# Project – High Level Design On Chat with Technology Documents

Course Name: Generative AI

***Institution Name:*** Medicaps University – Datagami Skill Based Course

*Student Name(s) & Enrolment Number(s):*

| S. No. | Student Name | Enrollment Number |
|---|---|---|
| 1. | Aayush Pal | EN22CS301022 |
| 2. | Aayushee Dhote | EN22CS301024 |
| 3. | Aditi Shrestha | EN22CS301044 |
| 4. | Aditya Chouhan | EN22CS301049 |
| 5. | Aditya Singh Chouhan | EN22CS301066 |

*Group Name: Group 09D1*

*Project Number: GAI-09*

*Industry Mentor Name: Aashruti Shah*

*University Mentor Name: Prof. Ajaj Khan*

*Academic Year: 2025 - 26*

# Table of Contents

## 1. Introduction

The Chat with Technology Documents project is an AI-powered, web-based application designed to enable users to interact with technical documents using natural language queries. Users can upload documents in PDF or DOCX format and receive accurate, context-aware responses generated directly from the content of the uploaded document.

The system is built using a Retrieval-Augmented Generation (RAG) architecture. It integrates OpenAI embedding models for semantic vector generation, FAISS (Facebook AI Similarity Search) for high-performance vector retrieval, and ollama as the Large Language Model (LLM) for answer generation.

This architecture ensures that all generated responses are strictly grounded in the uploaded document content rather than relying on general internet knowledge. As a result, the system provides reliable and domain-specific answers suitable for technical documentation, academic material, and structured knowledge sources.

### 1.1 Scope of the Document

This High-Level Design (HLD) document presents a comprehensive architectural and technical overview of the system. It includes:

- Overall system architecture and design principles
- Application layer design (Frontend, Backend, AI, and Data layers)
- Process flow and information flow
- Component-level technical design
- API specifications
- Database schema and data model
- Security considerations
- Performance and non-functional requirements

This document serves as a formal technical reference for evaluation, implementation, and future system enhancements.

### 1.2 Intended Audience

This document is intended for:

- **Students** – To understand the system design, workflow, and implementation approach.
- **Faculty Members & Academic Mentors** – To evaluate the architecture, technical depth, and academic relevance of the project.

- **Industry Mentor** – To review practical implementation and real-world applicability.
- **Development Team** – To understand system components and integration flow.
- **System Architects & Project Evaluators** – To assess design quality, scalability, and overall project structure.

### 1.3 System Overview

The system enables users to:

- Upload technical documents
- Automatically extract and semantically index document content
- Ask natural language questions
- Receive precise, document-grounded responses in under five seconds

The architecture consists of four primary layers:

**User Interface Layer** – Web-based frontend built using HTML, CSS, and JavaScript for PDF upload and chat interaction.

**Backend API Layer** – FastAPI-powered REST services for file handling, RAG orchestration, and query processing.

**AI Processing Layer** – LangChain-based Retrieval-Augmented Generation (RAG) pipeline integrating:

- PDF text extraction (PyMuPDF)
- Recursive text chunking
- Local embedding generation via Ollama
- Chroma vector similarity search
- Llama3 (via Ollama) for response generation

**Data Storage Layer** – Local storage consisting of:

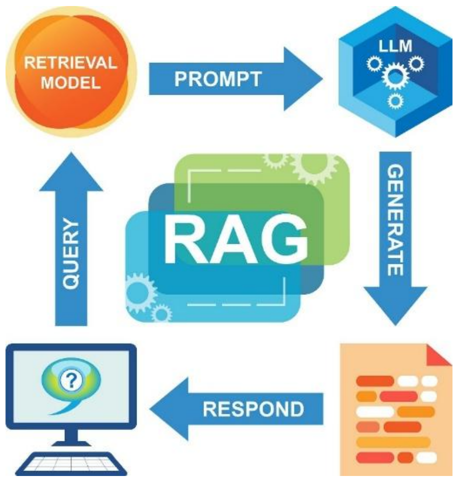
- Chroma Vector Database for embedding storage and similarity retrieval
- Local file system for uploaded PDF documents.

