RAG Chatbot Assessment

Assessment Overview

Develop a Retrieval-Augmented Generation (RAG) chatbot using FAISS as a vector database and LLM API (such as Gemini, OpenAI's GPT, or Hugging Face models), and LangChain's create_retrieval_chain. The goal is to create a chatbot that can provide accurate and contextually relevant responses to user queries.

Key Focus Areas

- **RAG Workflow**: Efficient integration of retrieval-based search with response generation.
- **Vector Database**: FAISS for effective vector-based data storage and retrieval.
- **LLM API Integration**: Use a suitable LLM API, including options like Gemini, OpenAI, or Hugging Face models.
- **Document Loading**: Utilize document loaders for processing data sources (PDFs, Word documents).
- **API Design**: Build a user-interactive API endpoint for seamless querying.
- **LangChain Usage**: Leverage LangChain's create_retrieval_chain to handle document retrieval within the RAG process.

Assessment Tasks

1. **Architecture Design**
    - **Task**: Design a high-level architecture for the RAG chatbot.
        - **Components**: Include an LLM (Gemini, OpenAI, or Hugging Face), FAISS vector database, LangChain's retrieval chain, and a chat API endpoint.
    - **Deliverable**: Submit a concise architecture overview.
2. **Document Loading and FAISS Vector Store Setup**
    - **Task**: Use LangChain's document loaders to load data from PDF or doc files and prepare them for retrieval with FAISS.
        - Load any publicly available documents (such as Wikipedia articles or research papers) to build data, and index this data in FAISS.
        - Configure retrieval to return the top 7 similar documents.
    - **Deliverable**: A FAISS-based vector store
3. **LLM API Integration with LangChain**
    - **Task**: Integrate an LLM API (Gemini, OpenAI, or Hugging Face models) using LangChain's create_retrieval_chain.
        - Ensure that FAISS retrieval provides relevant context to the LLM, which generates a response.

        - Handle API authentication securely using environment variables (do not hardcode API keys).
    - **Deliverable**: Effective integration of the LLM API, ensuring accurate, context-based responses.
4. **Chat API Endpoint Implementation**
    - **Task**: Create a /chat endpoint that:
        - Accepts user questions, retrieves the top 7 relevant documents via FAISS, and generates a response using the LLM.
        - Returns a JSON response that includes the chatbot's answer, along with an optional list of source document filenames.
    - **Deliverable**: A functional and documented chat API endpoint.
5. **Optional Development (Encouraged but not required)**
    - **Memory Management**: Implement memory to retain conversation context across interactions.
    - **Source Tracking**: Include a list of source document filenames with each response.

Additional Evaluation Criteria

- **Documentation**: Provide clear API documentation, including endpoint details, request/response formats, and sample calls.
- **Code Quality**: Ensure modular, readable code following Python best practices.
- **Efficiency**: Optimize for retrieval speed, response generation, and API performance.

Submission Guidelines

- **Framework**: You may use any Python framework, such as Flask or FastAPI, for the implementation.
- **Repository/Zip File**: Push all code and documentation to a public GitHub repository (with at least 3–5 commits), and share the link with us. Alternatively, you can submit a zipped file of your project if preferred.
- **README**: Include a detailed README with setup instructions, environment variable usage, and guidance for running the application.
- **Timeline:** Complete the main tasks within 3 days. Share your progress by the deadline.

Important Notes

- **Data Source Flexibility**: Use any publicly available documents for data retrieval.
- **API Key Security**: Use your own API key, but do not share it in the code or repository or with us. Use environment variables for secure handling