Name : Akshay Jadhav

Retrieval-Augmented Generation (RAG) : Question Answering System

1. Introduction

This project implements a Retrieval-Augmented Generation (RAG) based Question Answering system exposed as a RESTful API. The system allows users to upload documents and later ask natural language questions whose answers are generated strictly from the uploaded content. The application is designed with clarity, modularity, and evaluation-readiness in mind, following explicit engineering decisions rather than relying on default RAG templates.

The system supports asynchronous document ingestion, semantic search using vector embeddings, controlled answer generation using a large language model, and runtime observability through metrics tracking. All components are implemented using lightweight, well-justified libraries.

2. High-Level System Overview

At a high level, the system consists of three major pipelines:

1. Document ingestion pipeline, responsible for parsing, chunking, embedding, and storing documents.
2. Question answering pipeline, responsible for retrieving relevant document chunks and generating answers.
3. Observability pipeline, responsible for tracking usage and performance metrics.

These pipelines are orchestrated through a FastAPI application that exposes well-defined endpoints and enforces request validation.

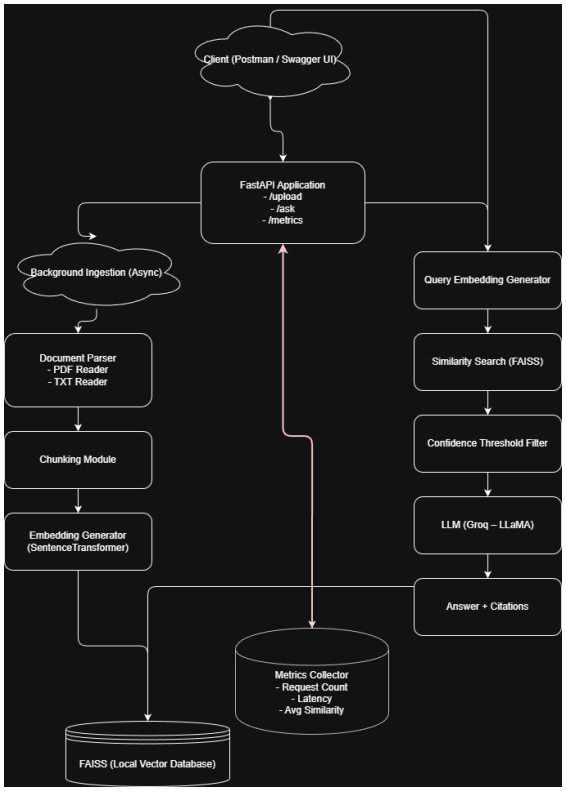


Figure 1 High Level System Design

3. API Design and Application Layer

The API layer is implemented using FastAPI due to its strong support for type validation, automatic OpenAPI documentation, and async-friendly design. The application exposes the following endpoints:

- POST /upload: Accepts document files (PDF or TXT) and triggers background ingestion.
- POST /ask: Accepts a user question and returns an answer generated from retrieved document context.
- GET /metrics: Returns aggregated system metrics such as request count and latency.

Each endpoint is strictly validated using Pydantic models to ensure predictable input and output formats.

