1. Document Structure and Chunking Logic

The system is designed to ingest raw text documents and process them into a structured, searchable format. This process is handled by the DocumentProcessor and TextChunker components.

- **Document Structure:** The initial step involves cleaning the raw text using the DocumentProcessor. This includes normalizing whitespace, removing extraneous characters or artifacts from document conversion (like headers and footers), and structuring the text into a clean, linear format. The primary structural element we aim to preserve is the paragraph.
- **Chunking Logic:** To create contextually coherent chunks for the vector database, we employ an intelligent, semantic chunking strategy rather than a simple fixed-size split.
    1. **Paragraph Prioritization:** The text is first split by paragraph breaks. Paragraphs are treated as the primary semantic unit.
    2. **Grouping and Splitting:**
        - If a paragraph is smaller than the target chunk size, it is kept whole and may be merged with subsequent small paragraphs to form a larger, more contextually complete chunk.
        - If a paragraph is larger than the maximum chunk size, the TextChunker then splits that paragraph into individual sentences. It groups these sentences together to form chunks that are within the desired word count range (e.g., 150-250 words).
    3. **Overlap:** To ensure context is not lost at the boundaries, a sentence-based overlap is implemented. Each subsequent chunk shares the last one or two sentences of the previous chunk.

This hierarchical approach ensures that chunks are as semantically complete as possible, which is critical for providing accurate context to the language model.

2. Embedding Model and Vector Database

- **Embedding Model:** The EmbeddingGenerator is responsible for converting the text chunks into dense vector representations. For this task, we utilize the **bge-small-en-v1.5** model from the Sentence-Transformers library. This model was chosen for its excellent balance of performance and efficiency. It is highly ranked on the MTEB (Massive Text Embedding Benchmark) leaderboard for retrieval tasks and is small enough to run efficiently, even without a high-end GPU. It excels at capturing the semantic meaning of text, allowing for nuanced and accurate similarity searches.
- **Vector Database:** The project uses **FAISS (Facebook AI Similarity Search)** as its vector database, managed by the VectorStore component. FAISS is a library, not a standalone database server, which makes it ideal for projects that require a lightweight, high-performance, and local-first solution. We chose FAISS because:
    - **Speed:** It is highly optimized for fast nearest-neighbor search in large, high-dimensional vector sets.

   - **Memory Efficiency:** It offers methods for compressing vectors, allowing large indexes to be stored in RAM.
   - **Simplicity:** It integrates directly into the Python application without requiring external database management.

The embeddings of all document chunks are indexed in FAISS, allowing the SemanticRetriever to query and retrieve the most relevant chunks for a given user question in milliseconds.

3. Prompt Format and Generation Logic

The generation process is designed to ground the language model firmly in the retrieved context, minimizing hallucinations and ensuring factual accuracy.

- **Prompt Format:** The PromptTemplate component dynamically constructs a prompt for the LLM. The format is crucial for controlling the model's output. A typical prompt structure is as follows:

```
You are an expert Q&A assistant. Your task is to answer the user's question based ONLY
on the provided context. If the information to answer the question is not in the context,
state that you cannot answer. Do not use any external knowledge.

CONTEXT:
{retrieved_chunk_1}

{retrieved_chunk_2}

...

USER QUESTION:
{user_query} [/INST]
```

- **Generation Logic:** The end-to-end RAGPipeline orchestrates the following sequence:
   1. The user's query is received by the application.
   2. The query is converted into an embedding using the same bge-small-en-v1.5 model.
   3. The SemanticRetriever uses this query embedding to search the FAISS index and retrieve the top-k most similar document chunks.
   4. The retrieved chunks and the original user query are inserted into the prompt template.
   5. This complete prompt is passed to the LLMManager, which sends it to the language model.
   6. The model generates a response token-by-token, which is streamed in real-time back to the user interface for an interactive experience.

4. Example Queries with Responses

Assuming the system was loaded with a software's User Agreement.

- **Success Case (Factual Retrieval):**
    - **Query:** " Does eBay guarantee the continuous operation of the services?"
    - **Response:** "Disclaimer of Warranties; Limitation of Liability We try to keep our Services safe, secure, and functioning properly, but we cannot guarantee the continuous operation of or access to our Services. Bid update and other notification functionality in eBay's applications may not occur in real time."
- **Success Case (Summarization):**
    - **Query:** " tell me about EIS?"
    - **Response:** "One of the ways we make eBay.com listings available to international buyers is through the eBay International Shipping Program ("EIS"). EIS enables buyers to obtain parcel processing, international shipping, and customs clearance services."
- **Failure Case (Potential Hallucination / Out-of-Scope):**
    - **Query:** "What is the monthly subscription fee for a business account?"
    - **Expected "Good" Response:** "I'm sorry, but the provided user agreement does not contain information about specific subscription fees or pricing for business accounts."
    - **Highlighted Failure:** A poorly constrained model might "hallucinate" a plausible but incorrect answer, like: *"The monthly fee for a business account is $29.99."* This highlights the importance of the strict prompt instructions.

5. Notes on Hallucinations, Model Limitations, and Slow Responses

- **Hallucinations:** The system is explicitly designed to combat hallucinations by instructing the LLM to ground its answers exclusively in the retrieved text. While this significantly reduces the risk, it does not eliminate it entirely. For highly inferential or creatively worded questions, the model may still occasionally synthesize information that isn't strictly present in the source text.
- **Model Limitations:**
    - **Retrieval Dependency:** The system's performance is fundamentally limited by the quality of its retrieval step. If the SemanticRetriever fails to find the correct document chunks, the LLMManager has no chance of generating a correct answer (the "garbage in, garbage out" principle).
    - **Context Window:** The LLM has a finite context window. While we retrieve the most relevant chunks, the full answer to a complex query might be pieced together from information scattered across more chunks than can fit in a single prompt. In such cases, the answer may be incomplete.
- **Slow Responses:**
    - **Initial Load:** The first time the system is initialized, loading the embedding model and the LLM into memory can take a significant amount of time (30-60 seconds or more, depending on the hardware). This is a one-time cost, as the models are cached.
    - **Local Hardware:** System performance, particularly generation speed, is highly dependent on the underlying hardware. Running the LLM locally without a

dedicated GPU will result in noticeably slower token-by-token streaming compared to running on a CUDA-compatible GPU.

- **Query Complexity:** The time to first token is very fast, but the total generation time is proportional to the length and complexity of the required answer.