Core RAG Architecture with AlloyDB AI

Retrieval Augmented Generation Architecture and Implementation Example

Christoph Bussler
8 min read · Just now

AlloyDB AI provides a built-in embedding retrieval function `embedding()` . The goal of this blog is to provide a core RAG (Retrieval Augmented Generation) architecture using AlloyDB AI's functionality like `embedding()` as much as possible. Further improvements or extensions are discussed at the end of the blog.

Preliminaries

The core RAG architecture in this blog focuses on text. However, RAG in general can be based on any type of artifact like text, image, audio, or video or their combination for which an embedding generation exists.

In this blog I use AlloyDB AI's database functionality to large extent to implement a core RAG architecture. AlloyDB AI uses the PostgreSQL extension pgvector for indexing and similarity search. As such AlloyDB AI can be used as a vector database (Vector Databases (are All The Rage)).

As a side note, AlloyDB AI runs not only as a managed service in Google Cloud, but also as an installable version on a VM or laptop as well. The following was implemented on an installed version on my laptop.

Overview of core RAG architecture in two diagrams

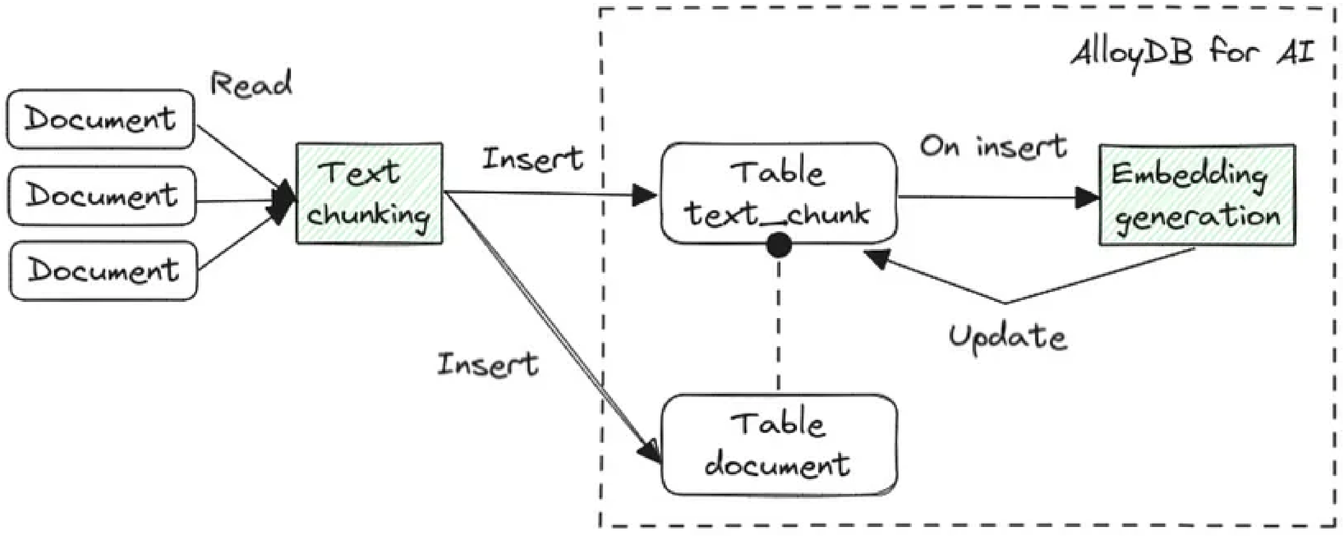
A RAG architecture can be structured along two workflows that can execute concurrently and continuously:

- **Embedding generation and storage**: Generating and storing embeddings for text chunks that are extracted from documents or retrieved from a data management system
- **Query execution**: querying embeddings from user prompts and post-processing the results through, e.g., prompt engineering

The following two diagrams show the two workflows indicating the elements of the functionality that is implemented with AlloyDB AI and the elements that that are implemented outside the AlloyDB AI database.

