RAG Improvements Needed

1. **Query Construction:** Does Not Implement: Text-to-Cypher (no GraphDB integration) or Self-query retriever (not automatically generating metadata filters from the query for the vector search yet).
2. **Query Translation:** HyDE, Multi-query generation, Step-back prompting, RAG-Fusion query generation.
3. **Routing:** Semantic Routing (comparing query embedding to prompt embeddings) or using the LLM itself to decide the route explicitly (though the Text-to-SQL part uses the LLM). It doesn't route to GraphDBs or generic Document stores.
4. **Indexing**: Advanced Semantic Splitter, Parent Document/Multi-representation strategies, specialized fine-tuned or token-level embeddings like ColBERT, or hierarchical methods like RAPTOR.
5. **Retrieval**: RankGPT, RAG-Fusion's reciprocal rank fusion during retrieval, CRAG's self-correction/refinement logic, or Active Retrieval mechanisms like falling back to web search if vector results are poor.
6. **Generation**: Advanced feedback loops like Self-RAG where the generation process itself critiques the retrieved documents and potentially triggers re-retrieval.
7. **Data Stores**: A separate Graph DB or a generic Document store (though PGVector stores the text chunks).
8. **LLM Capabilities Beyond Data**: The ability to answer questions using LLM knowledge, even when the answer isn't explicitly present in the dataset (i.e., not found in the DB or vector store), by leveraging the LLM’s pre-trained world knowledge.

# Breakdown:

**1. Query Construction**

* **Current:** Your system doesn't automatically generate metadata filters from the query.
* **Enhancement: Self-Query Retriever**
  + **Concept:** Enables the LLM to translate a natural language query into a structured query containing both a semantic vector search component and metadata filters. This is perfect for PGVector, which supports metadata.
  + **Implementation:**
    1. **Metadata Definition:** Define clear, filterable metadata fields during indexing (indexing.py, build\_index.py, api.py ingest endpoint). Examples: source, document\_type, creation\_date, author, keywords. Ensure these are consistently added when loading data (data\_processing.py) and indexing (indexing.py).
    2. **Attribute Info:** Create AttributeInfo objects describing your metadata fields (name, description, type) for the LLM.
    3. **Content Description:** Write a concise description of the type of content stored in your vector store.
    4. **Integration:** Use LangChain's SelfQueryRetriever.

Python

# In retrieval.py or rag\_pipeline.py

from langchain.chains.query\_constructor.base import AttributeInfo

from langchain.retrievers.self\_query.base import SelfQueryRetriever

from langchain\_google\_genai import ChatGoogleGenerativeAI # Ensure LLM is defined

# Define metadata field info (adjust based on your actual metadata)

metadata\_field\_info = [

AttributeInfo(name="source", description="The filename or URL of the document", type="string"),

AttributeInfo(name="document\_type", description="The type of document (e.g., 'report', 'email', 'article')", type="string"),

# Add other relevant metadata fields...

]

document\_content\_description = "Content chunks from various business documents and articles."

llm = ChatGoogleGenerativeAI(model=settings.llm\_model\_name, temperature=0) # Use a deterministic LLM

# Get the base PGVector vector store instance from your indexer

vector\_store = indexer\_instance.vector\_store # Assuming indexer\_instance is accessible

self\_query\_retriever = SelfQueryRetriever.from\_llm(

llm,

vector\_store,

document\_content\_description,

metadata\_field\_info,

# enable\_limit=True, # Consider enabling limits

# search\_kwargs={'k': settings.retriever\_k} # Pass underlying search args

verbose=True # Useful for debugging

)

# Replace or augment your existing retriever in EnhancedRetriever or RAGPipeline

# self.base\_retriever = self\_query\_retriever

* + 1. **Update retrieval.py:** Modify EnhancedRetriever to use SelfQueryRetriever instead of, or alongside, the basic vector\_store.as\_retriever(). Re-ranking can still be applied *after* the self-query retrieval.

**2. Query Translation**

* **Current:** Basic query passed directly to the retriever.
* **Enhancements:**
  + **HyDE (Hypothetical Document Embeddings):**
    - **Concept:** Generate a hypothetical answer to the query first, embed *that*, and use the resulting embedding for retrieval. Can improve retrieval for questions where the answer structure differs significantly from the query.
    - **Implementation:** Use HypotheticalDocumentEmbedder.