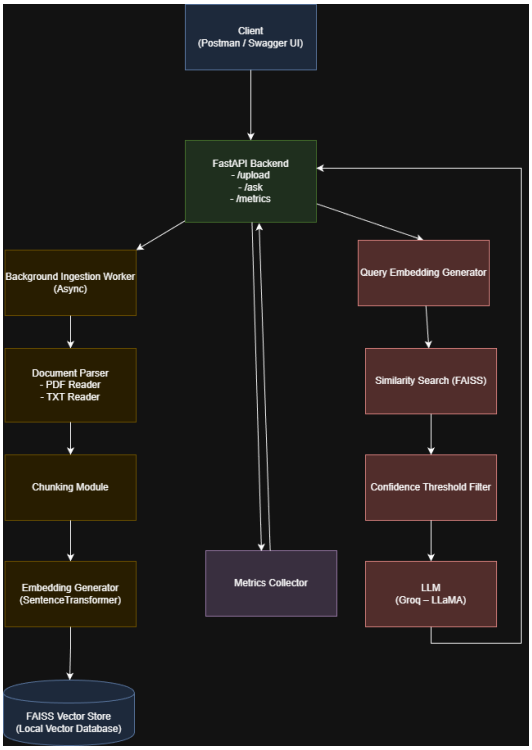
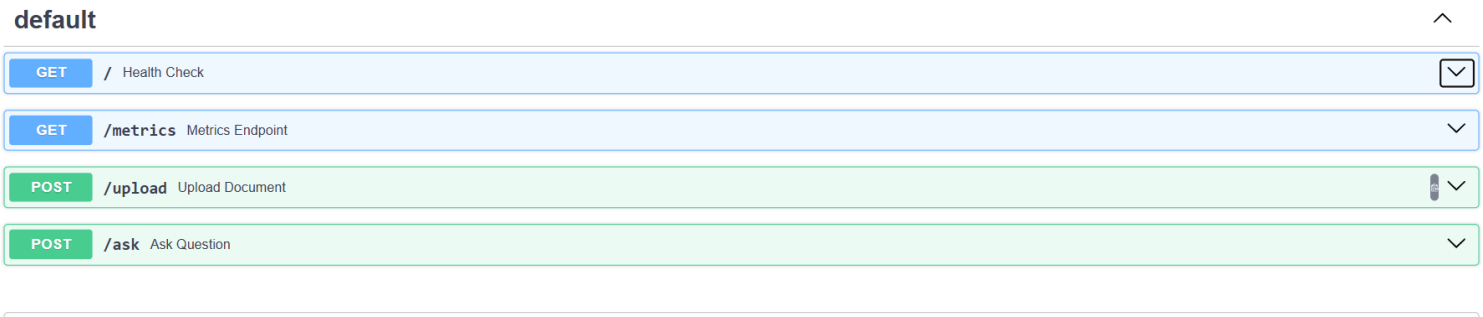


Figure 2 API Design Diagram



4. Document Ingestion Pipeline

When a document is uploaded, the API immediately returns a success response while the ingestion process runs asynchronously in the background. This design prevents large documents from blocking the request lifecycle.

The ingestion pipeline consists of the following steps:

1. Document Parsing: Uploaded files are routed to format-specific parsers. PDF files are processed using a PDF reader, while TXT files are read directly as plain text.
2. Text Chunking: The parsed text is split into fixed-size overlapping chunks. This improves retrieval granularity and ensures that semantic meaning is preserved across chunk boundaries.

3. Embedding Generation: Each chunk is converted into a dense vector embedding using a SentenceTransformer model.
4. Vector Storage: The embeddings, along with metadata such as source file name and chunk index, are stored in a local FAISS vector index.

The ingestion pipeline is fully decoupled from query-time logic, allowing documents to be indexed once and queried many times efficiently.

5. Chunking Strategy and Design Rationale

A fixed chunk size with overlap was chosen to balance semantic completeness and retrieval precision. Smaller chunks increase retrieval accuracy but may lose context, while larger chunks preserve context but reduce precision. The chosen chunk size represents a middle ground that enables effective semantic search while maintaining manageable embedding dimensions.

Overlapping chunks ensure that information spanning chunk boundaries is not lost during retrieval.

6. Vector Store and Retrieval Mechanism

FAISS is used as a local vector database to perform efficient similarity search over document embeddings. FAISS was chosen because it provides fast approximate nearest-neighbor search without requiring external infrastructure.

At query time, the user question is embedded using the same embedding model used during ingestion. The query embedding is then compared against stored embeddings to retrieve the most similar chunks.

Retrieved chunks include both text content and metadata, which is later used for citation generation.

