Retrieval Augmented Generation (RAG) — Basics (Part-1)

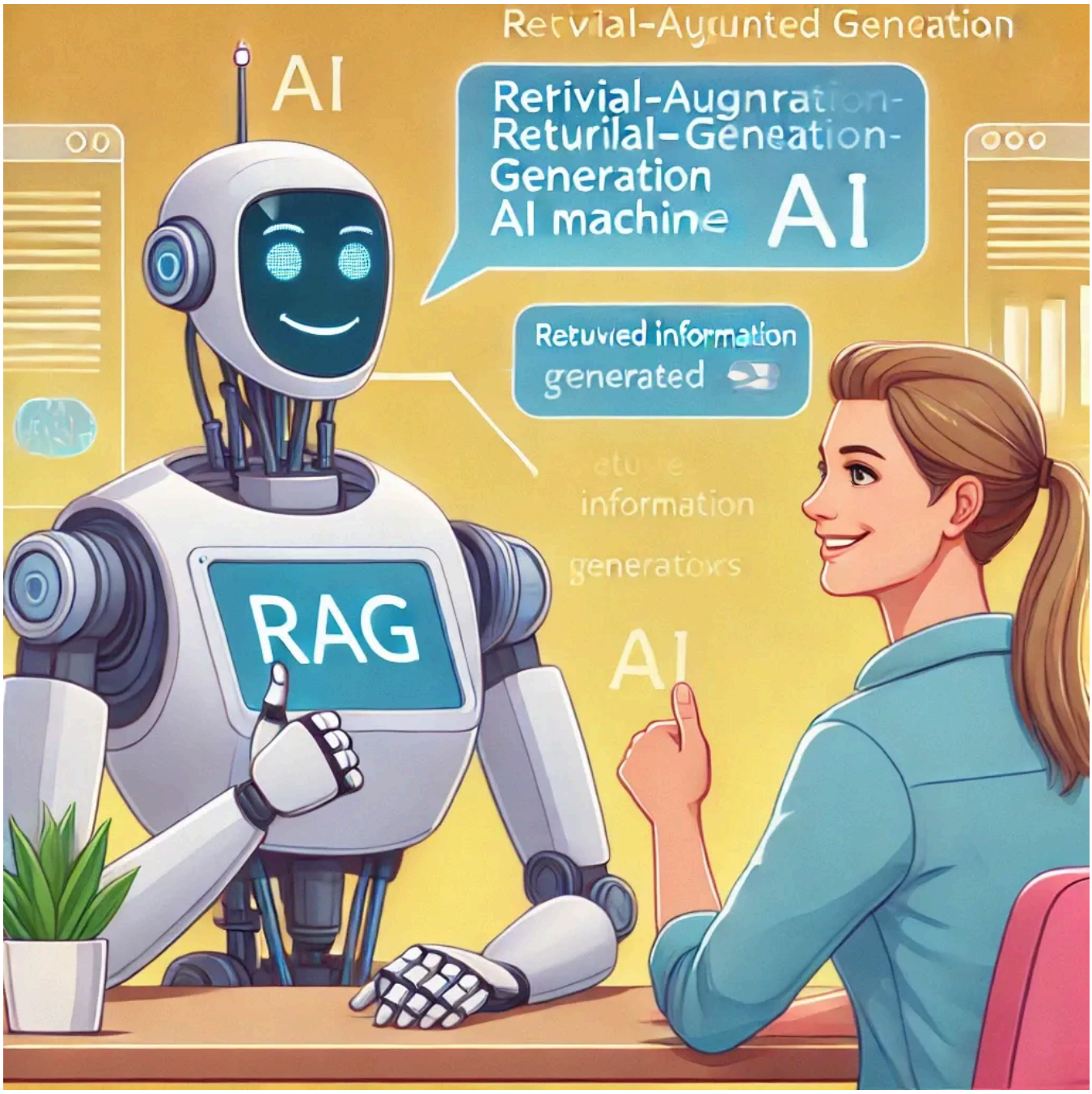
Shravan Kumar · Follow

8 min read · Aug 14, 2024

Listen Share

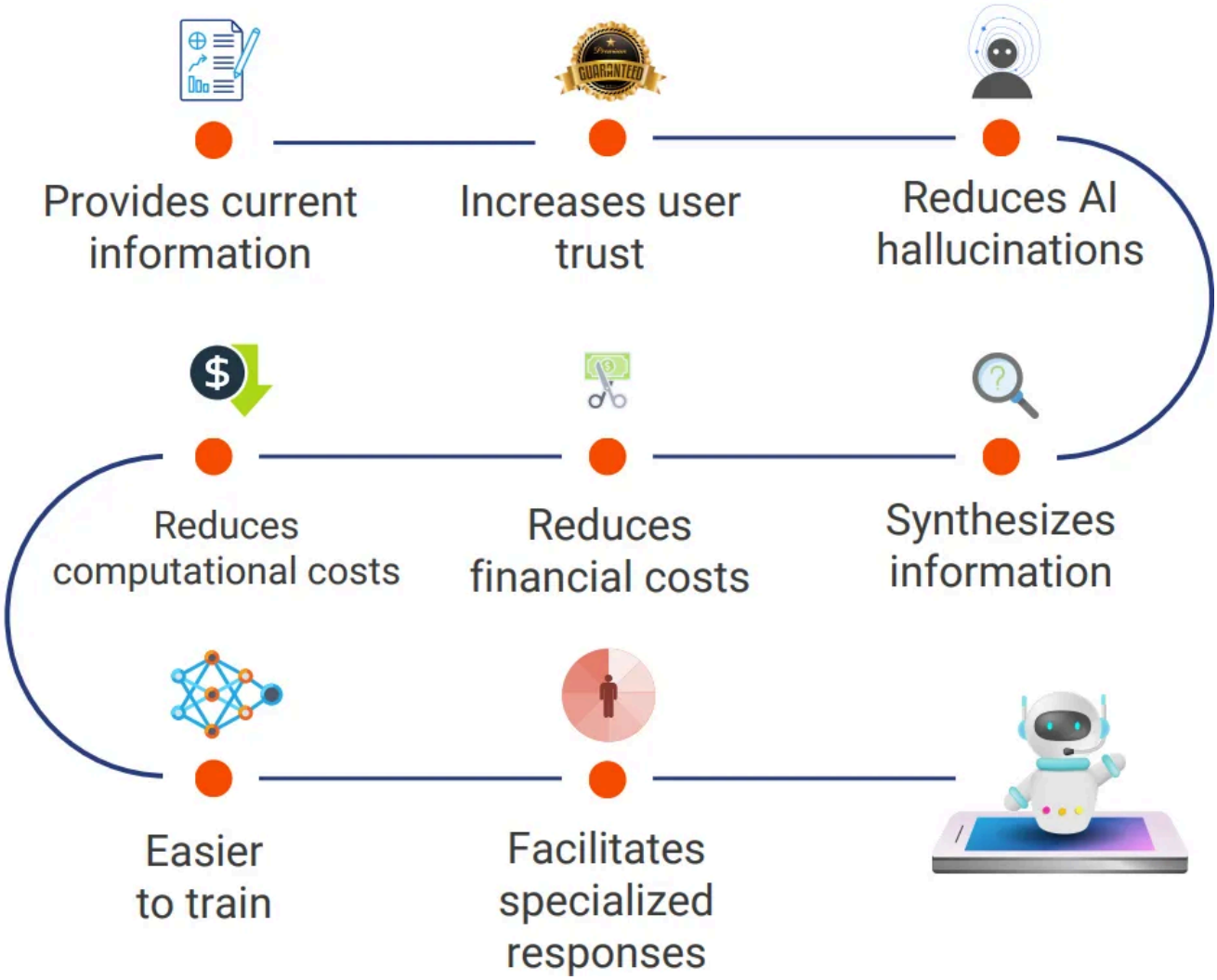
Since the AI revolution with Large Language Models (LLMs), the one key topic we must have heard is RAG . Here we will discuss in detail with concepts and code about What is RAG? How do we use it? What is the high level overview about this RAG.

