Retrieval-Augmented Generation (RAG) System for Policy Documents

1. Project Objectives

The primary objective of this project was to design and implement a sophisticated question-answering (Q&A) system capable of accurately responding to queries based on a specific set of policy documents. Traditional language models, while powerful, often hallucinate or provide generic answers when dealing with domain-specific knowledge. To address this, the project aimed to leverage a Retrieval-Augmented Generation (RAG) architecture, which grounds the language model's responses in factual information retrieved from the policy documents. The success of the project was measured by the system's ability to provide concise, accurate, and contextually relevant answers to a variety of policy-related questions.

2. System Design

The system was designed using the LangChain framework, which provides a modular and extensible approach for building language model applications. The architecture consists of the following key components:

- **Document Loader:** The PyPDFDirectoryLoader was selected to efficiently load all PDF files from a designated directory, making the system scalable for an increasing number of policy documents.
- **Text Splitter:** To handle documents of varying lengths and to optimize the retrieval process, a RecursiveCharacterTextSplitter was used. This component breaks down large documents into smaller, semantically meaningful chunks with a defined overlap.
- **Embedding Model:** GoogleGenerativeAIEmbeddings was chosen to convert the text chunks into high-dimensional vectors, enabling a semantic search.
- **Vector Store:** The FAISS library was utilized as the vector store for its efficiency and speed in performing similarity searches on the document embeddings.
- **Retriever:** The vector store was configured as a retriever, responsible for fetching the top k most relevant document chunks for any given user query.
- **Language Model (LLM):** The gemini-1.5-flash model from the Google Generative AI API was integrated to act as the generative component, synthesizing the final answer from the retrieved context.
- **Orchestration:** LangChain's Expression Language (LCEL) was used to create a seamless pipeline that connects the retriever to the LLM via a custom prompt.

3. Implementation Details

The implementation was a multi-step process, starting with data ingestion and culminating in

a functional RAG pipeline.

3.1 Data Ingestion and Preprocessing

The project began by using the PyPDFDirectoryLoader to load the policy documents. Following this, the documents were processed by the RecursiveCharacterTextSplitter. A chunk_size of 1000 characters with a chunk_overlap of 100 was chosen to ensure that each chunk contained sufficient context while minimizing the risk of losing information at the boundaries.

3.2 Vectorization and Retrieval

Once the documents were chunked, each piece of text was converted into a vector embedding using GoogleGenerativeAIEmbeddings. These embeddings were then stored in a FAISS index, creating the core searchable knowledge base. The FAISS index was subsequently exposed as a retriever, configured to perform a similarity search for the top 5 most relevant document chunks (k=5).

3.3 RAG Chain Construction

The final step involved creating the RAG chain using LCEL. This chain defines the flow of information: the user's question is first used by the retriever to find relevant context from the FAISS store. This context, along with the original question, is then passed to a carefully crafted prompt template. The prompt instructs the LLM to use the provided context to answer the question, with strict rules to prevent the generation of ungrounded information.

The complete chain was defined as follows:

```
chain = (
   {'context': faiss_retriever, 'question': RunnablePassthrough()}
   | PROMPT
   | llm
   | output_parser
)
```

4. Challenges and Solutions

4.1 API Key Security

Challenge: Initially, API keys were hardcoded directly into the script, posing a significant security risk.
Solution: The code was refactored to use environment variables (os.getenv). This practice ensures that sensitive credentials are not exposed in the codebase, a critical step for project security and maintainability.

4.2 Optimizing Document Chunking

Challenge: Determining the optimal chunk_size and chunk_overlap was a challenge. Chunks that are too small lack context, while chunks that are too large may include irrelevant information, diluting the quality of the retrieved results.
Solution: The chosen chunk_size of 1000 characters was found to be a good balance, as it typically contained complete paragraphs or sections of information. The chunk_overlap of 100 characters helped maintain continuity between chunks.

4.3 Prompt Engineering

Challenge: Ensuring the LLM would adhere to the provided context and not generate information from its pre-existing training data was difficult. The model could sometimes provide a correct answer but without using the specific language or details from the policy documents.
Solution: The prompt template was carefully designed with explicit instructions, such as "Use the following pieces of context to answer the question" and "If you don't know the answer, just say that you don't know." These clear rules guided the LLM to behave as a grounded, contextual assistant.

5. Lessons Learned

This project highlighted several key lessons for building effective LLM applications:

- **Modularity:** LangChain's component-based approach simplifies the development of complex pipelines. By treating the retriever, prompt, and LLM as distinct, reusable components, the system is easier to debug and modify.
- **The Power of RAG:** RAG is an incredibly effective technique for grounding language models in domain-specific data. It successfully transforms a general-purpose LLM into a specialized knowledge agent, dramatically improving the accuracy and relevance of its responses.
- **Security Best Practices:** The importance of secure credential management cannot be overstated. Using environment variables is a fundamental and non-negotiable practice for any project.
- **Iterative Design:** Both chunk_size and prompt design require an iterative approach. Testing and refining these parameters are essential for maximizing the system's performance.

6. Conclusion

This project successfully delivered a functional RAG-based Q&A system for policy documents. By combining efficient data processing with powerful generative and retrieval models, the system achieved its objective of providing accurate, grounded answers. The experience also provided valuable insights into the practical aspects of building LLM applications, including the importance of secure coding practices and the art of prompt engineering.