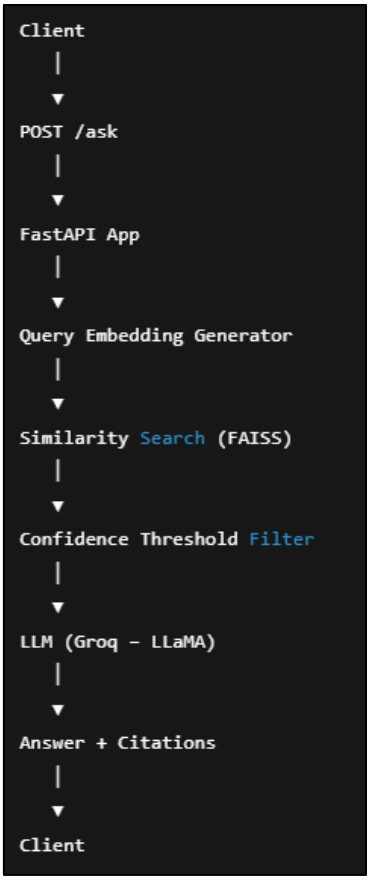
7. Question Answering Pipeline

The question answering pipeline begins when a user submits a query to the /ask endpoint. The system performs the following steps:

1. Query Embedding: The user question is converted into a vector embedding.
2. Similarity Search: FAISS retrieves the top matching document chunks based on vector similarity.
3. Confidence Threshold Filtering: Retrieved chunks below a similarity threshold are discarded to prevent irrelevant context from influencing the answer.
4. Prompt Construction: The remaining chunks are concatenated into a controlled prompt instructing the language model to answer strictly based on the provided context.
5. Answer Generation: The prompt is sent to a Groq-hosted LLaMA model, which generates a natural language answer.

6. Citation Assembly: Each answer is returned along with citations referencing the source document and chunk identifiers.

Figure 3: Question Answering Flow Diagram



8. Metrics and Observability

To demonstrate metrics awareness, the system tracks runtime statistics for question answering requests. The following metrics are collected:

- Total number of /ask requests processed
- Average response latency
- Average similarity score of retrieved chunks

These metrics are stored in memory and exposed via the GET /metrics endpoint. This allows evaluators to inspect system behavior under usage and verify performance characteristics.

Figure 4 Metrics

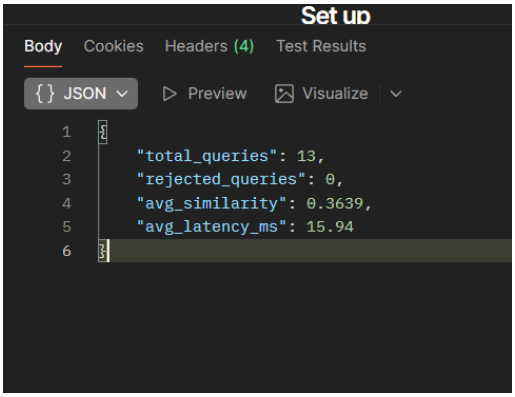
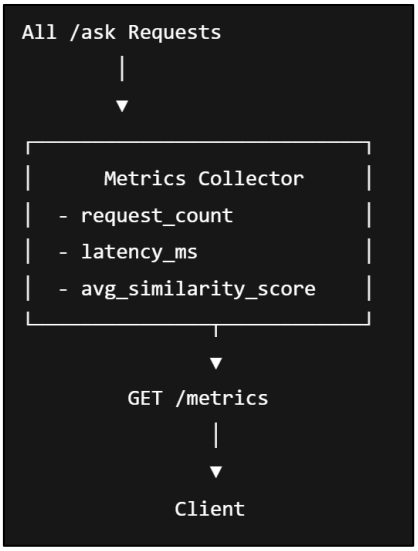


Figure 5 Metric API Design



9. Retrieval Failure Analysis

One observed failure case occurs when user questions are semantically vague or refer to information not explicitly present in the uploaded documents. In such cases, the similarity search may retrieve weakly related chunks, which are filtered out by the confidence threshold. The system then responds with "I don't know," preventing hallucinated answers.

*This **behavior is intentional** and demonstrates controlled failure handling rather than overconfident generation.*

10. Request Validation and Rate Limiting

All API inputs and outputs are validated using Pydantic schemas to enforce strict data contracts. Basic rate limiting is implemented to prevent excessive query traffic from overwhelming the system.

