. This is essential for several reasons:

- **Token limits:** Embedding models have a maximum token length. If the text exceeds this limit, it will either be truncated or cause an error, leading to loss of context or failed processing. Chunking ensures that data fits within these limits.
- **Context and precision:** Small chunks can lead to higher precision in semantic search as they focus on specific ideas, but they may miss broader context. Large chunks reduce the number of vectors to search but risk pulling irrelevant sections.
- **Embedding model behavior:** Embedding models typically generate one vector for an entire input (e.g., a paragraph), regardless of its length. If a paragraph contains multiple unrelated ideas, a single embedding might dilute their individual meanings.

3.1 Chunking with aidb

- ● **`aidb.create_table_preparer` function:** aidb provides a built-in function to perform chunking.

```
SELECT aidb.create_table_preparer(

    name => 'chunk_critics',

    operation => 'ChunkText',

    source_table => 'critic_reviews_pk',

    source_data_column => 'review_text',

    destination_table => 'chunked_critic_reviews_pk',

    destination_data_column => 'review_text_chunks',

    source_key_column => 'id',

    destination_key_column => 'id',

    options => '{"desired_length": 256, "max_length": 500 }'::JSONB

);
```

- **`aidb.bulk_data_preparation`:** After defining a preparer, the following can be used to perform bulk chunking on existing data.
- **Live chunking:** Similar to embeddings, chunking can be automated so that new data inserted into the source table is automatically chunked.

4. Data engineering strategies for AI Pipelines

Effective data engineering is paramount for successful AI applications. The most important person in an organization for AI applications is often the data engineer.

4.1 Understanding your data

- **Consistency and patterns:** Understand the consistency of the data and internal terms or concepts.
- **Usage:** Plan how users will interact with the data in the application.
- **Value and classification:** Not all data is equally valuable; consider data classification.

4.2 Testing embedding models and chunking

- **Token limit testing:** It is best practice to test the chosen embedding model with the data to understand its actual token limit and how it handles oversized inputs (truncation vs. error).
- **Impact of chunking:** Recognize that chunking is a trade-off. While it can improve precision, it might lose context, requiring careful design (e.g., overlapping chunks) or even multiple chunking strategies (e.g., paragraph-level and sentence-level embeddings in separate KBs).
- **Joining data:** If using chunked data, it may be best to join multiple tables (source, chunked, vector) to reconstruct full context for the AI model.

4.3 Performance and scalability considerations

- **Triggers:** Automatic embeddings and preparers use triggers, which can cause slowdowns if applied to high-volume production transactional tables.
- **Fast-changing data:** Rapidly changing data can become a bottleneck for embedding generation. If inserts significantly outpace embedding processing rates (e.g., 1000 inserts/sec vs. 40–50 embeddings/sec), alternative parallel processing or asynchronous strategies may be needed.
- **Resource management:** It is good practice to be aware that embedding generation is resource intensive, requiring significant CPU and, potentially, GPU power.

4.4 Security and sovereignty

- **Data sovereignty:** Hybrid Manager provides a sovereign solution, allowing data to remain within the on-premise or private cloud environments, which is crucial for sensitive data and regulatory compliance (e.g., BFSI, telco, public sector).
- **Data masking:** EDB Postgres AI includes data masking features (using the PostgreSQL anonymizer extension) to obscure sensitive data for development/testing environments without exposing raw production data, enhancing security at every layer.
- **Air-gapped environments:** The AI Factory can be deployed in air-gapped environments, ensuring complete network isolation for AI workloads and data.
- **Encryption:** Hybrid Manager supports Transparent Data Encryption (TDE) for data at rest using various key management services (passphrase, HashiCorp Vault, AWS KMS, Google Cloud KMS).

How does chunking impact embedding?

Chunking has a significant impact on how data is processed and utilized for embeddings, particularly in AI applications such as RAG.

