**Report on Retrieval-Augmented Generation (RAG) Solution and vector database.**

**Introduction**

In this project, I developed a solution using Retrieval-Augmented Generation (RAG) technology to handle natural language queries by leveraging a pre-existing document database. A solution can understand and respond to questions based on the content of a given PDF document. Below are the key components and steps involved in creating this solution.

**Key Components and Workflow**

The solution was developed using several key components, each playing a important role in the overall workflow. Below, I explain each part of the code and how it contributes to the final solution.

**Environment Setup**

from dotenv import load\_dotenv

I begin by loading environment variables using the load\_dotenv function. This allows me to securely manage configuration settings such as API keys.

**Loading and Splitting Documents**

from langchain\_community.document\_loaders import PyPDFLoader

from langchain.text\_splitter import CharacterTextSplitter

pdf\_path = "data/staff.pdf"

loader = PyPDFLoader(file\_path=pdf\_path)

documents = loader.load()

text\_splitter = CharacterTextSplitter(

chunk\_size=1000, chunk\_overlap=50, separator="\n"

)

docs = text\_splitter.split\_documents(documents)

1. **Document Loader**: I use PyPDFLoader to load the PDF document specified by pdf\_path. This loader reads the PDF and extracts its content into a usable format.
2. **Text Splitter**: The CharacterTextSplitter splits the extracted content into manageable chunks. I set a chunk\_size of 1000 characters with an overlap of 50 characters. This helps in handling large documents and ensures the text is appropriately divided for further processing.

**Generating Embeddings**

from langchain\_openai import OpenAIEmbeddings

embeddings = OpenAIEmbeddings()

Embeddings are numerical representations of text that capture semantic(relating) meaning. I use OpenAIEmbeddings to generate embeddings for the document chunks. These embeddings enable efficient comparison and retrieval of relevant text segments based on queries.

**Creating and Saving the Vector Store**

from langchain\_community.vectorstores.faiss import FAISS

vectorstore = FAISS.from\_documents(docs, embeddings)

vectorstore.save\_local("vector\_db")

1. **Vector Store**: I use FAISS (Facebook AI Similarity Search) to create a vector store from the document embeddings. This store allows me to quickly retrieve relevant document chunks based on similarity to the query embeddings.
2. **Saving the Vector Store**: The vector store is saved locally as vector\_db. This step ensures I can reload the vector store for future use without regenerating the embeddings.

**Setting Up the Retrieval Chain**

from langchain import hub

from langchain.chains import create\_retrieval\_chain

from langchain.chains.combine\_documents import create\_stuff\_documents\_chain

from langchain\_openai import ChatOpenAI

retrieval\_qa\_chat\_prompt = hub.pull("langchain-ai/retrieval-qa-chat")

llm = ChatOpenAI()

combine\_docs\_chain = create\_stuff\_documents\_chain(llm, retrieval\_qa\_chat\_prompt)

1. **Loading the Retrieval Prompt**: I pull a predefined prompt for retrieval-based question answering from the LangChain hub.
2. **Language Model**: I use ChatOpenAI, a language model capable of understanding and generating human-like text.
3. **Combining Document Chunks**: The create\_stuff\_documents\_chain function combines the retrieved document chunks into a cohesive response using the language model and the prompt.

**Creating and Using the Retrieval Chain**

retriever = FAISS.load\_local("vector\_db", embeddings).as\_retriever()

retrieval\_chain = create\_retrieval\_chain(retriever, combine\_docs\_chain)

1. **Loading the Vector Store**: I load the previously saved vector store and set it up as a retriever.
2. **Retrieval Chain**: The retrieval chain is created using the retriever and the document combination chain. This chain handles the end-to-end process of retrieving relevant document chunks and generating a response.

**Handling User Queries**

response = retrieval\_chain.invoke(

{"input": "What is office of Ms. Josephine K. Stephen"}

)

print(response["answer"])

The retrieval chain is invoked with a user query, and the response is generated by retrieving relevant document chunks and generating a coherent answer using the language model.

**Detailed Explanation of Key Concepts**

**Embeddings**

Embeddings are vector representations of text that capture semantic(relating) meaning. They allow me to compare and retrieve text based on similarity. In this project, I use OpenAIEmbeddings to convert document chunks into embeddings.

**Vector Database**

A vector database stores embeddings and allows efficient similarity search. I use FAISS to create and manage our vector database. FAISS is optimized for fast retrieval of similar vectors, making it ideal for our RAG solution.

**Retrieval-Augmented Generation (RAG)**

RAG combines retrieval and generation to handle queries. The retriever fetches relevant document chunks based on the query, and the generator (language model) combines these chunks into a coherent response. This approach ensures that the generated responses are grounded in the source documents.