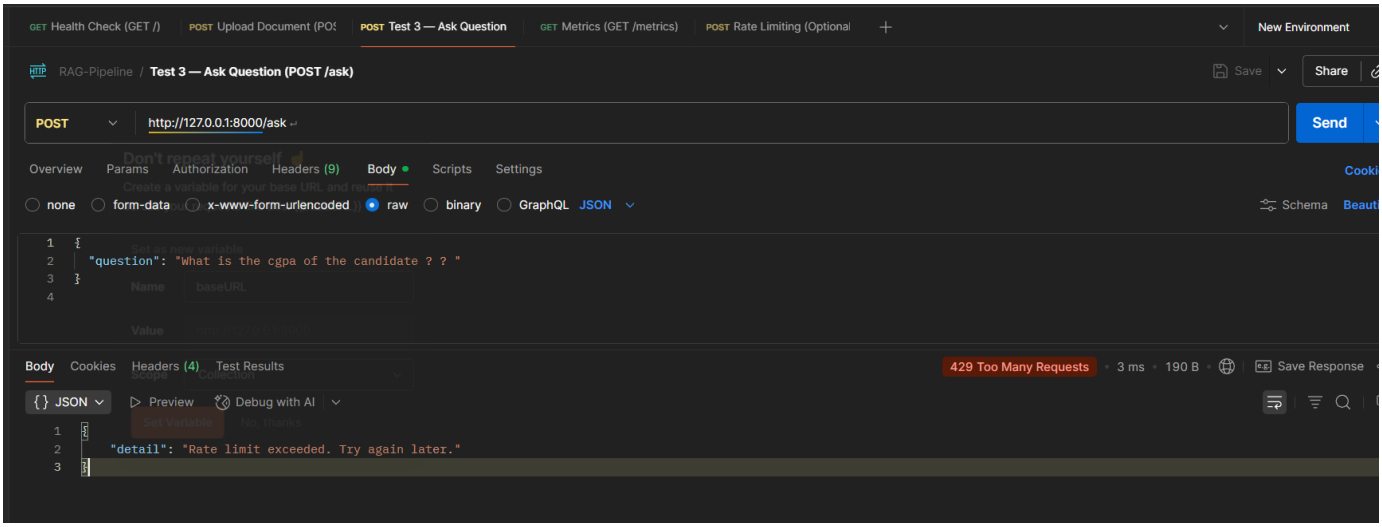


Figure 6 Rate-limit exceeded after 10+ API calls

11. Design Constraints and Justifications

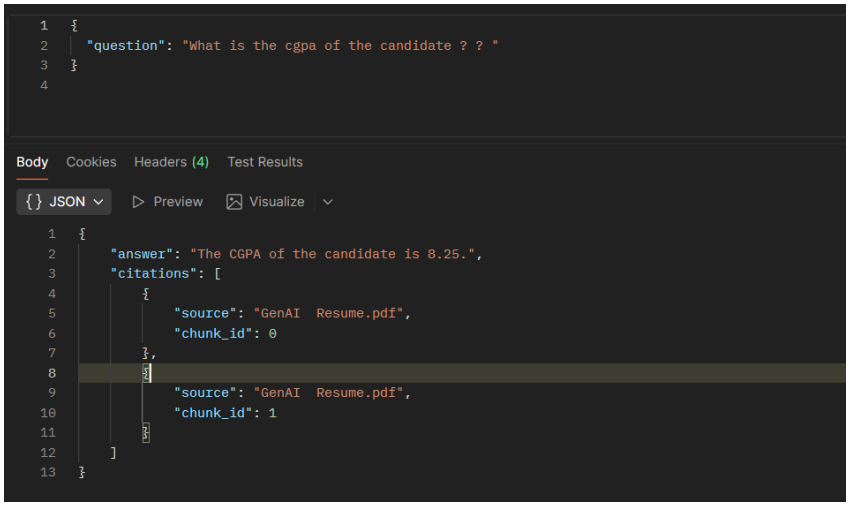
The system avoids heavy frameworks and cloud dependencies to remain lightweight and transparent. All design decisions, including chunk size, embedding model choice, and local vector storage, are explicitly justified to demonstrate understanding rather than reliance on defaults.

12. Conclusion

This project delivers a complete, modular, and evaluation-ready RAG-based Question Answering system. It satisfies all functional and technical requirements while providing clear explanations of design decisions, failure cases, and performance metrics. The system is suitable for extension into production-scale deployments while remaining simple enough for academic and technical evaluation.

RESULT

- Correct with-in Context Question



- Out of Context/Domain Question

- Metrics