Documentation for the Enterprise RAG Web Application

Table of Contents

1. Introduction

Welcome! This document explains the journey I took in building a web application designed to analyse and compare market research reports using a Retrieval-Augmented Generation (RAG) backend. Think of it as a behind-the-scenes look at how I brought together modern AI techniques and web technologies to create a system that's both powerful and user-friendly.

2. Project Overview

This project is divided into three main components:

1. **The Frontend:** Built with Streamlit, this is the interface where users can easily type in their queries and see results in a clear, interactive format.
2. **The Backend:** Powered by FastAPI, this part acts as the middleman—receiving user queries from the frontend, talking to the RAG system, and sending the answers back.
3. **The RAG System:** This is where the magic happens. I experimented with several approaches:
    - My **initial attempt** was using DeepSeek1, and Ollama gave me a good starting point.
    - Then, integrated a **knowledge graph** using GraphDB and LangChain, which improved the relevance of the responses.
    - Finally, implemented a **multimodal approach** with LangChain that allowed to process not just text but also images and tables from PDFs.

3. Architecture

I designed the system to be modular, meaning each part can evolve independently.

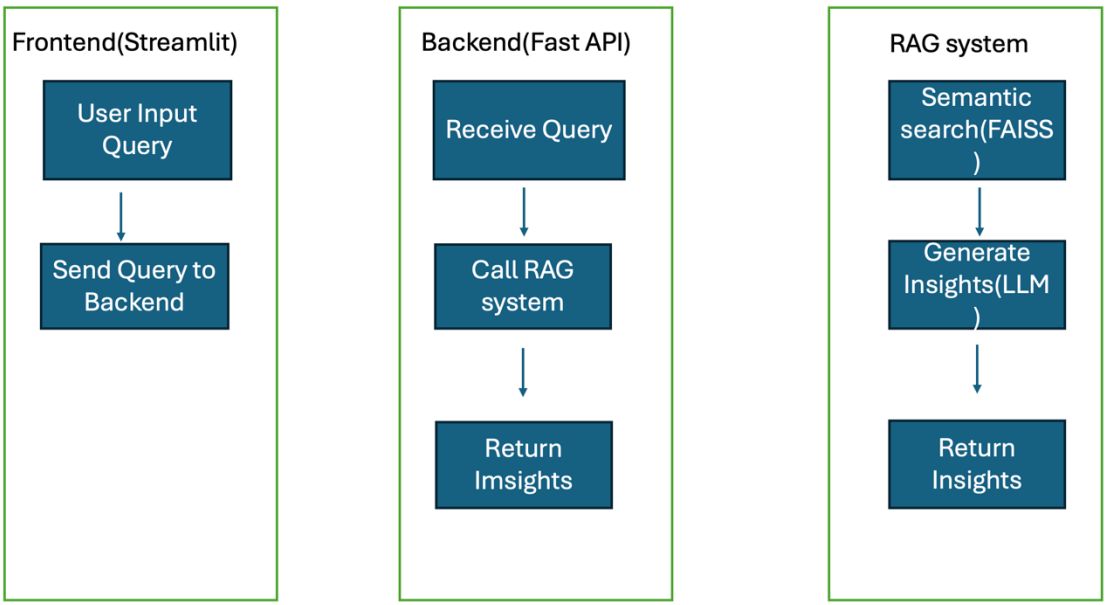

Fig: Architecture Diagram

Here's a quick rundown of how everything fits together:

- **User Interaction:** The journey begins with the user, who inputs their query via our Streamlit interface.
- **API Gateway:** The query is sent to our FastAPI backend, which acts as a reliable gateway.
- **RAG Engine:** This is the powerhouse that uses various techniques (including semantic search with FAISS and embedding storage with LangChain) to retrieve and generate the best answers.
- **Data Processing:** I even built the robust OCR pipelines to extract information from PDFs, ensuring that I don't miss out on any valuable data.
- **Response Delivery:** Finally, the processed response is sent back to the user through the FastAPI backend.

Imagine a flow where every component is carefully orchestrated to ensure speed, accuracy, and reliability.

4. Project Structure

Here's a quick look at our project directory:

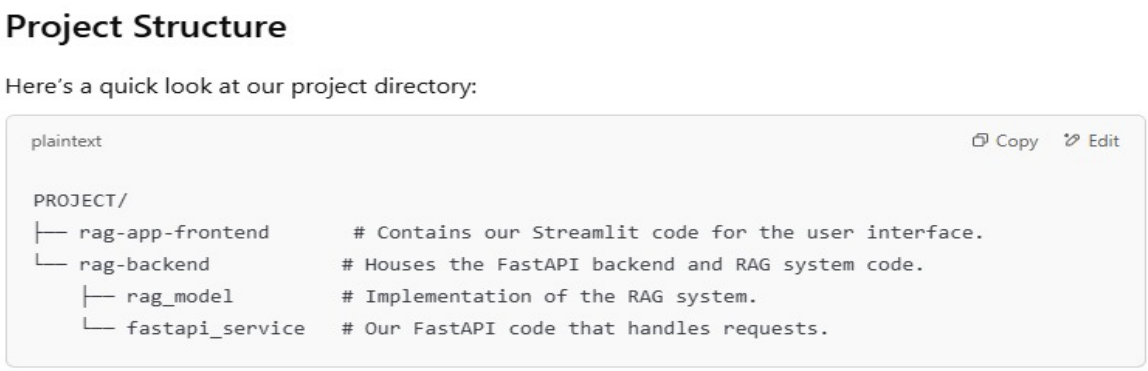

Fig 2: Project Structure

I kept things organised so that every piece is easy to locate, maintain, and upgrade as needed.