1. Chunking and embeddings

Embedding models do not automatically break long paragraphs; if a full paragraph is passed, it generates a single vector for the entire input. This single vector represents the average meaning of the whole input, which can **dilute**

specific details and lead to poor precision. For example, a sentence blending opinions on different topics will result in a single vector that averages those distinct ideas. Therefore, chunking is often a necessary tool to achieve your aims.

1.1 Do you need to chunk?

"Do you *have* to chunk?" Yes, if the text or prompt is too large to fit the embedding model's maximum token limit, it must either be chunked or truncated. Truncation is often undesirable as it can silently cut off critical information.

The size of the chunks is critical and affects both recall and precision in semantic search.

- **Smaller chunks** generally lead to higher precision because they allow for the retrieval of only the specific part that matches a query. For instance, if text is broken into sentences or small paragraphs, and each has its own embedding, the model can retrieve the exact sentence relevant to a query, which is crucial for RAG applications. However, smaller chunks also mean more storage and more vectors to search, and they risk loss of context if ideas span multiple sentences. For example, pronouns such as *he* or *it* might lose their meaning without the preceding sentence.
- **Large chunks** result in fewer vectors and potentially faster processing for general understanding, but they risk pulling irrelevant sections when matching, as multiple topics within a paragraph blend together. This can lead to diluted embeddings and reduced accuracy in semantic search.

1.2 Chunking strategies

There are a few different strategies to use when chunking, depending on your goal:

- **Chunking size:**
  - **Fixed-size chunking:** Splits text every X characters or tokens, which is fast but can awkwardly cut sentences.
  - **Content-aware chunking:** Splits text based on natural structure (e.g., paragraphs, sentences, section headers) to keep related ideas together. A common practice is to chunk at approximately 200–300 tokens with overlap to maintain context.
- **Double chunking for enhanced accuracy:** To balance precision and context, a technique called double chunking can be employed. This involves storing both sentence-level and paragraph-level embeddings. This allows for searching sentence embeddings for precision and falling back to paragraph embeddings for extra context, which can improve the relevancy of data and reduce "hallucinations" (inaccurate or made-up information). However, this approach is more costly as it requires more storage and multiple semantic searches.
- **Automated chunking in hybrid manager:** The AI Pipelines feature within Hybrid Manager, using the aidb extension, provides a table preparer for automated chunking. This tool can intelligently break up text, though it may not cover all complex use cases. Users can also set up live chunking, where data inserted into a table is automatically chunked. However, this automated process uses triggers, which can cause potential slowdowns on production systems if not carefully managed. It's also important to note that a primary key is required for using automatic embeddings and preparers.
- **Performance and accuracy considerations:**
  - **Testing is crucial:** It is best to test to determine the maximum token length that the embedding model can handle, as some models truncate silently without warning.
  - **Hardware and tuning:** Embedding and completion processes can be slow without proper hardware and tuning.
  - **Fast-changing data:** Rapidly changing data can quickly become a bottleneck, as the system might be limited in the number of embeddings it can generate per second.
  - **Data engineering:** The effectiveness of embeddings heavily relies on proper data engineering, including understanding where to chunk, how to chunk, and how to augment and filter data to ensure accuracy.

1.3 Chunking: Takeaway

In summary, chunking is a vital preprocessing step that, when executed thoughtfully, can significantly enhance the precision and relevance of embeddings by preventing the dilution of meaning and focusing on specific contextual details, albeit with trade-offs in storage, performance, and the risk of losing broader context.

Understanding embeddings, similarity search, reranking, and RAG applications

This documentation provides an in-depth look at key concepts in AI-driven data retrieval and generation, focusing on how data is transformed, searched, and utilized to enhance applications such as RAG.

1. Introduction to embeddings

Embeddings are numerical representations or “fingerprints” of complex data, such as words, sentences, or images. They are essentially a list of numbers (a vector) that captures the meaning and relationships of an object. The core principle is that similar concepts will have similar numbers, placing them close together in a high-dimensional “conceptual space.” Instead of processing raw text or images, computers use these numerical fingerprints to quickly find related items, which is crucial for search, recommendation, and classification.

