ChatGPT

RAG Indexer Build Guide

Overview

A Retrieval-Augmented Generation (RAG) Indexer is a system that ingests source documents and prepares them for efficient retrieval to augment LLM responses. The indexer's core components include: a document **chunking** module, an **embedding** generator, a **vector database** for storage, and a retrieval interface for performing **semantic search**. This guide will walk through building a RAG indexer with the following features: chunk text on logical boundaries (with token limits), use OpenAI's `text-embedding-3-small` model for embeddings, store vectors in PostgreSQL with the pgvector extension, and ensure multi-tenant data is isolated via organization IDs. We focus on a simple vector search for the MVP, while keeping the design flexible for future hybrid (keyword + vector) search and stricter multi-tenant security controls.

Document Chunking Strategy

Effective document chunking is crucial for retrieval quality. We will implement a **hybrid chunking approach** that prioritizes semantic coherence while respecting token limits. In practice, this means splitting documents along natural boundaries (sections, clauses, or paragraphs) and only further subdividing if a chunk exceeds a certain size (around 1000–1200 tokens). This strategy ensures each chunk is a self-contained unit of meaning and not just an arbitrary slice of text. If a single section is too large, we break it into smaller chunks and include a slight overlap (e.g. a sentence or two) between consecutive chunks. Overlapping helps preserve context and prevent abrupt cut-offs of important information at chunk boundaries.

This approach combines the best of both worlds: we **preserve section boundaries** to keep topics intact, and we **enforce a hard token limit** to fit within model context windows. For example, if a legal contract section or long paragraph exceeds 1200 tokens, the indexer will split it into multiple chunks, each no larger than 1200 tokens, and possibly repeat the last few sentences of one chunk at the start of the next for continuity. If smaller sections naturally fall under the limit, they remain as single chunks. By doing this, **distinct topics or sections are never merged into one chunk**, ensuring each chunk stays on a single subject [1]. At the same time, no chunk will be so large that it dilutes the embedding or exceeds the model's input capacity – any segment exceeding the max size is divided into two or more chunks [2].

This chunking strategy improves retrieval precision. Chunks that are too large and span multiple topics can produce coarse, less useful embeddings. Our method avoids that by **separating text on semantically meaningful boundaries**, which is a recommended best practice for RAG systems [3]. By keeping each chunk focused and appropriately sized, we increase the chance that a relevant chunk will closely match a query's embedding, and we reduce "noise" from unrelated content in the same chunk.

Implementation: We can implement chunking by first parsing the document into its natural sections (for example, splitting on headings or paragraph breaks). Then for each section, if it's over our token limit, split by sentence or paragraph until each chunk is within limit. A simple heuristic is to use newline or punctuation as split points. For example:

```
import textwrap

MAX_TOKENS = 1200  # our maximum chunk size (approx token count)
OVERLAP_PERCENT = 0.1  # 10% overlap for too-large chunks

def chunk_text(text: str, max_tokens: int = MAX_TOKENS) -> list[str]:
    # Simplistic token count (assuming roughly 4 chars ~ 1 token for
estimation)
    tokens_est = len(text) / 4
    if tokens_est <= max_tokens:
        return [text]  # text is already within size

    # If text is too long, split by two and overlap a bit
    halfway = len(text) // 2
    # Find a nearest sentence boundary around halfway point
    split_index = text.find('.', halfway)
    if split_index == -1:
        split_index = halfway
    part1 = text[:split_index+1]
    part2 = text[split_index+1:]
    # Include overlap: e.g., last 10% of part1 at start of part2
    overlap_size = int(len(part1) * OVERLAP_PERCENT)
    part2 = part1[-overlap_size:] + part2
    return [part1.strip(), part2.strip()]
```

Explanation: In this simple code, we estimate token length and recursively split a text roughly in half at a sentence boundary if it's over the limit. We then add a 10% overlap. In a real implementation, you might use a proper tokenizer (like GPT-3/4 tokenizers) to count tokens precisely, and a more robust method (perhaps recursive splitting by paragraphs, sentences, etc.). The key idea is that each chunk should be as large as possible **without** exceeding limits or mixing topics. After this process, we'll have a list of coherent text chunks ready for embedding.