5. Code Implementation Details

Frontend: Streamlit

Frontend is all about making things simple for the user:

- **What It Does:** It provides a clean text box where users can type in their queries and see the results displayed in an interactive format.
- **How It Works:** When a user submits a query, the front end sends it over HTTP to our FastAPI backend.
- **Tech Used:** I used Streamlit because of its simplicity and ease of integration with Python.

Backend: FastAPI

The backend is the unsung hero that ties everything together:

- **Role:** It receives user queries from the frontend, sends them off to the RAG system, and then sends the results back.
- **Key Features:** We built robust API endpoints to handle requests and ensure that the communication between the frontend and backend is smooth.
- **Tech Stack:** FastAPI is my choice here due to its speed and scalability.

RAG System

This is the core of the project, and I tried out several strategies to make it as effective as possible:

Initial RAG Implementation with DeepSeekr1 and Ollama

- **What We Tried:** I began with DeepSeekr1 and Ollama to see how well a basic RAG approach would work.
- **Results:** While it worked to some extent, I noticed that the results were not as accurate as I'd hoped. Issues like ineffective document chunking and suboptimal embeddings became apparent.

Knowledge Graph with GraphDB and LangChain

- **Our Next Step:** To improve accuracy, I integrated a knowledge graph using GraphDB alongside LangChain.

- **Outcome:** This approach significantly boosted response relevance, though it did shine best with smaller data chunks. Handling very large documents still posed a challenge.

Multimodal RAG with LangChain

- **Our Most Advanced Approach:** Further enabled the system to process multiple data types—text, images, and tables—from PDFs.
- **Process:**
    - Used OCR to extract text from images.
    - GPT-4o-mini helps with vectorizing the extracted content.
    - The multimodal LLM then processes these vectors to generate rich, detailed responses.
- **Benefits:** This method enhanced the accuracy and quality of the responses, giving me a comprehensive tool that stands out from traditional text-only RAG systems.

6. Technologies Used

I am proud of the tech stack that made this project possible:

- **Frontend:** Streamlit, combined with Python HTTP libraries.
- **Backend:** FastAPI for a fast and scalable API.
- **RAG System:**
    - **FAISS** for efficient semantic search.
    - **LangChain's vector stores** for embedding storage and retrieval.
    - **GraphDB** for structuring the knowledge graph.
    - **DeepSeekr1 & Ollama** for the initial experiments.
    - **GPT-4o-mini and multimodal LLMs** for advanced processing.
- **Additional Tools:** ChromaDB to manage text from OCR, Hugging Face models for enhanced NLP tasks, and custom OCR pipelines to extract data from PDFs.

7. Challenges and Insights

Every project has its hurdles. Here are some that I encountered:

1. **Efficiency in Retrieval:**
    - My first attempt showed that document chunking and embedding quality are key. I learned a lot from those early experiments.
2. **Scalability:**
    - While the knowledge graph approach improved accuracy, it struggled with larger datasets. This was a vital learning experience.
3. **Multimodal Processing:**
    - Integrating text, images, and tables was challenging, but it ultimately led to much richer responses.
4. **Optimization:**
    - Balancing speed with response quality pushed us to fine-tune our system, leading to the integration of FAISS and LangChain's vector stores for a smoother experience.

8. Future Directions

There's always room to grow! Here's what I am looking at next:

- **Better Chunking Strategies:**
    - Refining the document-chunking methods to improve retrieval, especially for larger documents.
- **Scalability Solutions:**
    - Exploring distributed processing for the knowledge graph to handle bigger datasets.
- **Model Upgrades:**
    - Investigating the latest embedding models and fine-tuning strategies to further boost response quality.
- **Enhanced User Interface:**
    - Expanding the frontend to offer more advanced visualizations and analytics.
- **Rigorous Testing:**

- Building a more robust suite of tests to ensure that the system remains reliable as it scales.

9. Conclusion

In a nutshell, the RAG web application is a fusion of advanced AI techniques and modern web technologies designed to tackle the challenge of comparing market research reports. Through careful experimentation and integration of various technologies, I've built a system that not only delivers accurate insights but also provides a solid foundation for future enhancements.

This project is a testament to the power of iterative development, and I am excited about the possibilities that lie ahead. Thank you for taking the time to explore this journey—here's to many more innovations in enterprise AI!