Python

# In retrieval.py or rag\_pipeline.py

from langchain.chains import LLMChain

from langchain.prompts import PromptTemplate

from langchain\_community.embeddings import HuggingFaceEmbeddings # Assuming same embeddings

from langchain\_google\_genai import ChatGoogleGenerativeAI # LLM for generation

# Define prompt for hypothetical answer generation

hyde\_prompt\_template = """Please answer the user's question concisely based on your general knowledge.

Question: {question}

Answer:"""

hyde\_prompt = PromptTemplate.from\_template(hyde\_prompt\_template)

# Use your existing LLM and embedding models

llm = ChatGoogleGenerativeAI(model=settings.llm\_model\_name, temperature=0.1)

embeddings = HuggingFaceEmbeddings(model\_name=settings.embedding\_model\_name, ...) # From indexer.py

hyde\_chain = LLMChain(llm=llm, prompt=hyde\_prompt)

hyde\_embedder = HypotheticalDocumentEmbedder(

llm\_chain=hyde\_chain,

base\_embeddings=embeddings

)

# Use hyde\_embedder when querying the vector store if needed,

# potentially as part of a custom retrieval chain or by modifying how

# embeddings are generated before the similarity search.

# Example (conceptual - direct integration depends on retriever):

# hypothetical\_embedding = hyde\_embedder.embed\_query(query\_text)

# results = vector\_store.similarity\_search\_by\_vector(hypothetical\_embedding, k=...)

* + **Multi-Query Generation:**
    - **Concept:** Generate multiple variations of the original query from different perspectives to broaden the search.
    - **Implementation:** Use MultiQueryRetriever.

Python

# In retrieval.py

from langchain.retrievers.multi\_query import MultiQueryRetriever

from langchain\_google\_genai import ChatGoogleGenerativeAI # LLM for query generation

llm = ChatGoogleGenerativeAI(model=settings.llm\_model\_name, temperature=0)

# Use your existing base retriever (e.g., from PGVector)

base\_retriever = indexer\_instance.vector\_store.as\_retriever(...)

multi\_query\_retriever = MultiQueryRetriever.from\_llm(

retriever=base\_retriever, llm=llm

)

# Replace self.base\_retriever or self.final\_retriever with this

# self.base\_retriever = multi\_query\_retriever

* + **Step-Back Prompting:**
    - **Concept:** Generate a more general, "step-back" question to retrieve broader context, alongside the original query for specific details.
    - **Implementation:** Requires a custom chain or manual implementation.
      1. Use an LLM to generate the step-back question.
      2. Retrieve documents using both the original and step-back questions.
      3. Combine and re-rank the results.
      4. Pass the combined context and original question to the final generation LLM. (Modify rag\_pipeline.py's query method).
  + **RAG-Fusion:**
    - **Concept:** Generate multiple query variations (like Multi-Query), retrieve results for each, and then re-rank the combined results using Reciprocal Rank Fusion (RRF) to prioritize documents appearing consistently across queries.
    - **Implementation:**
      1. Generate queries (use MultiQueryRetriever's generation logic or a custom LLM chain).
      2. Retrieve document lists for *each* query independently.
      3. Implement or use a library function for RRF scoring on the combined results. LangChain might have utilities, or you can implement the simple RRF algorithm.
      4. Select the top N documents based on RRF scores. (Modify retrieval.py).

**3. Routing**

* **Current:** Simple keyword/schema-based check (is\_structured\_query) to route between SQL and vector search.
* **Enhancements:**
  + **Semantic Routing:**
    - **Concept:** Embed the incoming query and compare its similarity to embeddings of predefined route descriptions (e.g., "Query about user data in tables" vs. "Query about general document contents").
    - **Implementation:** Use SemanticRouter or implement manually.