RAG is a framework that connects external sources of information with the LLM so that its responses are context-aware and accurate.



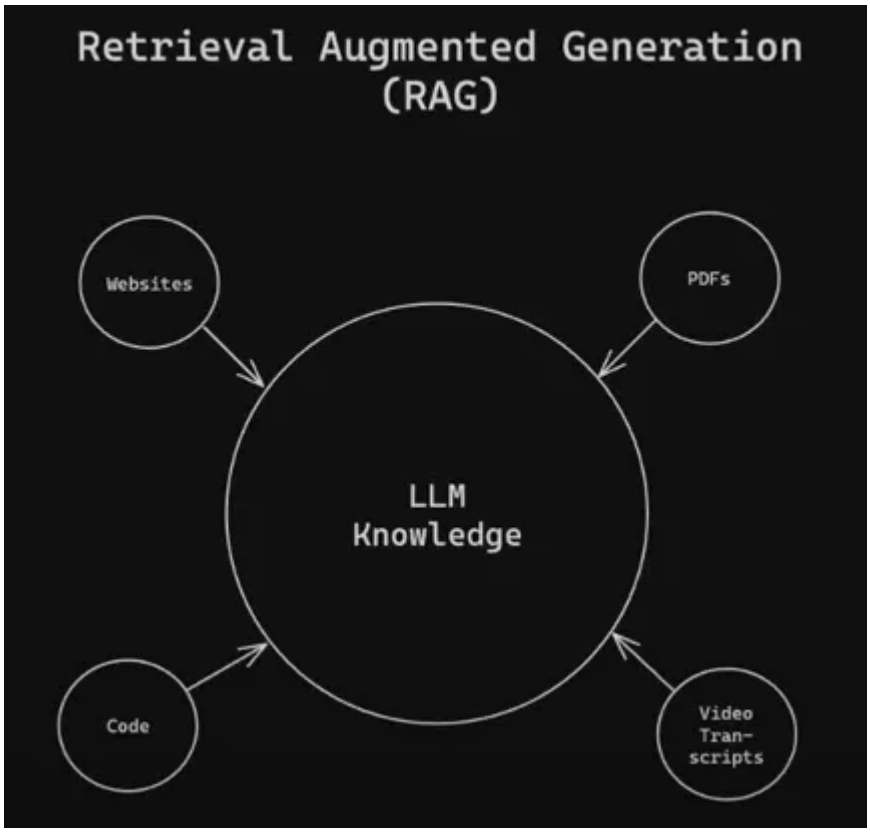
All the Large Language Models (LLMs) have a constraint on how much knowledge they have and that can be a problem whenever we are trying to answer some questions which is about additional things which is not present in the current context of data. For e.g., if the question is like what is happening at present in the current market for a particular stock price. This information cannot be retrieved from the context on which LLM is trained for as we know that all these LLMs are trained with certain time cutoff dates. Another example is that — suppose we are building a product within our Organization and it is focussed mainly with internal data and documents. So in this case again the LLMs cannot have visibility about these information — Hence RAG is helpful in these scenarios where this RAG provides additional information to these LLMs which can provide better answers. Hence the main goal of RAG is to enhance the model's responses with up-to-date,