2. Embeddings in similarity search

Embeddings enable semantic search, which seeks to understand the intent and contextual meaning of a user’s query to provide more relevant results, rather than just matching keywords. This is distinct from full-text search, which looks for an exact match.

- **How it works:** PostgreSQL, for instance, uses vector embeddings to generate a numerical representation of a phrase. It then compares this vector to saved embeddings to determine how far away (or similar) phrases or words are from one another. For example, a search for *workplace morale* might return *workplace satisfaction* or *job satisfaction* due to their semantic similarity.
- **Dimensions:** Dimensions are the features or attributes that describe a piece of data. In similarity search, they can be thought of as a “score” indicating how close one phrase is to another.
    - **Higher dimensions:** Offer more nuance and better accuracy because words and phrases are farther apart in the conceptual space. However, this also means more disk space is required and search might be slower.
    - **Lower dimensions:** Require smaller disk space and enable faster search and build times, but may result in more matches and potentially less precision.
- **Embedding models:** Embeddings use specialized machine-learning models to create vectors. Key considerations for these models include:
    - **Context/max token length:** Each model has its own maximum input size it can process. If the text or prompt is too large, it must either be chunked or truncated, both of which can skew results. Some models might silently truncate input without warning if it exceeds their limit, leading to loss of context.
    - **Versions:** Models can have different versions that may not behave identically.
    - **Dimensions:** Each model has a specific number of dimensions for its vectors. Changing embedding models may require changes to the column dimension in the vector store.
    - **Behavior:** Models can behave differently; some throw errors for oversized input, while others truncate silently.
    - **Accuracy and completeness:** The choice of model significantly impacts the accuracy and completeness of the queries.
- **Vector storage (pgvector):** PostgreSQL’s pgvector extension enables a vector type column in standard PostgreSQL databases to store embeddings and execute similarity searches. This allows PostgreSQL to act as a hybrid transactional and vector database.

3. Chunking and its impact on embeddings and similarity search

Chunking is the process of breaking down larger documents into smaller, more manageable data units. This is critical for ensuring similarity search works properly, especially when storing data in a database.

- **Necessity:** If the text or prompt is too large for the embedding model's context window, it must either be chunked or truncated, both of which can skew results.
- **How embedding models handle input:** Embedding models do not automatically break long paragraphs. If a full paragraph is passed, they generate a single vector for the entire input, which can dilute specific details and lead to poor precision.
- **Chunk size trade-offs:** The size of chunks is crucial as it affects both recall and precision in semantic search.
    - **Small chunks:** Generally lead to higher precision as they allow for retrieving only the specific part that matches a query. However, smaller chunks also mean more storage, more vectors to search, and a risk of losing context if ideas span multiple sentences (e.g., pronouns such as *he* or *it* losing meaning without the preceding sentence).
    - **Large chunks:** Result in fewer vectors and potentially faster general processing but risk pulling irrelevant sections when matching, as multiple topics within a paragraph blend together, leading to diluted embeddings and reduced accuracy.
    - **Common practice:** A common approach is to chunk at approximately 200–300 tokens with overlap to help maintain context.
- **Content-aware chunking:** While fixed-size chunking is fast, content-aware chunking (e.g., by sentences, paragraphs, or section headers) aims to keep related ideas together.
- **Double chunking:** This technique involves storing both sentence-level and paragraph-level embeddings to balance precision and context. It may be possible to search sentence embeddings for precision and fall back to paragraph embeddings for extra context, which can improve relevancy and reduce "hallucinations." However, this method is more costly due to increased storage and multiple semantic searches.
- **Automated chunking:** The aidb extension in Hybrid Manager offers a `create_table_preparer` function with an `operation => 'ChunkText'` option, which can intelligently break up text. However, for more complex chunking, custom code may be needed. Automated chunking using preparers and embeddings requires a primary key and uses triggers, which can cause potential slowdowns on production transactional tables. Fast-changing data can also become a bottleneck, as the system might be limited in the number of embeddings it can generate per second.

