Project Report: AI-Powered RAG Chatbot for Document Analysis

Submitted by: [SRIKAR PILLA]

Date: December 14, 2025

Subject: Retrieval-Augmented Generation (RAG) System with Gemini & FastAPI

"A specialized RAG chatbot designed to answer technical queries in Machine Learning (ML), Deep Learning (DL), Artificial Intelligence (AI), and Statistics for ML"

DEPLOYED WEBSITE LINK: https://ragchatbot-ybbddsjzrpquziut8phhai.streamlit.app/

1. Objective

The goal of this project was to build a Retrieval-Augmented Generation (RAG) chatbot capable of answering user queries based strictly on the content of uploaded PDF documents. The system needed to analyze logical thought processes, ensure precision by minimizing hallucinations, and be accessible via an API.

Key Requirements:

Ingest multiple PDF files (7-10 documents).

Store data in an open-source Vector Database.

Use Google Gemini as the LLM (Large Language Model).

Expose the functionality via FastAPI.

Demonstrate usage via cURL and SDK.

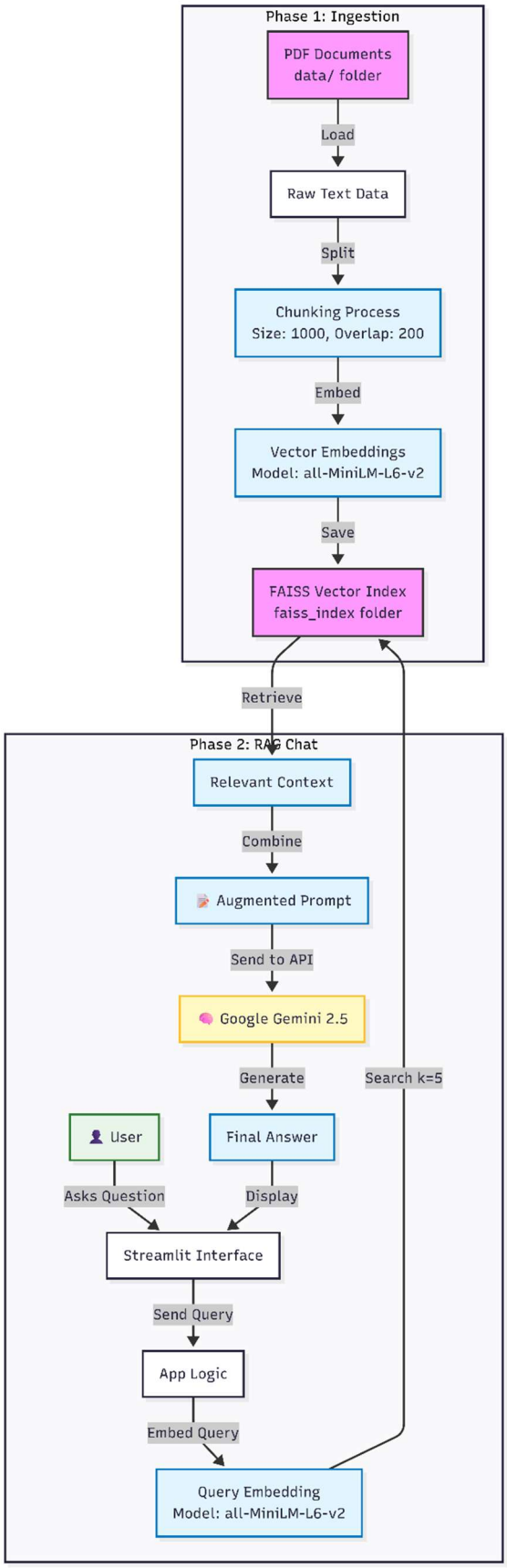
(Optional) Provide a user-friendly UI.

2. System Architecture

The application follows a standard RAG workflow, separated into three modular components:

1. The Brain (rag_engine.py): Handles document loading, splitting, embedding, and the RAG chain logic.
2. The Server (api.py): A FastAPI backend that acts as the bridge between the user and the logic.
3. The Interface (ui.py): A Streamlit frontend for easy interaction.

Workflow Diagram


Phase 1: Ingestion
PDF Documents
data/ folder
Load
Raw Text Data
Split
Chunking Process
Size: 1000, Overlap: 200
Embed
Vector Embeddings
Model: all-MiniLM-L6-v2
Save
FAISS Vector Index
faiss_index folder
Retrieve
Phase 2: RAG Chat
Relevant Context
Combine
Augmented Prompt
Send to API
Google Gemini 2.5
Generate
Search k=5
User
Final Answer
Asks Question
Display
Streamlit Interface
Send Query
App Logic
Embed Query
Query Embedding
Model: all-MiniLM-L6-v2

1. Ingestion: PDF $rightarrow$ Chunks $rightarrow$ HuggingFace Embeddings $rightarrow$ FAISS Vector DB.

2. Query: User Question $rightarrow$ Vector Search (Top 10 chunks) $rightarrow$ Con + Question sent to Gemini $rightarrow$ Answer.

