# AI Chat Agent:

## I. Executive Summary: High-Performance RAG System

This project delivers a fully operational Generative AI Chat Agent based on a modular, decoupled architecture leveraging Retrieval-Augmented Generation (RAG). All core requirements including support for PDF, TXT, and MD files, integration with LangGraph for state management, FAISS for high-speed indexing, and deployment via a FastAPI backend—have been fulfilled.

The architecture is built on a Service Layer (FastAPI) decoupled from the Core Agent Layer (LangGraph). This strict separation ensures robust performance, scalability, and seamless integration into existing enterprise environments, as the intelligence core can be consumed via a stable, high-throughput API.

## II. System Architecture and Design Rationale

The system operates on a three-layer model designed for separation of concerns and maintainability:

1. **Presentation Layer (Streamlit):** Handles user interface, file upload mechanism, and displaying responses.
2. **Service Layer (FastAPI):** Acts as the high-throughput, low-latency API gateway. It manages request validation, file handling, and delegates execution to the Core Agent.
3. **Core Agent Layer (LangGraph, RAG Components):** Contains the stateful logic, document processing pipeline, vector store, and LLM generation step.

### Technology Stack Justification

|  |  |  |
| --- | --- | --- |
| Component | Layer | Description |
| **FastAPI** | Service | Selected for superior performance (among the fastest Python frameworks) and robust data handling. Automatically enforces data validation and adheres to the OpenAPI standard, generating built-in documentation. |
| **LangGraph** | Core Agent | Provides a professional, low-level framework for building durable, stateful, and long-running agents. Its foundation enables seamless integration of complex agent trajectories, such as conditional routing or self-correction, in future iterations. |
| **FAISS** | Core Agent | Critical for scalability and efficiency. Utilizes optimized algorithms for lightning-fast similarity search, capable of handling datasets scaling into billions of vectors. Supports local persistence to avoid costly re-embedding. |
| **BGE (v1.5)** | Core Agent | State-of-the-art dense retrieval model for RAG. Transforms text into a high-fidelity 384-dimensional vector. Ensures semantically accurate retrieval through improved similarity distribution. |

## III. The RAG Pipeline Implementation

The RAG pipeline manages document ingestion, transformation, and indexing, forming the knowledge base.

### A. Document Ingestion and Segmentation

* **Loaders:** Specialized loaders are used for optimal text extraction: PyPDFLoader, TextLoader, and UnstructuredMarkdownLoader.
* **Chunking Strategy:** The RecursiveCharacterTextSplitter is configured with a 1000-character segment size and a 200-character overlap.
  + **1000 Chars:** Provides high contextual density for the LLM to synthesize comprehensive answers.
  + **200 Overlap:** A technical buffer that ensures semantic continuity, mitigating the risk of critical context being fragmented at chunk boundaries.

### B. Embedding and Indexing

* **Embedding Model:** The BAAI/bge-small-en-v1.5 model generates 384-dimensional vectors for text representation.7 Embeddings are explicitly normalized (normalize\_embeddings=True) for precise cosine similarity comparison.
* **FAISS Persistence:** The vector store is persisted locally (faiss\_index.save\_local(...)) upon index creation. The Load\_Docs utility checks for pre-existing indices, loading them directly if found. This feature eliminates the computational cost of regenerating embeddings during subsequent runs, ensuring operational efficiency.3

## IV. Agent Orchestration via LangGraph

The LangGraph framework provides a reliable, stateful execution environment for the RAG process.

### A. Agent State Definition

The agent's state is formally defined by the State TypedDict, which rigorously tracks the essential components of the workflow: question, retrieved context (List), and the final answer.

### B. Execution Nodes and Flow

The workflow utilizes two sequential nodes:

1. **Retriver Node:** Executes a similarity\_search against the loaded FAISS index based on the user's question, populating the state with relevant document context.
2. **Generator Node:** Synthesizes the final response. It is governed by a prescriptive system prompt that strictly mandates using **only** the provided context. This critical step mitigates hallucination and ensures the generated answer is factually grounded in the source document.