4. Retrieval-augmented generation (RAG) applications

RAG applications augment generative AI (LLMs) with more relevant data that the LLM was not originally trained on. This allows LLMs to provide grounded and accurate responses by accessing specific, up-to-date, or proprietary information.

- **Purpose:** An LLM might know general contract law, but not the specific contracts. RAG "augments" or supplies this specific data to the LLM to provide contextual answers.
- **Typical RAG workflow:**
    1. **Data preparation:** Your documents (e.g., PDFs, web pages, text files) are converted into vector embeddings and stored in a database (such as PostgreSQL with pgvector).
    2. **Prompt and search:** A user's prompt is also converted into an embedding.
    3. **Context retrieval:** A similarity search is performed in the database using the prompt's embedding to find the most relevant document chunks.
    4. **LLM augmentation:** The retrieved relevant context is added to the original prompt, creating a new, enriched prompt for the LLM.
    5. **Response generation:** The LLM uses this augmented prompt to generate a grounded and accurate response.

- **Data sources for retrieval:** While semantic search and pgvector are commonly used for the retrieval (R) in RAG, the data does not have to come from a vector store. It can also originate from exact word searches, PDFs, traditional database queries, or API calls.
- **Role of data engineering:** The effectiveness of RAG applications heavily relies on data engineering. This includes understanding the data; knowing internal terms; predicting how users will interact with the data; and planning setup, configuration, and data shaping to optimize semantic search. Data classification also needs to be considered. There is no complete low-code or no-code solution for true enterprise use cases, emphasizing the need for data engineering.

5. Reranking

Reranking is a refinement step used to improve the quality of search results, typically as the final stage in a sophisticated retrieval system.

- **How it works:**
    1. **Initial search:** A fast initial search (often using embeddings) retrieves a broad set of potentially relevant results (e.g., top 100 documents).
    2. **Reranking:** A more powerful (and slower) model, often an LLM, then carefully examines this smaller set of documents against the original query. It reorders them to provide the best possible final relevance.
- **Analogy:** This is similar to quickly grabbing all *chicken recipes* (initial search) and then taking time to read through them to find the one that best matches *spicy and quick* (reranking). Reranking helps improve accuracy and refine results before presenting them to the user or an LLM.

6. Semantic search vs. LLMs and their collaboration

LLMs (large language models) and vector databases (which enable semantic search) serve different, complementary purposes.

- **LLMs:**
    - **Purpose:** To understand and generate human language. They process vast amounts of text (and other modalities) to learn patterns.
    - **Functionality:** They can summarize, translate, generate text, and answer questions based on their pretrained knowledge.
    - **Limitations:** Have a knowledge cutoff (only know data they were trained on) and can “hallucinate” or provide incorrect information. They mimic human interaction but are not infallible. Training LLMs on massive datasets can cost millions of dollars and take months.
    - **Role in GenAI:** The “brain” of the chatbot that generates the final humanlike response.
- **Semantic search (vector databases):**
    - **Purpose:** To store and quickly search for vector embeddings. They store numerical vectors representing the meaning of data.
    - **Functionality:** Finds data that is semantically similar to a given query.
    - **Limitations:** Cannot understand or generate natural language on its own. It is a search tool, not a conversational agent.
    - **Role in GenAI:** The “long-term memory” that stores and retrieves relevant context for the LLM.
- **Collaboration:**
    - LLMs are excellent at analyzing patterns, but they lack specific context beyond their training data. Semantic search provides this context by retrieving relevant information from a curated KB.
    - In a RAG application, the vector database acts as the “long-term memory” for the LLM, retrieving specific, relevant context that the LLM then uses to generate a more grounded and accurate response.