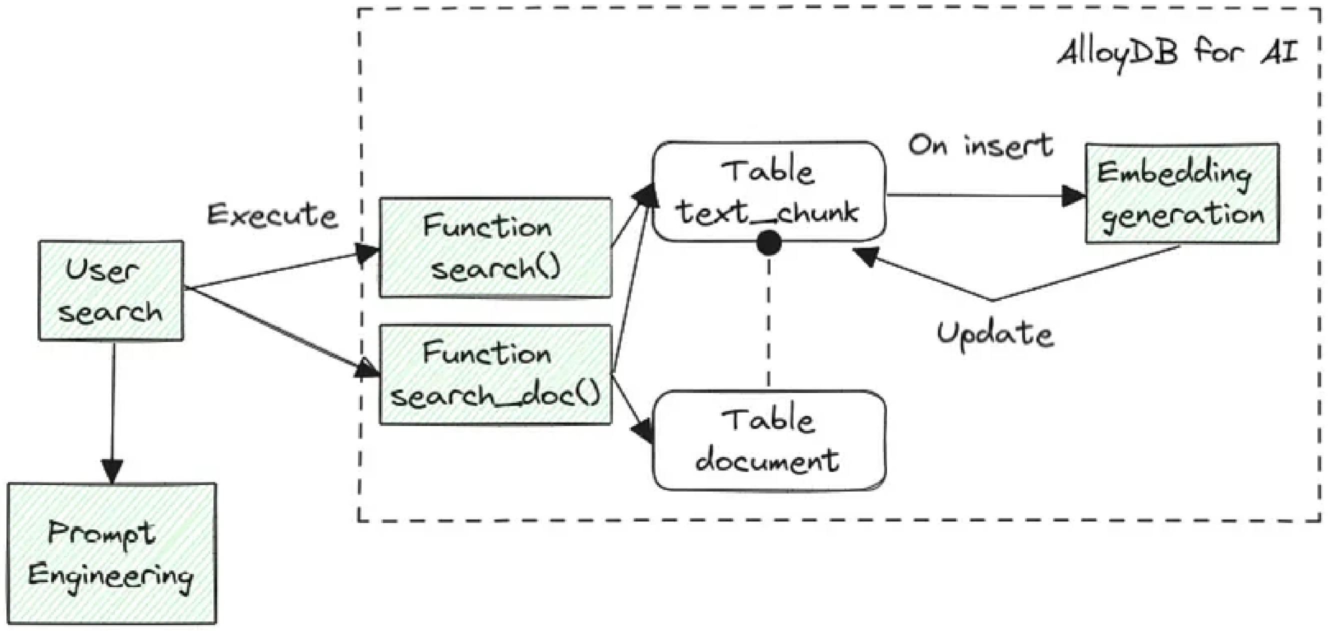
Rectangles with sharp corners represent functionality, rectangles with round corners represent data. Detailed explanations follow the diagrams.

The workflow for embedding generation and storage is (the dashed line is a foreign key relationship):



Embedding Generation and Storage

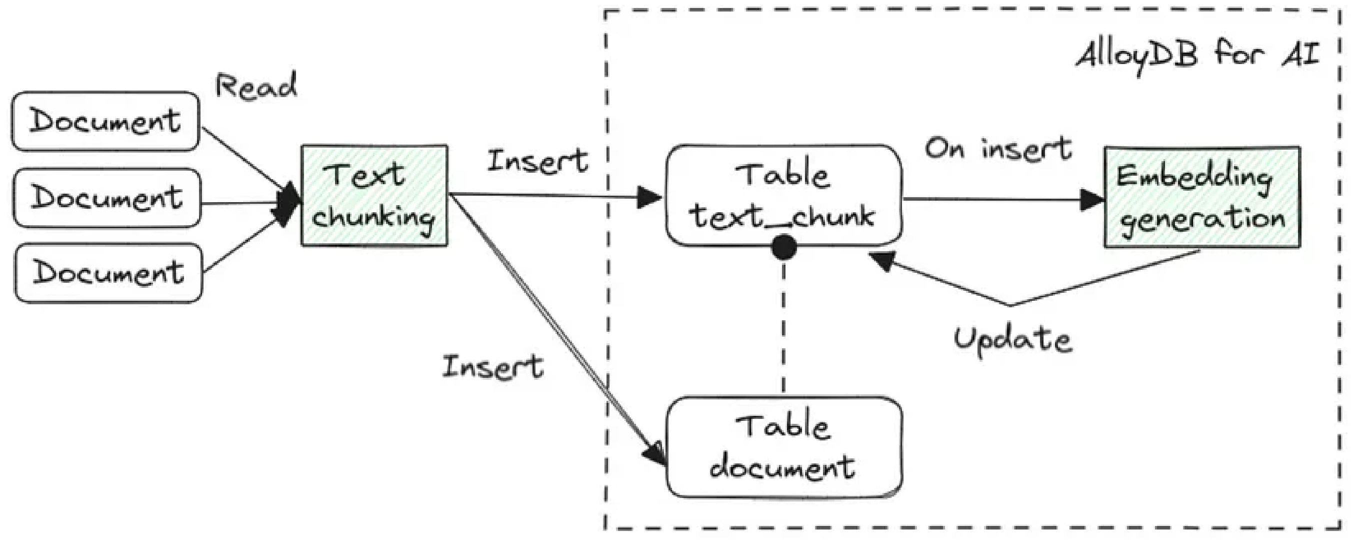
The workflow for query execution is:



