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

Introduction

In this project, we developed a solution using Retrieval-Augmented Generation (RAG) technology to handle natural language queries by leveraging a pre-existing document database. Our 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 vital role in the overall workflow. Below, we explain each part of the code and how it contributes to the final solution.

Environment Setup

```
from dotenv import load_dotenv
```

We begin by loading environment variables using the `load_dotenv` function. This allows us 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**: We 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. We 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 meaning. We 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**: We use `FAISS` (Facebook AI Similarity Search) to create a vector store from the document embeddings. This store allows us 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 we 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**: We pull a predefined prompt for retrieval-based question answering from the LangChain hub.
2. **Language Model**: We 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**: We 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 us to compare and retrieve text based on similarity. In this project, we use `OpenAIEmbeddings` to convert document chunks into embeddings.

Vector Database

A vector database stores embeddings and allows efficient similarity search. We 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.