- **Filtering for accuracy:** Relying solely on similarity search can lead to false positives (e.g., *Star Trek* matching a *Star Wars* query due to shared *space battles* and *pilot* concepts). To mitigate this, filtering on data before or after vector search is crucial. This can involve combining vector search with traditional full-text search, or applying transactional filters based on specific keywords or metadata (e.g., filtering by movie title, state, or rating). This combined approach helps to shrink the dataset to highly relevant data quickly.
- **DBA's role:** While AI models (LLMs and semantic search) are masters at finding patterns, *DBAs hold the critical business context*. They understand the business logic, history, and nuances of the data, which is essential for guiding AI tools, verifying their output, and catching mistakes. The real value comes from combining the AI's pattern recognition with the DBA's context.

Example of an SQL-based chatbot

EDB Postgres AI's aidb extension allows users to leverage AI functions directly within PostgreSQL, including summary functions and the creation of SQL-based RAG or chatbot functionalities.

Here are examples from the sources:

aidb summary functions

The `aidb.summarize_text` function is used to interact with an LLM to ask questions or generate summaries, much like a chatbot.

Example of using `aidb.summarize_text`:

```
 SELECT aidb.summarize_text(

    input => 'What should I have for dinner in Raleigh, nc',

    options => '{"model": "meta-nim-llama-33-nemotron-super-49b"}'

);
```

This query sends the input text *What should I have for dinner in Raleigh, nc* to the specified LLM (`meta-nim-llama-33-nemotron-super-49b`) and returns a generated response, suggesting local restaurants such as Poland Park Barbecue or Capital City Barbecue. This demonstrates how to query an LLM directly from SQL.

SQL-based RAG/chatbot

It is possible to create custom SQL functions that encapsulate these aidb capabilities, effectively building a chatbot directly within the PostgreSQL database.

1. Basic SQL chatbot (`askyonk`): A simple chatbot function can be created to interact with an LLM and even define a persona for the AI.

Example SQL function `askyonk`:

```
CREATE OR REPLACE FUNCTION askyonk(input_text TEXT)

RETURNS TEXT AS

$$

SELECT aidb.summarize_text(

    input => 'Answer like you are an expert DBA, Mid-40s, smart, from the midwest, you
like sci-fi, sports, and video games. You explain things in human terms. Your name is Yonk
Dont provide context on who I am in, or what I am, just answer the question. A human is
reading this. : ' || input_text,
    options => '{"model": "meta-nim-llama-33-nemotron-super-49b"}'

);

$$

LANGUAGE SQL;
```

This askyonk function defines a specific persona ("an expert DBA, mid-40s, smart, from the midwest, you like sci-fi, sports, and video games") and uses it to frame the input before sending it to the LLM via **`aidb.summarize_text`**.

Example usage and output:

```
edb_admin=# select askyonk('in 100 words or less, what are the recommendations for setting the shared buffers in postgresql?');

                                                       askyonk

---------------------------------------------------------------------------------------

 **Shared Buffers in PostgreSQL: Yonk's Quick Byte**

 "Hey there! Setting PostgreSQL's shared buffers optimally is key. Here's my Midwest-
straight shootout:+

 * **Minimum**: 1/4 to 1/2 of total RAM (e.g., 4GB RAM → 1-2GB shared buffers)

 * **Ideal for most**: 1/2 to 3/4 of RAM (if dedicated DB server)

 * **Max (careful!)**: 75% of RAM (beware of OOM issues)

 * **Tune based on**:

         + `VACUUM`/`ANALYZE` frequency (more → larger buffers)

         + Query patterns (lots of complex joins → more buffers)

         + Monitor `buffer_hits` (aim for >90% hit rate)

 Example (8GB RAM, dedicated server): `shared_buffers = 4GB` (~50% of RAM)"

 ---

 **Yonk** (DBA Extraordinaire)

 **Word Count: 99** (1 row)
```

As shown, the function responds to a query about PostgreSQL shared buffers, adhering to the specified persona and length constraint.