Python

# In rag\_pipeline.py

from langchain.utils.math import cosine\_similarity

from langchain\_core.prompts import PromptTemplate

# Assume embeddings instance is available (from indexer)

embeddings = indexer\_instance.embeddings

# Define route descriptions and embed them

route\_descriptions = {

"sql\_route": "Queries about specific structured data like user counts, sales figures, table schemas.",

"vector\_route": "General questions requiring information retrieval from uploaded documents, articles, or text.",

"general\_knowledge\_route": "Questions that likely require general world knowledge beyond the stored data." # Added route

}

route\_embeddings = embeddings.embed\_documents(list(route\_descriptions.values()))

route\_map = dict(zip(route\_descriptions.keys(), route\_embeddings))

def get\_semantic\_route(query: str) -> str:

query\_embedding = embeddings.embed\_query(query)

similarities = cosine\_similarity([query\_embedding], list(route\_map.values()))[0]

best\_match\_idx = similarities.argmax()

chosen\_route = list(route\_map.keys())[best\_match\_idx]

logger.info(f"Semantic routing: Query routed to '{chosen\_route}' (Similarity: {similarities[best\_match\_idx]:.4f})")

return chosen\_route

# Replace is\_structured\_query logic in RAGPipeline.query

# route = get\_semantic\_route(query\_text)

# if route == "sql\_route": ...

# elif route == "vector\_route": ...

# elif route == "general\_knowledge\_route": ... handle LLM directly ...

# else: ... fallback ...

* + **LLM-Based Routing (using Function Calling/Structured Output):**
    - **Concept:** Use the LLM itself (Gemini supports function calling) to decide which tool/route (SQL query, vector search, web search, general knowledge) is appropriate based on the query and descriptions of the available tools.
    - **Implementation:**
      1. Define "tools" or "routes" with clear descriptions (e.g., SQLDatabaseTool, VectorStoreRetrieverTool).
      2. Use a LangChain agent (like create\_structured\_chat\_agent or create\_tool\_calling\_agent with Gemini) or a chain that forces the LLM to output the name of the route to take.
      3. Modify rag\_pipeline.py to use this agent/chain for routing.

**4. Indexing**

* **Current:** RecursiveCharacterTextSplitter based on token count, basic PGVector indexing.
* **Enhancements:**
  + **Advanced Semantic Splitter:**
    - **Concept:** Split text based on semantic meaning rather than fixed chunk sizes, potentially keeping related sentences together even if it slightly exceeds the target chunk size.
    - **Implementation:** Use SemanticChunker with your embedding model.

Python

# In data\_processing.py - TextProcessor.split\_documents

from langchain\_experimental.text\_splitter import SemanticChunker

from langchain\_community.embeddings import HuggingFaceEmbeddings # Or your embedding model

# Ensure embeddings are initialized (pass from Indexer or init here)

# embeddings = HuggingFaceEmbeddings(...)

# Replace RecursiveCharacterTextSplitter

# self.text\_splitter = SemanticChunker(embeddings, breakpoint\_threshold\_type="percentile") # Or other types like "standard\_deviation"

# logger.info(f"Semantic text splitter initialized.")

# Use self.text\_splitter.split\_documents(docs) as before

# Note: Semantic chunking can be slower than fixed-size chunking.

* + **Parent Document Retriever / Multi-representation:**
    - **Concept:** Index smaller, derived chunks (e.g., summaries, hypothetical questions) that point back to larger parent documents. Retrieve based on the small chunks but return the larger parent documents for better context to the LLM.
    - **Implementation:**
      1. **Indexing (indexing.py, build\_index.py):**
         * Use ParentDocumentRetriever. This requires a docstore (like InMemoryStore, or a persistent one) to hold the original large documents and a vectorstore for the small chunks/representations.
         * Alternatively, implement manually: Create summaries or hypothetical questions for each large doc/section. Index these small pieces with metadata linking back to the original document ID/content.
      2. **Retrieval (retrieval.py):**
         * If using ParentDocumentRetriever, use it directly.
         * If manual: Retrieve the small chunks, extract the parent document IDs, and fetch the full content of those parent documents from your storage (Postgres table, file system, etc.).
  + **ColBERT / Token-level Embeddings:** More research-oriented, involving fine-grained embeddings. LangChain might have experimental integrations, but often requires dedicated vector databases or libraries (e.g., ColBERT via RAGatouille). This is a significant architectural change.
  + **RAPTOR (Recursive Abstractive Processing for Tree-Organized Retrieval):** Involves recursively clustering chunks and summarizing clusters to build a hierarchy. Requires custom implementation, likely outside standard LangChain components for now.