Query Execution ("search")

Embedding generation and storage

This section discusses the following diagram in more detail (same diagram as above):



Embedding Generation and Storage

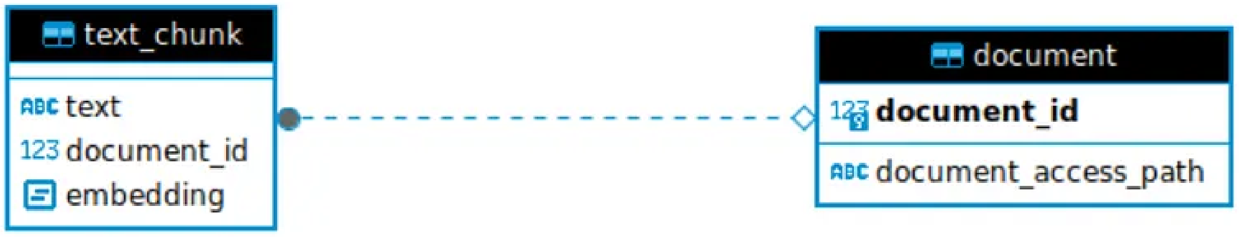
Document management and text chunking

Text chunks are extracted from documents and stored in AlloyDB AI. Document management and text chunk extraction (text chunking) are done outside AlloyDB AI and are a significant area of design decisions and implementation in itself.

Here is a discussion on text chunking and its complexity: How to Chunk Text Data — A Comparative Analysis. This video is a brief discussion on Semantic Chunking. Example discussions of text chunking are Document Sections: Better rendering of chunks for long documents, How to Optimize Text Chunking for Improved Embedding Vectorization? This query gives an impression of the importance of the topic on just one community site: https://community.openai.com/search?q=chunking.

Storing text chunks and their document relationship

The core RAG architecture provides two related tables, one for storing text chunks, and one for storing metadata of documents from which text chunks were extracted.



Database table schema

The table `text_chunk` stores the text chunks and for each a foreign key relationship to the `document` it was extracted from. The table `document` stores an identifier and an access path that is meaningful in context of the document management system to retrieve the document (if needed). For example, an access path could be the URI of Google Cloud Storage resources (https://cloud.google.com/storage/docs/gsutil#syntax).

The table are kept as simple as possible; your use case might require additional columns or tables.

The following index (index types are provided by pgvector) is created on the `text_chunk` table to improve similarity search performance:

```
CREATE INDEX embedding_index ON text_chunk
    USING ivf (embedding vector_ip_ops)
    WITH (lists = 1, quantizer = 'SQ8');
```

Alternative index types are discussed at the end of the blog.

Embedding generation

The table `text_chunk` contains an embedding corresponding to the text chunk. This embedding is inserted by an insert trigger when a text chunk is inserted by using the function `embedding()` provided by AlloyDB AI.

```
CREATE FUNCTION insert_text_chunk_create_embedding()
    RETURNS trigger
    LANGUAGE 'plpgsql'
AS
$$
BEGIN
    SELECT embedding('textembedding-gecko@001', NEW.text)
    INTO NEW.embedding;

    RETURN NEW;
END;
$$;

CREATE TRIGGER insert_text_chunk_trigger
    BEFORE INSERT
    ON text_chunk
    FOR EACH ROW
EXECUTE PROCEDURE insert_text_chunk_create_embedding();
```