Embedding Model Integration

For converting text chunks into vectors, we will use OpenAI's `text-embedding-3-small` model as the default. This model generates 1536-dimensional embeddings and offers an excellent balance of performance and cost for semantic search [4]. In fact, `text-embedding-3-small` is noted to be more accurate and efficient than the older `text-embedding-ada-002` model, while being cost-effective, making it a strong choice for our MVP [4]. It supports a large token input (up to ~8191 tokens) which aligns well with our chunk size limit.

By using a high-quality pre-trained embedding model, we ensure that semantically similar chunks and queries end up close in vector space. For instance, two chunks discussing "database indexing" will have embeddings with a small cosine distance between them. When a user query like "How are documents indexed for RAG retrieval?" is embedded, it should be nearest to those vectors among others.

Implementation & Swappability: We will call the OpenAI API to generate embeddings for each chunk. It's important to design this part in a modular way so that the embedding mechanism can be swapped out in the future (for example, if we want to use a local model or a different provider). To achieve this,

we can abstract the embedding logic into a function or class. For example, we might have an `Embedder` class with a method `get_embedding(text)` that encapsulates the API call. This way, all higher-level code calls `Embedder.get_embedding()`, and if we later replace OpenAI with another model, we only need to change the implementation of that class.

Using OpenAI's Python SDK, embedding a chunk is straightforward. Below is a sample of how we might integrate the embedding call:

```
import openai

openai.api_key = os.getenv("OPENAI_API_KEY")  # ensure your API key is set
EMBED_MODEL = "text-embedding-3-small"

def get_embedding(text: str) -> list[float]:
    response = openai.Embedding.create(model=EMBED_MODEL, input=text)
    # Extract the embedding vector (1536-dim list of floats)
    vector = response['data'][0]['embedding']
    return vector

# Example usage:
chunk = "This is a sample chunk of text that we want to embed."
vector = get_embedding(chunk)
print(len(vector), vector[:5])  # should print 1536 and the first five
dimensions
```

In this snippet, `openai.Embedding.create` returns an object containing the embedding. We extract the vector for the input text. The `text-embedding-3-small` model gives us a 1536-length vector for any input text. By wrapping this in `get_embedding`, we hide the specifics of the API. If in the future we want to use, say, a HuggingFace embedding model or another service, we could implement a different `get_embedding` function or class method with the same interface. The rest of the indexing code would remain unchanged, simply calling `get_embedding`. This fulfills the requirement of making the embedding backend **easily swappable/configurable**. We could even make the model name configurable (e.g., via an environment variable or config file) so that switching to `text-embedding-3-large` or a local model is a one-line change in configuration.

Vector Database Selection and Setup

For storing and searching the embeddings, we will use **PostgreSQL with the pgvector extension** as our vector database. Since our stack already includes PostgreSQL, this choice allows us to integrate vector search without introducing a new system, simplifying deployment. Pgvector enables a special column type for vectors and provides similarity search operators. This means we can store each document chunk's embedding in a Postgres table and perform efficient nearest-neighbor searches using SQL. Moreover, by choosing Postgres, we get to use traditional relational features (like filtering by org_id, metadata storage, transactions) alongside vector search, all in one database.

Using Postgres/pgvector for RAG is a proven approach: the typical workflow involves **chunking the data, embedding each chunk, and storing those vectors in a Postgres table with pgvector** [5]. Once stored, queries can be answered by performing a similarity search on this vector column. This effectively turns Postgres into a hybrid data store that can handle semantic search queries.