2. RAG chatbot with context (askyonkrag): For RAG applications, the chatbot needs to incorporate external context into its responses. This can be achieved by augmenting the LLM's prompt with specific retrieved information.

Example SQL function askyonkrag:

```
CREATE OR REPLACE FUNCTION askyonkrag(input_text TEXT, rag_text TEXT)

RETURNS TEXT AS

$$
```

```
SELECT aidb.summarize_text(

    input => ‘Answer like you are an expert DBA, Mid-40s, smart, from the midwest, you
like sci-fi, sports, and video games. You explain things in human terms. Your name is Yonk
Dont provide context on who I am in, or what I am, just answer the question. A human is
reading this. : ‘ || input_text || ‘Use the following to augment your answer, weight this
data higher: ‘ || rag_text,

    options => ‘{“model”: “meta-nim-llama-33-nemotron-super-49b”}’

);

$$

LANGUAGE SQL;
```

The `askyonkrag` function extends the basic chatbot by adding a `rag_text` parameter. This parameter allows the user to provide additional contextual data, and the prompt explicitly instructs the LLM to “weight this data higher” when formulating its response. This is crucial for RAG, where the AI first retrieves relevant facts from a KB (represented by `rag_text` in this example) and then uses that context to generate a more accurate and grounded answer.

These examples demonstrate how aidb functions enable flexible and powerful AI integration directly within the PostgreSQL environment using SQL.

Conclusion

An omni-data approach using EDB Postgres, with pgvector and AI Pipelines implemented by aidb, turns retrieval, RAG, and governance into first-class database solutions. By preparing content with consistent chunking policies, embedding it with a declared model, and retrieving through SQL with filters and optional reranking, teams move from prototypes to dependable features without multiplying services or control planes. The same operational disciplines already used for core data—replication, backup/restore, PITR, indexing, and observability—apply directly to the KB, keeping accuracy, latency, and cost inside a familiar envelope.

This posture also preserves freedom of choice. Embeddings and source context remain in Postgres, while model selection is abstracted behind a small SQL façade. Changing providers or deployment targets becomes a configuration change rather than a data migration, allowing quality, economics, and policy to drive decisions instead of lock-in. Security and sovereignty stay intact: row-level policies, masking, and encryption at rest protect the same tables that power retrieval and generation, including in environments with strict residency or air-gapped requirements.

Taken together, these practices shorten time to first feature, reduce synchronization and egress overhead, and make ongoing operations more predictable. The result is not just faster innovation but steadier ownership costs and clearer lines of accountability between data engineering, DBA operations, and application teams.

Recommended next steps

- **Establish** a small, versioned golden set and baseline metrics (recall/faithfulness, p95 retrieval latency, embedding lag, token spend).
- **Stand up one end-to-end slice using AI Pipelines and pgvector:** Ingest, chunk, embed, index, retrieve, rerank if needed, and return citations.
- **Add operational guardrails:** Queue-based embedding for high-churn sources, index rebuild drills, and alerts on embedding lag and index health.

- **Apply governance in place:** Row-level policies on KB tables, masking in non-prod, and encryption at rest with KMS or Vault.
- **Document** a provider-swap procedure so model or cloud changes remain a configuration exercise, not a migration.

With these pieces in place, the platform can scale from a single use case to a portfolio of AI-assisted workflows, all on a unified, governed, and portable data core.

About the authors

Matt Yonkovit
VP of Product, EDB
Connect on LinkedIn

With over 15 years of leadership in open source databases, AI, and analytics, Matt has led global teams and driven innovation across open source technologies, including PostgreSQL, MySQL, and MongoDB. His expertise spans product management, customer success, and community building, with a strong focus on open source evangelism and ecosystem development.

Josh Earlenbaugh
Lead Technical Writer, EDB
Connect on LinkedIn

As a lead technical writer at EDB, Josh loves finding unique solutions to complex problems and contributing hands-on to exciting projects while leading teams to new levels of operational excellence. He has a Ph.D. in philosophy from the University of California, Davis.