This approach ensures that

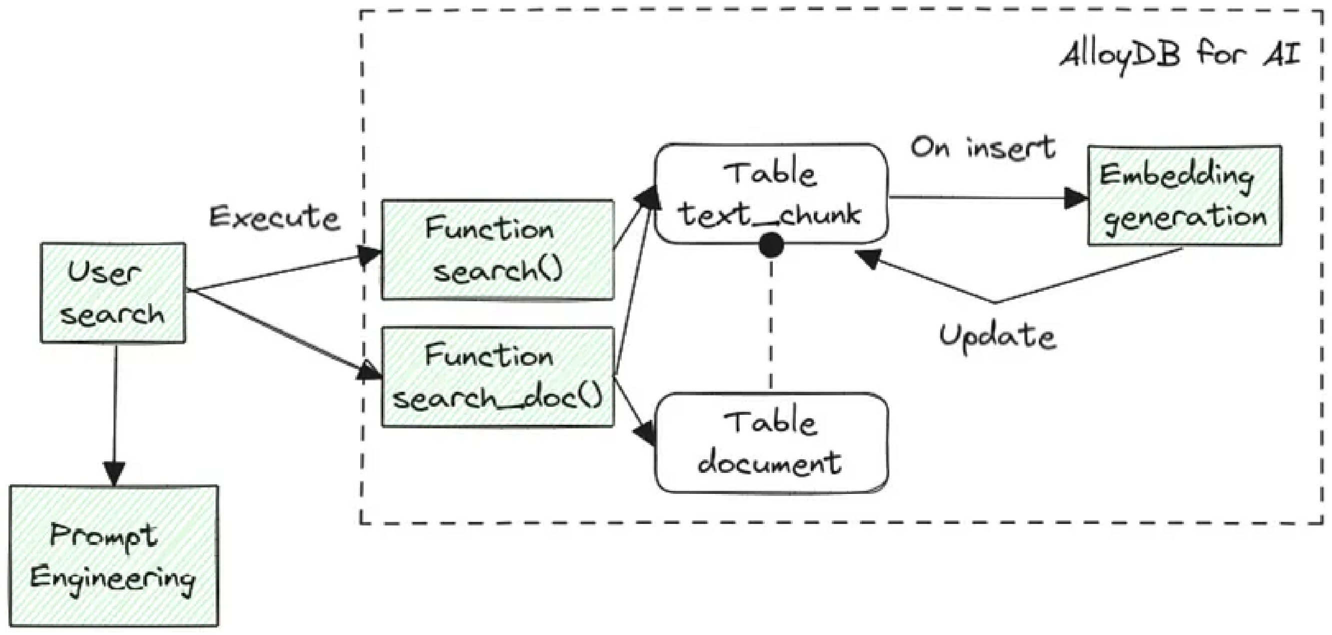
- A text chunk is related to the document it was extracted from
- Each text chunk in the system has a corresponding embedding

As documents are added to the system, text chunks are extracted from them and inserted into the core RAG architecture. This can take place "forever". Each time a text chunk is inserted, the corresponding embedding is created and inserted as well.

This continuous addition of document and text chunks can take place while in parallel query execution (next section) is performed.

Query execution (“search”)

This section discusses the following diagram in more detail (same diagram as above):



Query Execution (“search”)

Search in context of the core RAG architecture executes similarity search on the table `text_chunk`.

AlloyDB AI as a derivative of PostgreSQL supports defining functions that can be used in SQL statements. In this example, I implemented two search functions (the operator `<=>`, one of several, is provided by pgvector):

```
CREATE FUNCTION search(IN p_prompt VARCHAR, IN p_limit INT)
    RETURNS TABLE
            (
                text        TEXT,
                document_id INTEGER
            )
    LANGUAGE 'plpgsql'
AS
```

```
$$
BEGIN
    RETURN QUERY
        SELECT tc.text, tc.document_id
        FROM text_chunk tc
        ORDER BY tc.embedding <=> embedding(
                'textembedding-gecko@001', p_prompt)
        LIMIT p_limit;
END;
$$;

CREATE FUNCTION search(IN p_prompt VARCHAR)
    RETURNS TABLE
            (
                text        TEXT,
                document_id INTEGER
            )
    LANGUAGE 'plpgsql'
AS
$$
BEGIN
    RETURN QUERY
        SELECT *
        FROM search(p_prompt, 5);
END;
$$;
```

One function sets a default limit, whereas the other supports specifying the limit at invocation time. The latter helps when performing initial exploratory searches as it supports the user to get an impression of the vector space before narrowing it down.

To search, a function is invoked in a SQL statement:

```
SELECT * FROM search('How fast are GPUs executing code?')
```

Extended search returning documents

Another search function returns for each text chunk the corresponding document as well, including the document access path:

```
CREATE FUNCTION search_doc(IN p_prompt VARCHAR, IN p_limit INT)
    RETURNS TABLE
            (
                text                 TEXT,
                document_id          INTEGER,
                document_access_path VARCHAR
            )
    LANGUAGE 'plpgsql'
AS
$$
BEGIN
    RETURN QUERY
        SELECT tc.text, tc.document_id, d.document_access_path
        FROM text_chunk tc,
             document d
        WHERE tc.document_id = d.document_id
        ORDER BY tc.embedding <=> embedding(
                'textembedding-gecko@001', p_prompt)
        LIMIT p_limit;
END;
$$;
```

The document access path being returned by query execution allows a client to make the document access path available as well so that document retrieval is possible immediately upon query result availability.