Table Schema: We will create a table (say, `documents`) with columns for an `id`, the `org_id` (tenant identifier), the chunk text content, and the embedding vector. For example:

```
-- Enable the pgvector extension (run once per database)
CREATE EXTENSION IF NOT EXISTS vector;

-- Create documents table for chunks
CREATE TABLE documents (
    id SERIAL PRIMARY KEY,
    org_id UUID NOT NULL,
    content TEXT NOT NULL,
    embedding vector(1536) NOT NULL  -- 1536 dim vector for text-embedding-3-
small
);
-- Optional: create an index to speed up vector searches (IVFFLAT index
requires pgvector 0.4.0+)
CREATE INDEX documents_embedding_idx
ON documents USING ivfflat (embedding vector_cosine_ops)
WITH (lists = 100);
```

In the above DDL, we specify the `embedding` as a vector of dimension 1536 (matching the model). We also create an **IVFFLAT index** with cosine distance (assuming we plan to use cosine similarity) to accelerate the nearest-neighbor search. The `lists = 100` is a parameter tuning the index for faster search vs. accuracy trade-off (100 is a reasonable default for many cases). The `org_id` will be used to scope queries for multi-tenancy (more on that later).

Abstraction for Future Vector DBs: Although we are using pgvector now, it's wise to abstract the database operations behind a repository interface. This way, if we later switch to a dedicated vector DB like Qdrant or Weaviate, the rest of the system doesn't need to change. For instance, we might create a class `VectorStorePostgres` with methods like `add_embeddings(chunks)` and `query_embeddings(query_vec)` implementing the logic in Postgres. If we want to support another backend, we create a `VectorStoreQdrant` with the same interface. At runtime, we can choose which one to use via configuration. This design ensures that adding support for a new vector DB (or a service like Pinecone) would not require a major refactor, just a new implementation of the interface.

For now, using Postgres, we can directly use SQL via a library like psycopg2 or SQLAlchemy. Here's a simplified Python example using psycopg2 for inserting embeddings and querying the nearest neighbors:

```
import psycopg2
import numpy as np

# Connect to the Postgres database
conn = psycopg2.connect(database="mydb", user="user", password="pass",
host="localhost")
cur = conn.cursor()

# Example: inserting a document chunk with its embedding
org_id = "123e4567-e89b-12d3-a456-426614174000"  # example UUID for tenant
```

```
content = "Sample chunk text about databases and indexing."
embedding_vector = get_embedding(content)  # from our embedder
# Convert embedding list to PostgreSQL array format
embedding_sql = "[" + ",".join(f"{x:.6f}" for x in embedding_vector) + "]"
cur.execute(
    "INSERT INTO documents (org_id, content, embedding) VALUES (%s, %s, %s)",
    (org_id, content, embedding_sql)
)
conn.commit()
```

In the above code, we format the Python list `embedding_vector` into a Postgres array literal (e.g., `[0.123, 0.456, ...]`) for insertion. Newer versions of psycopg2 and Postgres allow binding the vector parameter directly, but converting to an array string is a clear way to illustrate the insert. We include `org_id` so each record is tied to a tenant.

We should repeat the insert for each chunk in the document ingestion pipeline. After all chunks are inserted, the indexer has effectively built the vector index.

Indexing Workflow (Putting it Together)

With the components defined (chunking, embedding, storage), the end-to-end indexing process for a new document looks like this:

1. **Document Ingestion**: Take the raw document (e.g., a PDF text, webpage content, etc.) and identify logical sections. For example, if it's a PDF, you might already have it split by page or section; if it's a long article, you can split by headings.
2. **Chunking**: Apply the chunking strategy to each section. Split on paragraphs or sentences to ensure chunks are semantically coherent and within the token limit. This yields a list of chunk texts for the document.
3. **Embedding Generation**: For each chunk text, call the embedding model to get its vector representation (using `get_embedding`). This produces a list of embedding vectors corresponding to the chunks.
4. **Store in Vector DB**: For each chunk and its embedding, insert a record into the `documents` table (or the vector store interface). Each record includes the tenant ID (org_id), the chunk text (which can be useful later for reconstruction or debugging), and the embedding vector. This step builds up the vector index over time.
5. **Metadata (optional)**: Alongside each chunk, you can store metadata like the source document ID, section name, page number, etc. This can be helpful for reconstructing answers or filtering results, but the MVP can start with just the text and org_id.