**5. Retrieval**

* **Current:** Basic vector similarity search, optional Cohere re-ranking.
* **Enhancements:**
  + **LLM Rerankers (RankGPT, etc.):**
    - **Concept:** Use a powerful LLM (like Gemini itself, potentially a smaller/faster model fine-tuned for ranking) to re-rank the initial set of retrieved documents based on relevance to the query. More powerful but potentially slower/costlier than embedding-based re-rankers like Cohere.
    - **Implementation:** Look for LangChain integrations like RankGPT or implement a custom chain using Gemini with a specific ranking prompt. Pass the query and document snippets to the LLM and ask it to score or reorder them. Integrate this into EnhancedRetriever.\_setup\_retriever.
  + **RAG-Fusion (Retrieval Stage):** As mentioned in Query Translation, implement Reciprocal Rank Fusion (RRF) after retrieving results from multiple query variations. retrieval.py.
  + **CRAG (Corrective-Action RAG):**
    - **Concept:** Adds a self-correction/refinement step. Retrieve documents, evaluate their relevance using an LLM component, and if relevance is low, trigger an alternative retrieval strategy (like web search) before generation.
    - **Implementation:**
      1. **Relevance Check:** After initial retrieval (retriever.retrieve\_documents), add an LLM call (in rag\_pipeline.py) to assess if the retrieved docs likely contain the answer. Design a prompt for this assessment.
      2. **Fallback:** If the assessment is negative, trigger a web search (using Google Search tool or LangChain's DuckDuckGoSearchRun, etc.).
      3. **Combine:** Combine results from the initial retrieval (if any relevant) and the web search for the final generation step.
  + **Active Retrieval:** Similar to CRAG's fallback, explicitly add logic to use Google Search or another web search tool if the vector store retrieval yields low-confidence results (e.g., low similarity scores, few documents) or if the relevance check fails. Implement in rag\_pipeline.py.

**6. Generation**

* **Current:** Generate answer based on retrieved context using a static prompt (RAG\_PROMPT).
* **Enhancement: Self-RAG:**
  + **Concept:** Makes the generation process more dynamic. The LLM generates the answer *and* reflection tokens/critiques about the retrieved documents (e.g., checking for relevance, hallucinations, supporting evidence). Based on these critiques, it might decide to proceed, refine the answer, or even trigger re-retrieval.
  + **Implementation:** This is complex and often requires custom LLM prompting and chain logic.
    1. **Modify Prompt (prompts.py):** Update RAG\_PROMPT to instruct the LLM to evaluate the provided context for relevance and accuracy *while* generating the answer. Ask it to output special tokens or structured JSON indicating its confidence or if context is missing/contradictory.
    2. **Modify Generation (generation.py):** Parse the LLM's output. If it signals issues (low relevance, missing info), implement logic to:
       - Trigger re-retrieval (perhaps with a modified query based on the critique).
       - Filter out problematic documents and regenerate.
       - Explicitly state the limitations in the final answer.
    3. **Orchestration (rag\_pipeline.py):** Adapt the main pipeline to handle potential loops or conditional steps based on the generation output.

**7. Data Stores**

* **Current:** PostgreSQL for structured data (via SQLDatabase) and PGVector for text chunks/embeddings.
* **Enhancements:**
  + **Graph Database (e.g., Neo4j):**
    - **Concept:** Store and query data based on relationships (nodes and edges). Excellent for knowledge graphs, recommendations, fraud detection.
    - **Integration:**
      1. **Setup:** Deploy a Graph DB (e.g., Neo4j).
      2. **Indexing:** Extract entities and relationships from your documents (using an LLM) and populate the Graph DB.
      3. **LangChain:** Use LangChain's Graph DB integrations (e.g., Neo4jGraph).
      4. **Routing (rag\_pipeline.py):** Add a new route for graph queries (potentially using Text-to-Cypher generation similar to Text-to-SQL). Modify the router (LLM-based or semantic) to direct appropriate queries to the Graph DB tool/chain.
  + **Generic Document Store (e.g., MongoDB, Elasticsearch):**
    - **Concept:** Store documents (potentially JSON) alongside vector embeddings or use their own text search capabilities. Useful if you need flexible document structures or mature full-text search features alongside vector search.
    - **Integration:**
      1. **Setup:** Deploy the document store.
      2. **Indexing:** Store documents/metadata. If using its search, index accordingly.
      3. **LangChain:** Use relevant LangChain loaders/integrations.
      4. **Routing (rag\_pipeline.py):** Add a route for queries best suited for the document store's capabilities (e.g., complex filtering combined with text search).