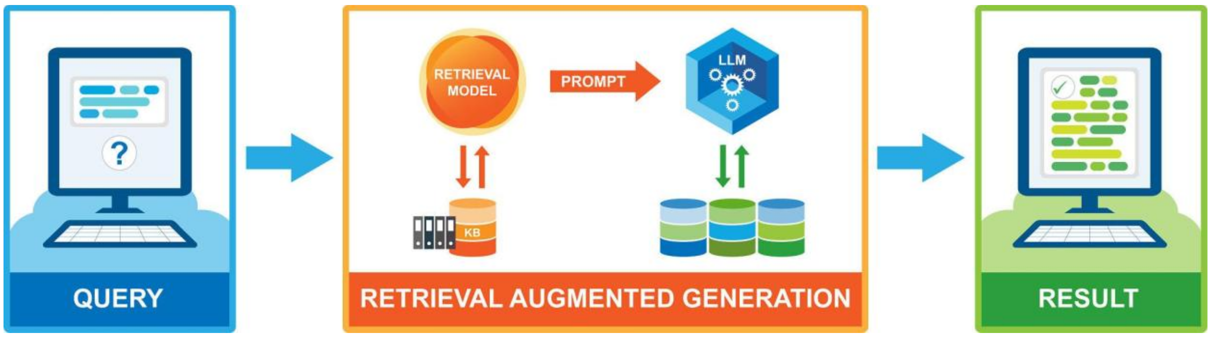
## 2. System Design

### 2.1 Overall Architecture

The system follows a client-server architecture enhanced with a dedicated AI Processing Layer.

The frontend communicates exclusively with the backend via REST APIs over HTTPS. The backend orchestrates document ingestion, embedding generation, vector indexing, similarity search, and LLM-based response generation.





**Architectural Layers**

**User Layer**

- Web Browser / Desktop Client
- Built using HTML, CSS, JavaScript

**Backend Layer (FastAPI)**

- REST endpoints: /upload, /process, /chat, /history
- Manages authentication, validation, orchestration

**AI Processing Layer**

Built using:

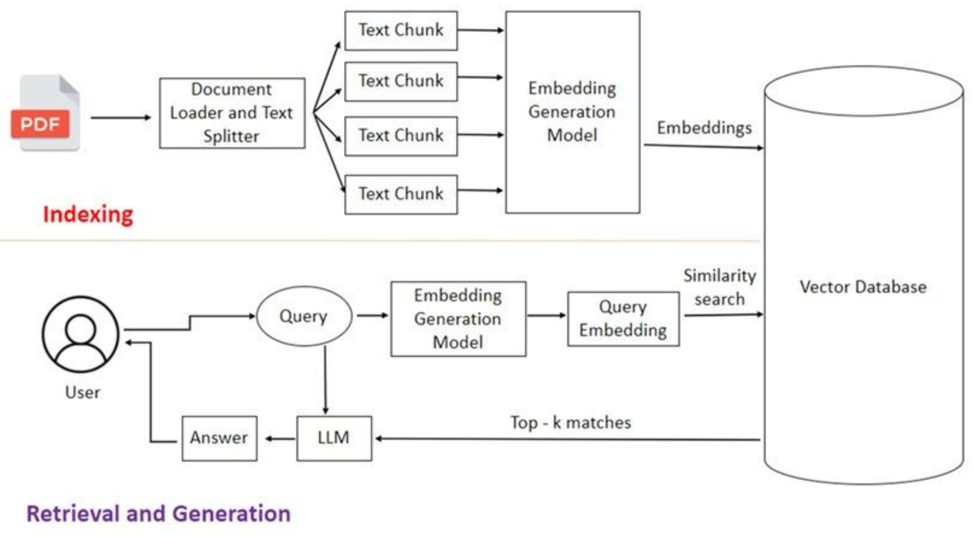
- LangChain
- Ollama
- Llama3
- Chroma Vector Database
- Components:
- PDF Loader (PyMuPDF)
- Text Chunking (RecursiveCharacterTextSplitter)
- Embedding Generation (Ollama embeddings)
- Chroma Vector Storage
- Similarity Retrieval
- Llama3 Response Generation

**Data Storage Layer**

- PostgreSQL / SQLite (Relational metadata storage)
- FAISS Index (Vector storage)
- Redis Cache (Optional performance enhancement)

The modular design allows components to be replaced independently (e.g., FAISS with Pinecone or GPT-4 with a self-hosted LLM) without affecting other layers.