After these steps, the document's chunks are indexed and ready to be retrieved. The indexer can repeat this process for every new document or on a schedule for document updates. Because we chose a relational database, we can also update or delete chunks easily if documents change or are removed.

Here's a brief code sketch of the indexing pipeline using the pieces we defined:

```
def index_document(doc_id: str, org_id: str, text: str):
    # Step 1 and 2: Chunk the document text
    sections = text.split("\n\n")  # naive split by blank line (as example)
```

```
    chunks = []
    for sec in sections:
        sec = sec.strip()
        if not sec:
            continue
        # If section is too large, further chunk it
        for chunk in chunk_text(sec):
            chunks.append(chunk)
    # Step 3: Generate embeddings for all chunks
    embeddings = [get_embedding(chunk) for chunk in chunks]
    # Step 4: Insert each chunk and embedding into the database
    for chunk_text_val, embed in zip(chunks, embeddings):
        embed_sql = "[" + ",".join(f"{x:.6f}" for x in embed) + "]"
        cur.execute(
            "INSERT INTO documents (org_id, content, embedding) VALUES (%s,
%s, %s)",
            (org_id, chunk_text_val, embed_sql)
        )
    conn.commit()
    print(f"Indexed document {doc_id} into {len(chunks)} chunks.")
```

In a production setting, you'd likely use a more robust chunking method (maybe using a library like *Unstructured* or *LangChain*), handle errors from the embedding API, and batch inserts for efficiency. But the above pseudo-code illustrates the flow.

Notice that the design is **tenant-aware**: we always supply `org_id` when indexing so that each chunk knows which tenant's data it belongs to. This is critical for the multi-tenancy requirements.

Querying and Retrieval

Once documents are indexed, the RAG system can retrieve relevant chunks to answer user queries. The retrieval process (for the MVP) will be purely vector-based semantic search (with potential reranking). The steps for query-time retrieval are:

1. **Embed the Query**: When a user asks a question, use the same embedding model to transform the query text into a vector (using `get_embedding(query)`).
2. **Vector Similarity Search**: Query the vector database for the closest chunk vectors to the query vector. In SQL/pgvector, this is done with the `<->` operator (which by default computes Euclidean distance or cosine distance depending on index setup). We also **filter by org_id** to ensure we only search within the user's own documents. For example, the SQL might look like:

```
SELECT id, content, embedding
FROM documents
WHERE org_id = <current_user_org>
ORDER BY embedding <-> <query_vector>
LIMIT 5;
```

This will return the top 5 most similar chunks (the smallest distance to the query vector) for that tenant's data. In Postgres, `<->` is the distance operator for vectors (if using `vector_cosine_ops`, it

gives the cosine distance). We would pass the query vector as a parameter in the SQL query similarly to how we inserted it (often by casting the array to vector type). 3. **(Optional) Rerank**: For the MVP, we might trust the vector similarity order. However, one could optionally re-rank the retrieved chunks by running a secondary scoring (for instance, using a cross-attention model or simply by some heuristic). Since the user specified **"reranked vector search only"**, we interpret that as *retrieve by vector similarity (and you might use the LLM itself to pick which chunks to use in the answer)*. We will not integrate keyword (BM25) search at this stage. 4. **Return Results**: The top-k chunk texts (and any metadata) are returned. These can then be fed into the LLM prompt (along with the original question) to generate a final answer.

Here is a code snippet demonstrating a simple query retrieval using our Postgres setup:

```
def query_documents(org_id: str, query: str, top_k: int = 5) -> list[str]:
    # Embed the user query to a vector
    query_vec = get_embedding(query)
    query_vec_sql = "[" + ",".join(f"{x:.6f}" for x in query_vec) + "]"
    # Execute similarity search in Postgres
    cur.execute(
        "SELECT content, embedding <-> %s::vector(1536) AS distance "
        "FROM documents WHERE org_id = %s "
        "ORDER BY embedding <-> %s::vector(1536) ASC LIMIT %s",
        (query_vec_sql, org_id, query_vec_sql, top_k)
    )
    results = cur.fetchall()
    # results is a list of tuples (content, distance). We extract content.
    return [row[0] for row in results]

# Example usage:
user_query = "How do we ensure data isolation in a multi-tenant RAG system?"
matched_chunks = query_documents(org_id="123e4567-e89b-12d3-
a456-426614174000", query=user_query)
for text in matched_chunks:
    print("Retrieved chunk:", text[:60], "...")
```