**8. LLM Capabilities Beyond Data**

* **Current:** Primarily focused on answering from retrieved context or SQL results. The default prompt asks the LLM to state if the context is insufficient.
* **Enhancement:** Allow LLM to use its pre-trained knowledge when appropriate.
  + **Implementation:**
    1. **Routing:** Introduce a specific route (e.g., general\_knowledge\_route in the semantic router example above) or allow the LLM-based router to decide when *no* specific data store is needed.
    2. **Prompting (prompts.py):** Create a separate, simpler prompt for these cases, or modify the main RAG\_PROMPT to explicitly allow using general knowledge *only* if the provided context is insufficient or irrelevant.

Diff

# In prompts.py (modified RAG\_PROMPT\_TEMPLATE)

RAG\_PROMPT\_TEMPLATE = """

CONTEXT:

{context}

QUERY:

{question}

INSTRUCTIONS:

Based primarily on the provided CONTEXT, answer the QUERY comprehensively.

Cite the relevant sources using the metadata provided (e.g., [Source: <source\_name>]).

If the context does not contain the answer or is clearly insufficient, state that the provided documents don't contain the answer, but then answer the question based on your general knowledge. Do not cite internal documents if answering from general knowledge. If you use general knowledge, mention that you are doing so.

ANSWER:

"""

* + 1. **Pipeline (rag\_pipeline.py):** Ensure the pipeline handles the "no context" or "general knowledge route" scenario by calling the AnswerGenerator with an empty context or using the modified prompt.

**General Best Practices Implementation:**

* **Architecture:** Your current separation into data\_processing, indexing, retrieval, generation, sql\_processing, api, and config is good. Continue this modular approach. Consider dependency injection frameworks (like FastAPI's Depends) more extensively for managing components like the LLM, embedding models, and database connections. Your startup/shutdown events in api.py are good for resource management.
* **Code Quality:** Add type hinting more comprehensively. Increase unit and integration tests for each component (loaders, splitters, retrievers, routers, generators). Use a linter (like Flake8 or Ruff) and formatter (like Black). Your logging setup is good (utils.py, usage across files); ensure consistent and informative log messages.
* **Performance:**
  + **Indexing:** Batch document processing and indexing (Indexer.index\_documents). Consider parallel processing for loading/splitting if dealing with large volumes (build\_index.py ). Use appropriate database indexing in PostgreSQL for metadata filtering alongside PGVector's HNSW/IVFFlat index.
  + **Retrieval:** Optimize PGVector index parameters (num\_lists, probes for IVFFlat; m, ef\_construction, ef\_search for HNSW). Cache embeddings if feasible. Use asynchronous operations (asyncio) in your FastAPI endpoints (api.py ) and potentially for I/O-bound LangChain calls if supported by the specific components.
  + **LLM Calls:** Use asynchronous LLM clients (ChatGoogleGenerativeAI.ainvoke). Batch LLM calls where possible (e.g., embedding generation if not using a dedicated embedding model). Consider smaller/faster models for intermediate LLM tasks like routing or relevance checks.
* **Security:**
  + **Secrets Management:** Continue using .env and pydantic-settings (config.py). Avoid hardcoding credentials. Use environment variables in production deployments (Docker, Kubernetes secrets).
  + **Input Validation:** Sanitize and validate all inputs (API queries (api.py ), SQL results (sql\_processing.py)). Be *extremely* cautious with executing LLM-generated SQL; the clean\_sql\_string is a basic measure, but consider allow-listing query patterns or using query parameterization if possible, though harder with LLM generation. Prevent prompt injection attacks by instructing the LLM to disregard meta-instructions within user input and clearly separating context/instructions/query in prompts (prompts.py).
  + **Database Access:** Use dedicated, least-privilege database roles for the RAG application. Ensure network security for your database (firewalls, private networks). Your docker-compose.yml exposes the DB port; in production, this should typically not be exposed publicly.
  + **Dependencies:** Regularly update dependencies (requirements.txt) and scan for vulnerabilities.
* **Scalability & Maintainability:**
  + Use asynchronous processing (asyncio, background tasks in api.py ).
  + Design components to be stateless where possible.
  + Containerize the application (docker-compose.yml) for easier deployment and scaling (e.g., using Kubernetes).
  + Implement robust monitoring (logging, metrics, tracing) using tools like Prometheus, Grafana, and LangSmith (which you have configured in config.py).
  + Keep configuration separate (config.py).
  + Maintain clear documentation for code and architecture.