## Architecture Diagram



## 2.2 Application Design

### Frontend Design

The frontend consists of three primary interfaces:

- **Document Upload Interface** – Drag-and-drop file upload with validation and progress feedback
- **Chat Interface** – Conversational UI with streaming AI responses and optional source highlighting
- **Document Management Panel** – Displays uploaded documents, processing status, and deletion options

Technologies used:

- HTML
- CSS
- JavaScript

The frontend communicates with the backend exclusively through secure REST APIs.

### Backend Design

The backend is implemented using FastAPI, a high-performance asynchronous Python web framework.

Key responsibilities include:

- File validation (type, size, MIME verification)
- Secure file handling and metadata persistence
- Orchestration of text extraction and chunking
- Embedding generation via OpenAI API
- FAISS index creation and similarity search
- Prompt construction and LLM invocation
- Session and chat history management

FastAPI provides:

- Asynchronous request handling
- Automatic OpenAPI documentation
- Built-in data validation using Pydantic

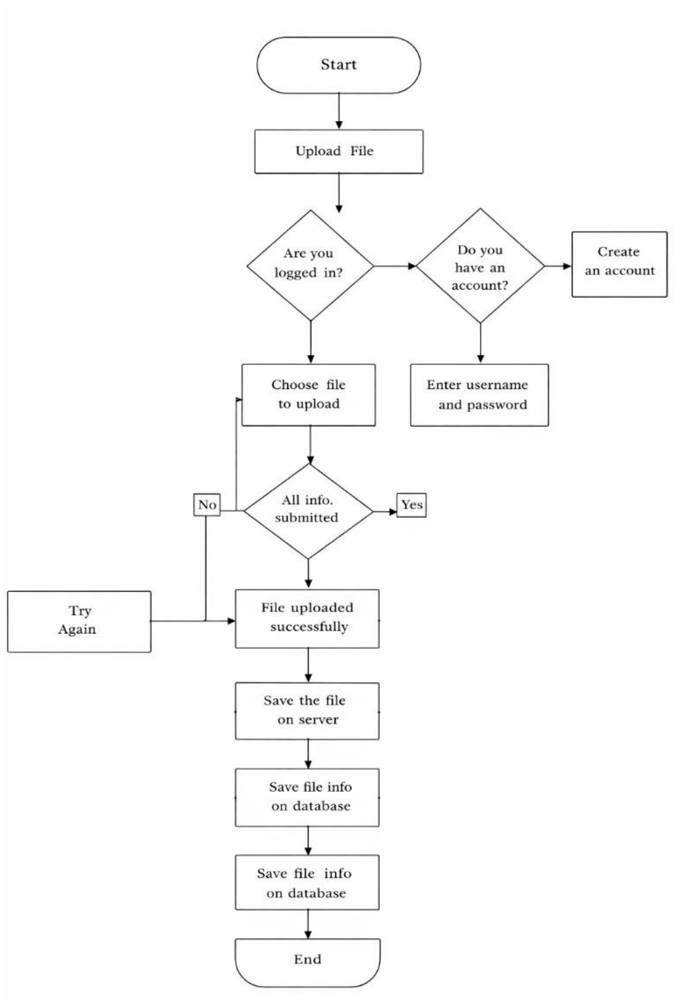
## 2.3 Process Flow

The system includes two major workflows:

**Document Ingestion Pipeline**

1. User uploads a PDF/DOCX file via frontend
2. Backend validates file type and size
3. Text is extracted using PyMuPDF or python-docx
4. Text is split into 512-token chunks with 50-token overlap
5. Each chunk is converted into a vector embedding
6. Embeddings are stored in FAISS index

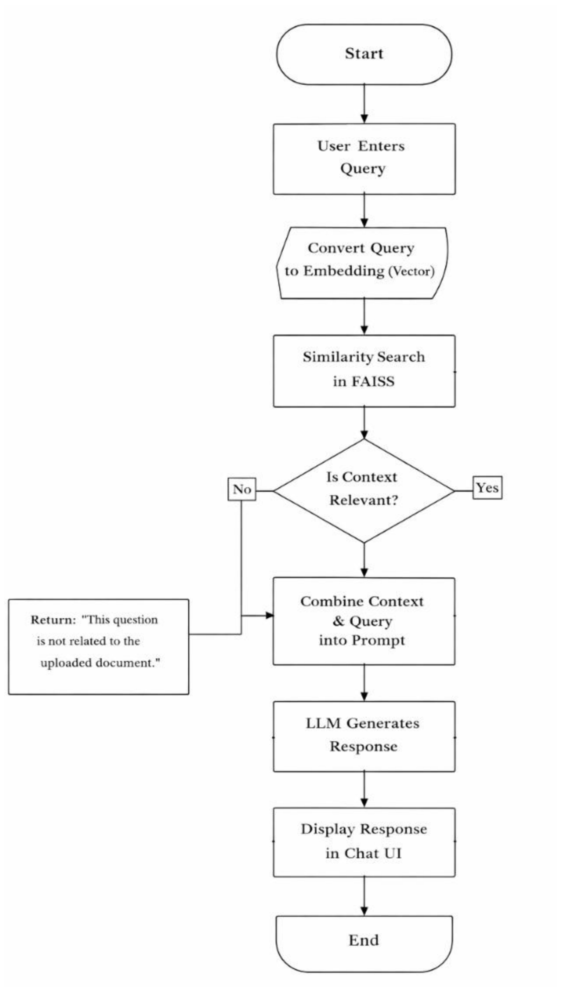
This process is executed once per uploaded document.



