Architecture Document

System Overview

The RAG backend ingests multi-format technical documents like PDFs, DOCX and TXT, chunks them into logical sections with identification of tables and headers (context-aware chunks). Then these chunks are converted into vector embeddings representations. This RAG system supports hybrid search approach which is semantic using vector embeddings and lexical. Redis is used for caching and request coalescing.

Component architecture

Document processing – This is the pipeline to identify the document type and based on that, it uses corresponding processor to extract context aware piece of information for chunking stage. Currently there are three files types that are being handled, PDF, DOCX and TXT and this step has been implemented using python libraries like PyMuPDF, python-docx and pdfplumber. There is a provision to seamlessly add other file formats as per requirement.

Chunking – Chunking is the next step where it gets the context aware structured blocks from Document processing stage and converts them into variable sized, context aware chunks. This step is a custom implementation which merges or breaks the structured tables, text into manageable pieces of information with required overlapping.

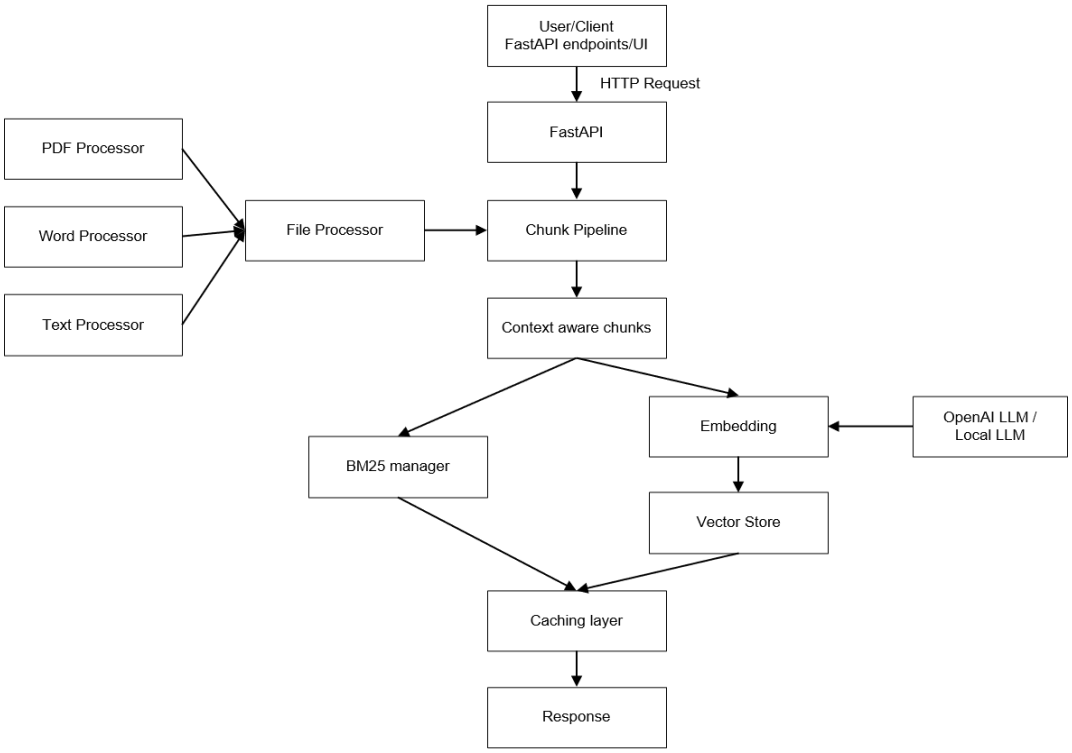
Embeddings – Embedding step is responsible for converting the chunk into vector representations. In this implementation, both OpenAI based embeddings and local setup like sentence transformers can be used to generate these embeddings.

Vector database – Vector database will be storing the embeddings created in the previous step. In this case, in order to get a better performance and controlling the type of algorithms to be used FAISS has been chosen. FAISS comes with CPU and GPU based retrieval. So, based on the system configuration like CUDA supported or GPU exists or not, the appropriate implementation can be used. Here CPU one has been used. FAISS provides better performance with respect to ChromaDB in the local setup.