3. Technology Stack & Justification

LLM : Google Gemini (Flash Latest)

Selected for its high speed, large con window, and free-tier availability. It provides concise and logical answers.

Embeddings: HuggingFace (all-MiniLM-L6-v2)

Crucial Decision: I used a local CPU-based embedding model instead of Googles API. This eliminated Rate Limit (429) errors during the heavy ingestion of multiple PDFs.

Vector DB : FAISS (Facebook AI Similarity Search)

A lightweight, open-source vector store that runs locally. It is faster than cloud databases for this scale and requires no setup.

Orchestrator : LangChain (v0.3)

Managed the complex pipeline of splitting , connecting to the Vector DB, and formatting prompts for the LLM.

Backend : FastAPI

Chosen for its asynchronous performance and automatic Swagger UI documentation.

Frontend : Streamlit Allowed me to build a professional-grade UI in pure Python without needing HTML/CSS knowledge.

4. Key Configurations & Parameters

Temperature (0.3):

I set the temperature low (0.3) to ensure Logic Precision. A lower temperature makes the model more deterministic and factual, reducing the risk of "hallucinations" (inventing facts not in the PDFs).

Chunk Size (1000) & Overlap (200):

PDFs were split into chunks of 1000 characters. The 200-character overlap ensures that sentences arent cut off in the middle, preserving the meaning across chunk boundaries.

Retrieval Count (k=10):

Initially, I set k=4, but the model sometimes lacked con. I increased this to k=10 to provide the model with more comprehensive information from the documents before answering

5. Code Implementation Details

A. The Engine (rag_engine.py)

This script uses PyPDFDirectoryLoader to scan the data/ folder.

Vectorization: It uses HuggingFaceEmbeddings to convert into numerical vectors.

Storage: These vectors are saved locally in a faiss_index folder.

RetrievalQA Chain: This LangChain component takes a user query, searches the FAISS database for relevant matches, and constructs a prompt for Gemini.

B. The API (api.py)

I implemented two main endpoints using FastAPI:

POST /ingest: Triggers the document processing pipeline.

POST /ask: Accepts a JSON payload {"query": "..."} and returns the answer plus the source document names.

C. The UI (ui.py)

I added a "Smart Layer" to the UI. It detects simple greetings (like "Hi" or "Thanks") and responds instantly without querying the database/API. This saves costs and makes the bot feel more responsive.

6. Challenges Faced & Solutions

During development, I encountered several technical hurdles. Here is how I solved them:

1. Dependency Conflicts ("Dependency Hell")

Problem: The latest versions of langchain (v0.3) were incompatible with older pydantic versions, causing ModuleNotFoundError.

Solution: I created a strict requirements.txt file pinning specific versions (e.g., langchain==0.3.0 and langchain-google-genai==2.0.0) to ensure stability.

2. Google API Rate Limits (Error 429)

Problem: When using gemini-2.0-flash or the Google Embedding API, the system frequently crashed with "Quota Exceeded" errors during ingestion.

Solution:

1. I switched the Embedding Model to run locally (HuggingFace) so ingestion is free and unlimited.
2. I switched the LLM to gemini-flash-latest, which proved to be more stable for this account tier.

3. "I Dont Know" Responses

Problem: The bot was too strict and often said "I dont know" even when the answer was in the document.

Solution: I increased the con window (k=10) so the bot reads 10 chunks instead of just 1, giving it a broader understanding of the topic.

7. Setup & Execution Guide (A-Z)

Step 1: Prerequisites

Ensure Python 3.10 is installed.

Step 2: Install Dependencies

Create a requirements.txt file with the following pinned versions:

fastapi

uvicorn

python-multipart

langchain==0.3.0

langchain-community==0.3.0

langchain-core==0.3.0

langchain--splitters==0.3.0

langchain-google-genai==2.0.0

google-generativeai==0.8.3

sentence-transformers==3.0.1

faiss-cpu==1.8.0

python-dotenv

streamlit

requests

Run installation:

pip install -r requirements.txt

Step 3: API Key Setup

Create a .env file in the project root:

GOOGLE_API_KEY=your_api_key_here

Step 4: Execution

To run the full system, open two separate terminals:

Terminal 1 (Backend Server):

uvicorn api:app --reload

Wait for "Application startup complete".

Terminal 2 (Frontend UI):

streamlit run ui.py

Step 5: Usage (cURL)

As per the requirement to demonstrate API usage, here is the cURL command to query the bot:

```
curl -X POST "http://127.0.0.1:8000/ask"
    -H "Content-Type: application/json"
    -d "{"query": "Summarize the document."}"
```

8. Conclusion

The final system successfully meets all project requirements. It processes multiple PDFs efficiently using local embeddings, avoiding API bottlenecks. The integration of FastAPI and Streamlit results in a robust, user-friendly application that provides accurate, con-aware answers with high logic precision.