**Query–Answer Pipeline**

1. User submits a natural language query
2. Query is converted into an embedding vector
3. Top-K relevant chunks are retrieved using cosine similarity
4. Retrieved context is combined with user query
5. GPT-4 / GPT-3.5 generates a context-aware response
6. Response is returned and displayed in chat interface

This process executes for every user query.


Start
User Enters Query
Convert Query to Embedding (Vector)
Similarity Search in FAISS
Is Context Relevant?
No
Yes
Return: "This question is not related to the uploaded document."
Combine Context & Query into Prompt
LLM Generates Response
Display Response in Chat UI
End

## 2.4 Information Flow

**Information Flow Sequence:**

User → Frontend → Backend API → Document Processor → Vector Database → LLM → Backend → Frontend → User

This ensures controlled and secure data movement within the system.

## 2.5 Components Design

The system consists of five major modules:

**1. Document Upload Module**

- Accepts only PDF and DOCX formats
- Maximum file size: 20MB
- Temporary storage during processing
- Metadata stored in SQL database

**2. Text Extraction Module**

- PyMuPDF for PDF parsing
- python-docx for DOCX parsing
- Cleans formatting artifacts
- Preserves semantic paragraph structure

**3. Chunking Module**

- 512-token fixed-size chunks
- 50-token overlap
- Configurable strategy

**4. Embedding Module**

- OpenAI text-embedding model (1536-dimensional vectors)
- Handles API rate limiting
- Stores embeddings in FAISS and optionally database

**5. FAISS Vector Database**

- IndexFlatIP for cosine similarity
- Top-K retrieval (default K=5)

- Serialized for persistence

## 2.6 Key Design Considerations

- Scalability for multiple concurrent users
- Secure handling of uploaded documents
- Low latency response time
- Efficient vector storage
- Context window optimization
- Prompt engineering for accurate responses

## 2.7 API Catalogue

All APIs follow RESTful architecture standards.

All endpoints require JWT Bearer Token authentication.

| **Method** | **Endpoint** | **Description** | **Request Type** | **Response** |
|---|---|---|---|---|
| POST | /api/v1/upload | Upload PDF or DOCX file for processing | multipart/form-data | { document_id, status, message } |
| POST | /api/v1/process | Extract text, chunk, and generate embeddings | application/json | { document_id, chunk_count, status } |
| POST | /api/v1/chat | Submit a natural language query | application/json | { response, sources, session_id } |
| GET | /api/v1/history/{session_id} | Retrieve full chat history for a session | — | { session_id, messages[] } |
| DELETE | /api/v1/document/{doc_id} | Remove document and associated embeddings | — | { status, message } |

## 3. Data Design

### 3.1 Data Model

It uses:

- Chroma vector store
- Local file system for PDFs

### 3.2 Data Access Mechanism

- REST APIs serve as the primary data access interface for the frontend.SQL-based database access
- FAISS Python library is used directly for vector index read/write operations.
- Vector similarity search queries

### 3.3 Data Retention Policies

- Uploaded documents are stored only for a defined duration and then automatically deleted.
- Associated chunk text and embeddings are deleted when the parent document is removed.
- Chat history securely maintained.
- Compliance with data privacy standards.