In this code, we parameterize the query with the query vector (cast to the appropriate vector type in SQL). The query uses `embedding <-> %s` twice – one to select the distance for output (optional) and one to order by it. The `org_id = %s` filter guarantees that we search only within the current tenant's data. The result is a list of chunk texts that are most relevant to the query. These chunks would then typically be concatenated or individually fed to the LLM as context for answering the question.

Hybrid Search Considerations (Future Enhancement)

While our initial implementation uses pure vector similarity search, it's designed such that adding a **hybrid search** component later is straightforward. Hybrid search refers to combining traditional keyword (lexical) search with vector (semantic) search to improve result relevance [6]. In a hybrid approach, a query would retrieve results using both BM25 (or another sparse retrieval method) and embedding similarity, then merge or rerank them. This can capture exact keyword matches (which vectors might miss) as well as semantic matches.

For example, OpenSearch and Elastic offer built-in hybrid search that balances BM25 and vector scores [7]. To prepare for this, we have a few considerations in our design: - **Storing Text for BM25**: We have kept the original chunk text in our index (`content` field). This means we could leverage Postgres full-text search on that field, or pipe the data into an external search engine, to get keyword-based results. Our system can then intersect or merge those with vector results. - **Modular Retrieval Interface**: By abstracting the retrieval logic, we could implement a `HybridRetriever` later. For instance, a future method `query_hybrid(org_id, query)` might internally do something like: run `tsvector` full-text search on `content` *and* vector search on `embedding`, then combine the results (maybe taking the union of top 10 from each and sorting by a weighted score). Because our current code is structured in functions that could be swapped out, we won't need to rewrite the indexing pipeline to introduce hybrid search. - **OpenSearch Option**: If requirements grow, we might choose to use an OpenSearch cluster to handle hybrid queries (since OpenSearch can natively index both text and vectors and perform combined scoring). Our architecture would allow this: we could write a new indexing routine to also push data to OpenSearch, or replace the query function to call OpenSearch's API. Since we've kept the system decoupled (not hard-coding specifics beyond our data store interface), these additions can be made without breaking the existing functionality.

In summary, the MVP focuses on **semantic vector search** for simplicity. However, we are mindful of future needs – the code structure and data we store (like raw text) will make it easy to layer in BM25 or other sparse search for a more **comprehensive hybrid search** solution down the line [7]. This will help us improve recall and precision, especially for queries where keyword matching is important.

Multi-Tenancy and Data Isolation

Our RAG indexer is built to serve multiple tenants (organizations) securely. Multi-tenancy means each tenant's data should be isolated – a user from Tenant A should never retrieve chunks from Tenant B's documents. We achieve this through **logical partitioning by** `org_id`: every chunk stored in the vector index is tagged with an `org_id`, and every query filters by the user's `org_id`.

In practice, this was implemented by adding an `org_id` column to the `documents` table and always including `WHERE org_id = <current_org>` in our SQL queries (as shown in the examples). This simple approach ensures that even though data is in a single table, tenants can only search their own vectors. It leverages application-level control – the application must be diligent in applying the filter on every query and insert.

For example, in our `query_documents` function, we make sure to pass the `org_id` and include it in the `WHERE` clause. Similarly, the indexing routine takes an `org_id` for each document and uses it when inserting records. By designing the functions to always require an `org_id` parameter, we reduce the chance of accidentally running a cross-tenant query.

Future Hardening (RLS): While the current approach is sufficient for now, it does rely on the application to enforce the tenant filter. In the future, we can adopt **PostgreSQL Row-Level Security (RLS)** to make the database itself enforce tenant isolation. With RLS, we can create a policy on the `documents` table such that each Postgres role (or user) only sees rows where `org_id` matches their tenant ID. This would add an extra layer of protection beyond the application logic, ensuring *accidental or malicious cross-tenant access is prevented at the DB level* [8]. Enabling RLS might involve setting the current tenant ID as a session variable or using a separate DB role per tenant – details to be worked out if we go down that route.