The searches are similarity searches based on distance metrics. pgvector provides several of those and later alternative implementations are discussed.

Prompt engineering as extension to query execution

The diagram shows a component called “Prompt Engineering” subsequent to the user search. This is indicating that searching embeddings might not be sufficient for your use case and that you might consider post-processing of query execution results in context of LLMs to make the vector similarity search results more accessible to end users. More on the topic of prompt engineering is introduced later in the blog.

Regular pgvector parameter review

pgvector implements parameters like `lists` and `probes` (https://github.com/pgvector/pgvector#ivfflat, https://github.com/pgvector/pgvector#query-options). Review those on a regular basis to decide if the values have to be changed for an optimal system.

Core architecture design decisions

If you build a core architecture in AlloyDB AI, you have to make at least the following design decisions:

- Table specifications to store text chunks, their embeddings, and any additional data clients might need when querying the system. Above showed storing document metadata as an example.
- Index type used for embeddings that support your use case best.
- Embedding generation for text chunks. Above design uses triggers to ensure immediate storage of embeddings for each inserted text chunk.
- Similarity query implementation and which similarity metrics to use depending on your use case.

Alternative design decisions and extensions

The following is a list of topics that present alternatives or extensions to a core RAG architecture on AlloyDB AI.

Indexing and similarity search

In pgvector different types of indexes are available (https://github.com/pgvector/pgvector#indexing). Your use case dictates which index type to use.

The same is true for distance metric and similarity search (https://github.com/pgvector/pgvector#querying). Different operators are available that use a specific distance metric. Chose the appropriate one for your use case.

Prompt engineering

As indicated in the Query Execution diagram earlier, queries results obtaining embeddings can be further post-processed using prompt engineering to result in a better answer with the help of a Large Language Model (LLM).

Some resources (and a lot more exist) on prompt engineering are the following:

- Prompt engineering
- Should you Prompt, RAG, Tune, or Train? A Guide to Choose the Right Generative AI Approach
- Prompt Engineering: A Practical Example

Prompt engineering receives a lot of attention and different techniques exist to improve embedding query results.

High volume text chunk insertions

The above implementation computes the embedding for each inserted text chunk using an insert database trigger. If you have a high volume of text chunk inserts at a high frequency, you might consider changing the trigger so that it computes the embedding for batch sizes of text chunks. For example, after 1000 inserts the embedding generation takes place.

Another alternative is to decouple the inserting of text chunks from the embedding generation and initiate the latter from external to the database instead of an insert trigger. However, in this case a client system has to ensure it triggers the embedding generation.

Additional function abstractions

Instead of clients accessing tables for inserting text chunks or documents you can encapsulate the functionality in functions and make those accessible to clients.

For example, a function `insert_text_chunk()` or a function `insert_document()` could abstract the table operations properly.

A function `delete_document()` could delete the document and all corresponding text chunks.

Using alternative embedding models

AlloyDB AI provides currently a specific set of embedding models like `textembedding-gecko@001` (https://cloud.google.com/alloydb/docs/ai/work-with-embeddings#generate). If you need to use an embedding model not supported by AlloyDB AI, a client has generate embeddings and store them itself.

Full text search

PostgreSQL provides full text search functionality as well (https://www.crunchydata.com/blog/postgres-full-text-search-a-searchengine-in-a-database). For your use case it might be relevant or interesting to provide full text search in addition to the similarity search.

AlloyDB AI providing the function `embedding()` makes is possible to implement embedding related functionality (embedding generation and similarity search) within the boundaries of the database, reducing the architectural complexity of database clients in a RAG architecture

However, AlloyDB AI does not enforce the use of `embedding()` and therefore it supports alternative RAG architectures as well that rely on the embedding generation taking place outside of the database functionality.

As an architect and designer, you therefore have the option to chose architectural approaches to RAG, or even combine several approaches if your use case benefits from it.

References

Some key AlloyDB AI references are as follows. They lead you to further references as well.

- AlloyDB AI
- Work with vector embeddings
- AlloyDB AI Embedding Lab
- Building AI-powered apps on Google Cloud databases using pgvector, LLMs and LangChain