**Technical Report: Building a RAG System with Ollama, Llama2, and FastAPI**

**Overview**

This project focuses on building a **Retrieval-Augmented Generation (RAG)** system for document management and question answering. The backend is built using **FastAPI** to handle document upload, search, and question answering. The **Next.js frontend** interacts with the FastAPI backend to allow users to upload documents and query the system for answers based on the stored content.

The system leverages **Ollama**, an open-source platform, for managing embeddings and running generative models, specifically the **Llama2** model for text generation and the **nomic-embed-text** embedding model for document retrieval. The **Chroma** database is used to store embeddings, and **LangChain** is used to manage the interaction between the backend, database, and models.

**Approach and Architecture**

The architecture of the solution is designed to handle various document processing tasks, including uploading, embedding, searching, and generating responses. Here's a brief breakdown:

1. **Document Upload**: Users can upload documents (PDFs) through the FastAPI backend. These documents are then processed into smaller text chunks, and embeddings are generated using the nomic-embed-text model provided by **Ollama**. The embeddings are stored in **Chroma DB**, enabling efficient retrieval of relevant information.
2. **Search and Retrieval**: When a user submits a query, the system retrieves the most relevant documents by comparing the embeddings of the query with those stored in the Chroma database. The response is then generated by the Llama2 model, which is served by **Ollama** for text generation.
3. **Backend and Frontend**: The FastAPI backend exposes endpoints for document upload, querying, and other interactions. The Next.js frontend provides an interface for users to upload documents, ask questions, and view responses. The system operates in containers, with separate Dockerfiles for both the FastAPI backend and the Next.js frontend, making it easy to deploy and manage.
4. **Ollama Container**: Ollama is used to manage both the embedding generation and text generation processes. The embeddings are generated with the nomic-embed-text model, and responses are generated using the Llama2 model. These models are managed within a Docker container, and they are pulled and initialized upon container startup.

**Challenges and Solutions**

During the development of this project, several challenges were encountered and addressed as follows:

1. **Handling Large Datasets**: One of the primary challenges was processing large documents for embedding generation. To solve this, documents were broken down into smaller chunks, ensuring that each chunk could be processed and stored individually in the Chroma database for efficient retrieval.

2. **Containerizing Multiple Services**: Managing multiple services in a single deployment pipeline (FastAPI backend, Next.js frontend, and Ollama server) posed a challenge. The solution was to create separate Dockerfiles for each service, ensuring that they could be built and managed independently while maintaining smooth interaction between them.

3. **Model Integration Issues**: Initially, there were challenges related to loading and ensuring the availability of models within the Ollama container. This was resolved by explicitly pulling the required models using docker exec commands to guarantee that the models were available for inference when needed.

4. **Defining the Right Similarity Threshold**: Given the use of Ollama's embedding model (nomic-embed-text), one of the challenges was determining the right threshold for similarity scoring during document retrieval. While more advanced commercial models (like AWS Bedrock or ChatGPT embeddings) may offer superior performance, the decision was made to use Ollama's open-source model for this challenge to meet the project's technical requirements.

5. **Linking Results to Specific Document Sections**: Another challenge was linking the search results back to the specific section of the document that generated the response. This was solved by assigning unique IDs to each chunk of text created during the document splitting process. These IDs are formatted as pdf:page:chunk_id, ensuring that each chunk has a clear reference back to its origin in the document.

6. **CORS Issues**: Cross-origin resource sharing (CORS) issues occurred during development when the Next.js frontend made API requests to the FastAPI backend. These issues were resolved by configuring CORS in the FastAPI backend, allowing requests from the frontend to be processed without issues.

**Potential Improvements**

While the current solution is functional and meets the project's requirements, there are several areas where it could be improved:

1. **Increased File Format Compatibility**: The solution currently supports only PDF documents. Expanding compatibility to include additional formats, such as Word (.docx), plain text (.txt), and even presentations (.pptx), would make the system more versatile and accessible for a wider range of document types.

2. **Multiple Document Upload**: Implementing a feature that allows users to upload multiple documents at once via a single API call would streamline the process for users who need to process large numbers of documents quickly, improving efficiency.

3. **Integration of More Advanced Models**: The current solution uses open-source models from Ollama, such as nomic-embed-text and Llama2. However, integrating more powerful, commercial models like **AWS Bedrock** or **ChatGPT** for embeddings and text generation could significantly improve performance. These models would likely provide higher accuracy and faster response times, especially for larger datasets and more complex queries.

4. **Preprocessing and Document Cleaning**: To improve the quality of the embeddings and search results, it would be beneficial to implement a preprocessing and cleaning system for the uploaded PDF documents. This system could remove irrelevant text, handle image-based text, and standardize formatting to ensure that only clean, structured data is fed into the embedding model. This would lead to better performance when generating responses and retrieving relevant documents.