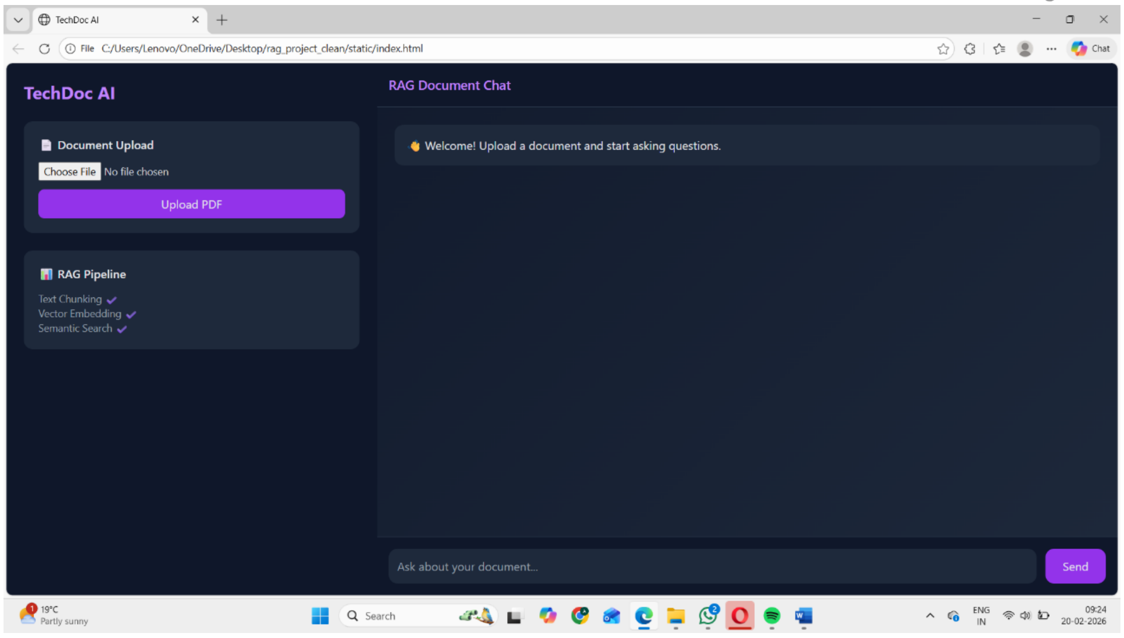
### 3.4 Data Migration

- Database backup procedures
- Schema version control
- Re-indexing of vector database when required

## 4. Interfaces

### 4.1 User Interface

- Chat screen with real-time streaming responses
- File upload screen with drag-and-drop and progress indicator
- Document management panel displaying upload history and processing status
- Fully responsive design supporting desktop and mobile browsers



Welcome! Upload a document and start asking questions.

## 4.2 External Interfaces

- OpenAI Embeddings API — for generating chunk and query vectors
- OpenAI Chat Completions API — for response generation
- FAISS Library — for vector indexing and similarity search

## 5. State and Session Management

- Token-based authentication
- Session ID per user
- Context maintained per document session

## 6. Caching

- Redis cache stores embeddings of frequently asked queries (TTL: 1 hour)
- FAISS search results cached within session scope
- Document metadata cached in memory at backend startup
- Cache invalidated upon document deletion or system restart

## 7. Non-Functional Requirements

| Category | Requirement |
|---|---|
| Performance | Response time under 5 seconds |
| Scalability | Supports multiple users simultaneously |
| Availability | High system uptime |
| Security | Encrypted communication |
| Usability | Simple and intuitive interface |
| Maintainability | Modular architecture with replaceable components |

### 7.1 Security Aspects

- All communication over HTTPS/TLS 1.3
- JWT authentication with 1-hour expiry and refresh token support
- File type validation via MIME verification
- Prompt injection protection through input sanitization
- API keys stored securely in environment variables
- Input length restrictions to prevent resource exhaustion

### 7.2 Performance Aspects

- Optimized chunk size selection
- Efficient similarity search algorithms
- Asynchronous request handling
- Caching mechanisms

## 8. References

- Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks – Lewis et al., 2020
  Introduced the RAG architecture, combining retrieval and generation for knowledge-based question answering.
- FastAPI Documentation
  Used for building the backend APIs and handling requests in the system.

- Billion-scale similarity search with GPUs – Johnson et al., 2017
  Describes FAISS, used for efficient vector similarity search.
- python-docx Documentation
  Used for extracting text from DOCX files.
- Vector Database Documentation
  Referenced for embedding storage and similarity search implementation.
- Large Language Model Documentation
  Referenced for model usage, embeddings, and response generation.