The execution flow is linear: START Retriver Generator END.

## V. Service Layer Specification: FastAPI and Presentation Layer

### A. FastAPI Endpoints

|  |  |  |  |
| --- | --- | --- | --- |
| Endpoint | Method | Function | Result |
| /Upload\_File | POST | Accepts UploadFile, handles temporary storage, and invokes Load\_Docs. | Triggers document processing, chunking, and FAISS index creation (or loading). |
| /chat | POST | Accepts and validates the user question via Pydantic model. Invokes the LangGraph agent. | Returns the final, synthesized answer from the agent core. |

### B. Streamlit Presentation

Streamlit is utilized for rapid deployment of the user interface.8 It enforces the two-stage user workflow: File upload (communicating with /Upload\_File) followed by the interactive query session (communicating with /chat).

## VI. Deployment and Technical Specifications

### A. Mandatory Dependencies

The project requires a Python environment (minimum 3.10). The following dependencies must be installed to ensure the full RAG pipeline and service layer functionality:

* fastapi
* uvicorn (ASGI server for FastAPI)
* streamlit
* langchain-openai
* langchain-community
* langgraph
* faiss-cpu (CPU-only version of the vector store)
* pypdf (for PDF loading)
* unstructured (for Markdown and general text processing)
* python-dotenv

### B. Configuration Prerequisites

1. **Virtual Environment:** All installation and execution must occur within an isolated Python virtual environment.
2. **API Credentials:** The LLM service credentials must be securely defined in a .env file in the project root:
   * OPENROUTER\_API\_KEY
   * OPENAI\_BASE\_URL (Pointing to the OpenRouter endpoint)

### C. Execution Protocol (Two-Step Launch)

The system requires two distinct, sequential steps to become fully operational:

1. **Backend Service Start (FastAPI):**
   * **Command:** uvicorn main:app --reload
   * **Purpose:** Initializes the FastAPI service, loads the Core Agent (LangGraph, FAISS index loading utilities), and exposes the /Upload\_File and /chat API endpoints.
2. **Frontend Interface Launch (Streamlit):**
   * **Command:** streamlit run frontend.py
   * **Purpose:** Launches the user interface, which will automatically attempt to connect to the active FastAPI backend URL (http://127.0.0.1:8000).

## VII. Operational Readiness and Maintenance

### A. Scalability and Efficiency

System efficiency is driven by the persistent FAISS index.3 This methodology prevents the time-consuming operation of re-embedding documents, minimizing computational overhead. The current faiss-cpu configuration is highly efficient for most workloads, and the system is architected to allow a seamless upgrade to faiss-gpu for hardware acceleration if extremely high query volumes or massive document datasets are introduced.

### B. Modularity and LLM Flexibility

The system uses an abstract LLM connection. Switching the underlying model (e.g., migrating to GPT-4, or a compliant internal model) is a low-friction process requiring modification only of the connection parameters in the .env file or the llm instantiation block, without any changes to the core RAG or orchestration logic.

### C. Data Governance Recommendation

To maintain optimal resource utilization and security, a scheduled data governance procedure is recommended. This procedure should periodically identify and delete temporary files created during the file upload process and ensure that only relevant FAISS index directories are retained on disk.

#### Works cited

1. FastAPI, <https://fastapi.tiangolo.com/>
2. LangGraph, <https://langchain-ai.github.io/langgraph/>
3. <https://medium.com/@amrita.thakur/understanding-faiss-vector-store-and-its-advantages-cdc7b54afe47>
4. Faiss | <https://python.langchain.com/docs/integrations/vectorstores/faiss/>
5. Streamlit - ️ LangChain, <https://python.langchain.com/docs/integrations/callbacks/streamlit/>