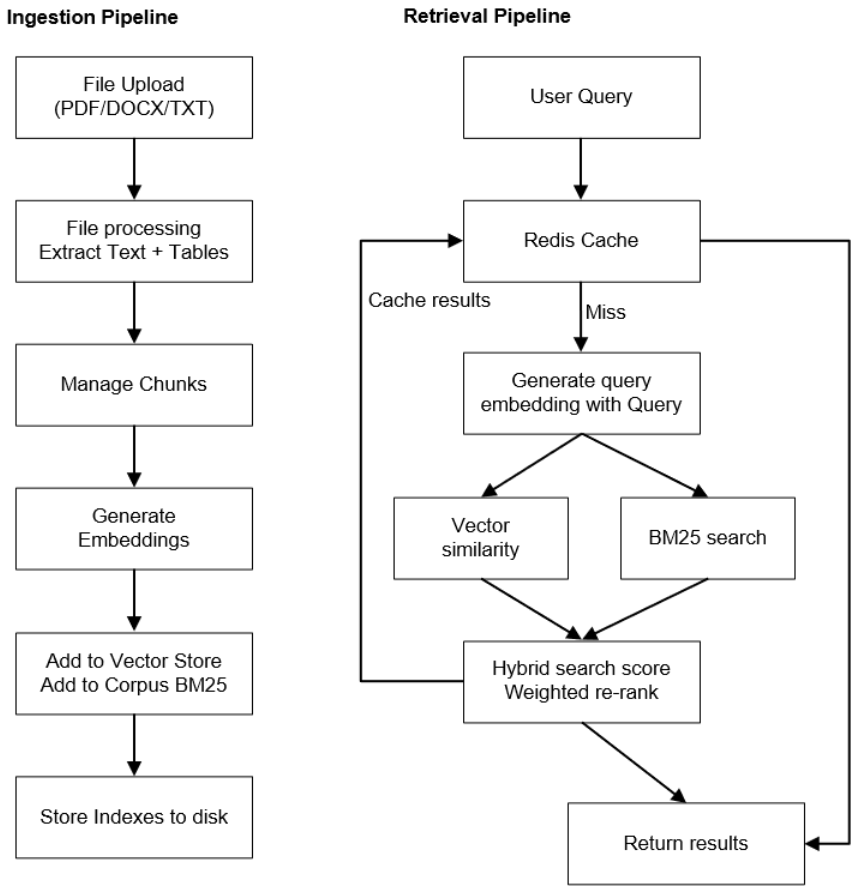
Keyword based searching (Reranking) – For Lexical Indexing in order to enable quick keyword-based query search and for using it along with vector databases for a hybrid approach. Here BM25 Implementation has been chosen for its re-ranking capabilities and better performance on this kind of scenarios when compared to TF-IDF based implementations. Using an weighted average of lexical searching and semantic searching for reranking purposes.

Caching – Redis is used for temporarily holding the query values for faster results and avoiding duplicate computation if same request comes in multiple times.

API Endpoints – In order to provide an interface to this entire RAG pipeline, a FastAPI based implementation has been provided. This provides multiple endpoints to user to upload files, perform quick or deep searches, check status etc. Later, these endpoints can be integrated into a frontend interface for a better user experience.



Architectural Diagram



Dataflow diagram

Design Decisions

Chunking Strategy – Chunking has been implemented as context aware where the primary step us to identify the headers, content and tables. In some implementations, the detection of headers and content and other structures become easy but, in some cases, there has to be a done heuristically which performs well in most of the cases. There are solutions available which can perform this activity seamlessly but it is expensive (like unstructured.io).

FAISS Vector DB – Pinecone, Weaviate, Milvus, FAISS and ChromaDB were a few options to select from. In this case, used FAISS as vector database because of performance and speed. Easy to setup locally and it provides multiple algorithms to test out – L2, IP with normalization (cosine distance), HNSW etc. In this case, IP with normalization (cosine distance) is used.

Hybrid retrieval – Combines semantic similarity with lexical similarity, both combined can provide a more precise and realistic result.

Redis – Storing in memory data (query information and output) to quickly respond to user and avoid duplicate vector store, lexical similarity operations. Implemented for query coalescing.

Scalability analysis

Data size and vector store efficiency – If using FAISS, HNSW will be able to retrieve the top k vectors based on query in logarithmic order time complexity. Also, Pinecone or open-source versions like Milvus can be used as vector store as well.

Computation of embeddings – Computation of embeddings can be performed using local sentence transformers (encoder models) through GPUs which can be scaled as per the requirements for very high throughput. OpenAI based approach can be good but it is an API based approach with throughput limits and can have high latency.

Storage for files, indexes – S3 storage can be used for storing files, indexes due to its flexible volume size and can be distributed across multiple regions for faster access.

Application scaling – Can be deployed with nginx which can improve performance of the FastAPI based implementation. If the application is containerized then it can be auto scaled using Kubernetes for load distribution.

Monitoring and observability – Certain metrices like latency during chunking, embedding and vector store retrieval can be tracked and monitoring tools like Prometheus can be used for that purpose.

Cost optimization – As in the current setup, the chunking strategy is completely python based, so for this no third-party tools are required. Deploying embedding models locally on GPUs might incur some costs but things will be quicker (external network latency won't be there). Vector store like Pinecone is not open-source alternatives hence retrievals will cost. Whereas Milvus, FAISS or Weaviate might be a better alternative.

Trade offs

Local FAISS – Local FAISS has been chosen for speed and local persistence. This is a lightweight alternative for both CPU and GPU usage and provides a good starting point to experiment and build quickly. Fast low cost but it is not production grade and distributed replication options available.

FastAPI – Used here for maintaining simplicity but might not be a good option for a very high concurrent usage load.

OpenAI embeddings – OpenAI Embeddings are good and sometimes faster if GPU is not available for local sentence transformer models. Sometimes, while running these models, it uses a lot of memory and CPU and performs way less than API based Open AI alternatives. It incurs cost but for smaller load it is a better option than running a model on cloud GPUs.

Python based file content processing – Python based context aware chunking and data processing works on heuristics especially in some cases where structural information is not available properly. Even though this approach might fail in some PDFs but works with most of the cases and better option than using third party cloud-based solutions for context aware content structure extraction.

Simple BM25 for lexical similarity – Not a good alternative for fuzzy queries but works well keyword-based query.

Local hosting – Hosting on a single machine locally or on cloud will be feasible in terms of cost and will be able to handle 100 users concurrently but not suitable for very high concurrency as this might need setting up Kubernetes cluster with load balancers.