Another consideration is scaling: if one tenant's data becomes very large, we might consider physically separating data (for example, a separate schema or even a separate database per tenant) [9] [10]. PostgreSQL allows multiple strategies, such as schema-based or database-based multitenancy, each with pros and cons. For now, a single table with an `org_id` key is operationally simple and works well given our expected scale. It also keeps all tenants in one index which is easier to manage initially. As we grow, we'll monitor performance and isolation needs – the design can evolve to use **schema-level partitioning** or other methods if required, without changing the core indexing logic (we'd mostly adjust how the database stores the data, not how we generate embeddings or chunks).

In summary, we've implemented **tenant-aware filtering** throughout the indexer. Each data record is tagged and all queries scoped. This provides basic data isolation out of the box. Combined with potential database features like RLS in the future, we can confidently serve multiple tenants in one system without risk of data leakage between them [11]. Multi-tenancy is therefore a first-class concern in our build, addressed from day one.

Conclusion

By following this build guide, we set up a robust RAG indexing system that covers the essential stages: document chunking for coherent context slices, embedding those chunks with a state-of-the-art model, and storing them in a vector-searchable database. The system design respects the user's preferences for using familiar infrastructure (Postgres) and keeps future extensibility in mind.

We chunk documents on natural boundaries (with token limits) to maximize semantic relevance in retrieval [1] [2]. We leverage OpenAI's efficient embedding model to get quality vector representations of text [4]. Those vectors live in Postgres with pgvector, which simplifies our stack and allows easy querying with SQL [5]. The query flow transforms questions into the same vector space and finds matching chunks, ensuring the LLM gets the right context to answer accurately. We've also structured the solution to be multi-tenant from the ground up by partitioning data with org IDs, and we discussed how to further tighten security (e.g. with RLS) as needed [8].

While the initial implementation uses pure vector similarity search, the groundwork is laid for enhancements like hybrid search – combining lexical BM25 with semantic search for even better results in the future [7]. Thanks to the abstractions around embedding generation and data access, we can swap models or even the vector store with minimal changes to the overall system.

With this foundation, you can proceed to implement the actual code in your project. The provided code snippets illustrate how each part can be realized. From here, one would integrate the indexer into a larger RAG pipeline: after retrieval, the top chunks would be fed into a prompt for an LLM to generate final answers (often with a template prompt that cites the retrieved texts). By building the indexer as described, we ensure that step – retrieval – is reliable, fast, and easily maintainable.

Sources: The strategies and technologies used are informed by current best practices in RAG system design. Chunking on semantic boundaries is recommended to maintain context coherence [3], and using a high-performance embedding model like OpenAI's `text-embedding-3-small` provides strong semantic search capabilities out of the box [4]. PostgreSQL with pgvector has been recognized as a solid choice for RAG, allowing vector similarity search within a familiar database environment [5]. Our approach to multi-tenancy aligns with common techniques for data isolation in multi-tenant AI applications, where application-level filtering and database features like row-level security ensure each tenant's data remains private [11]. By following this guide, you'll implement a modern RAG indexer that is both **effective for today's needs and adaptable for tomorrow's improvements**.

1 2 3 Chunking for RAG: best practices | Unstructured
https://unstructured.io/blog/chunking-for-rag-best-practices

4 The guide to text-embedding-3-small | OpenAI
https://zilliz.com/ai-models/text-embedding-3-small

5 8 9 10 11 Building Multi-Tenant RAG Applications With PostgreSQL: Choosing the Right Approach | TigerData
https://www.tigerdata.com/blog/building-multi-tenant-rag-applications-with-postgresql-choosing-the-right-approach

6 7 Building effective hybrid search in OpenSearch: Techniques and best practices - OpenSearch
https://opensearch.org/blog/building-effective-hybrid-search-in-opensearch-techniques-and-best-practices/