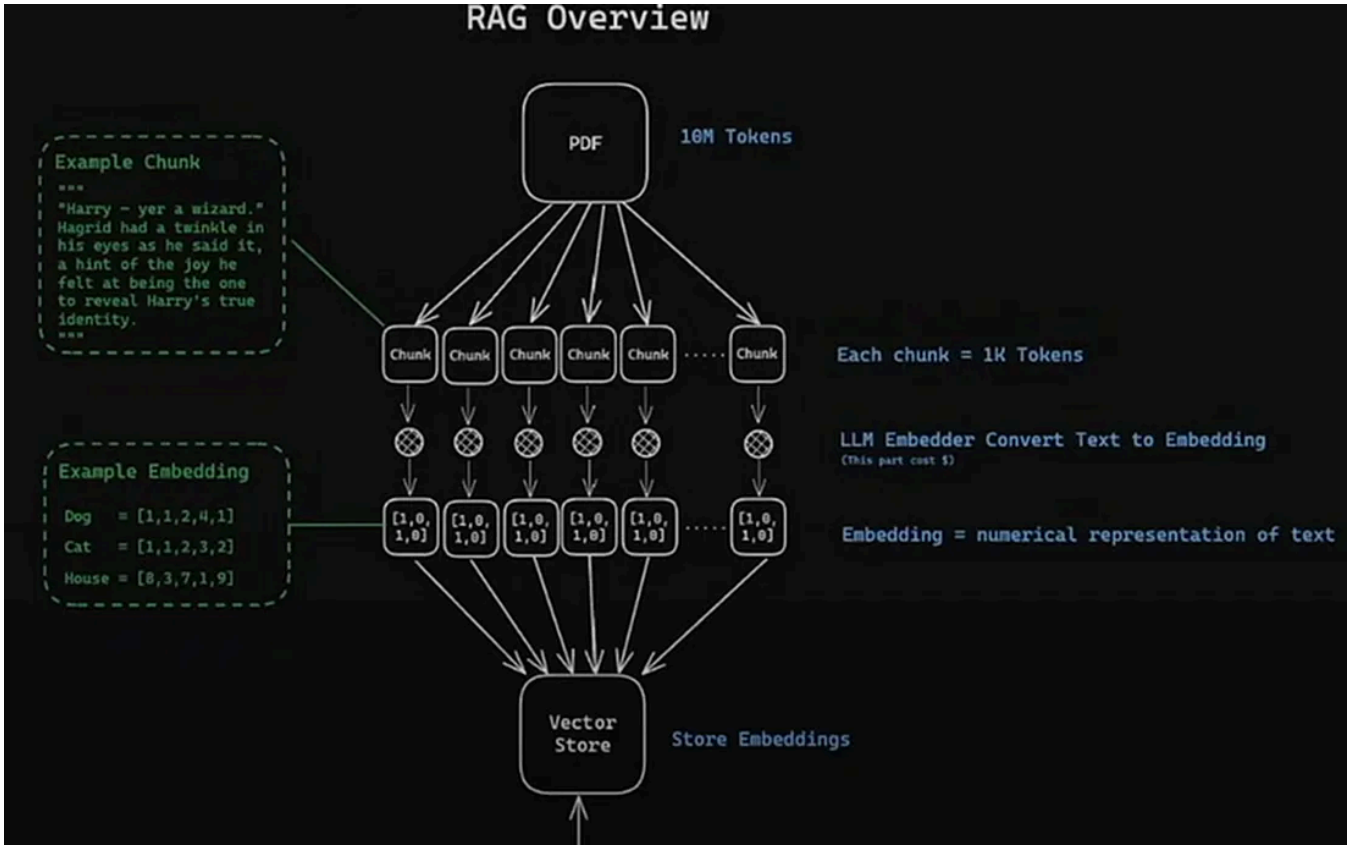
accurate, and contextually relevant information that may not be contained in its initial training data.



source: Data Science Dojo

So here below you can see that different types of data sources becomes as input data sources to LLMs.





Let us look at how the RAG process begin — the chart above is just for illustration purpose only :

Input Document (PDF with 10M Tokens):

- The document, which contains a large number of tokens (e.g., words or characters), is broken down into smaller, manageable chunks. Each chunk typically contains around 1,000 tokens. This makes it easier to process and analyze large texts.

Chunking Process:

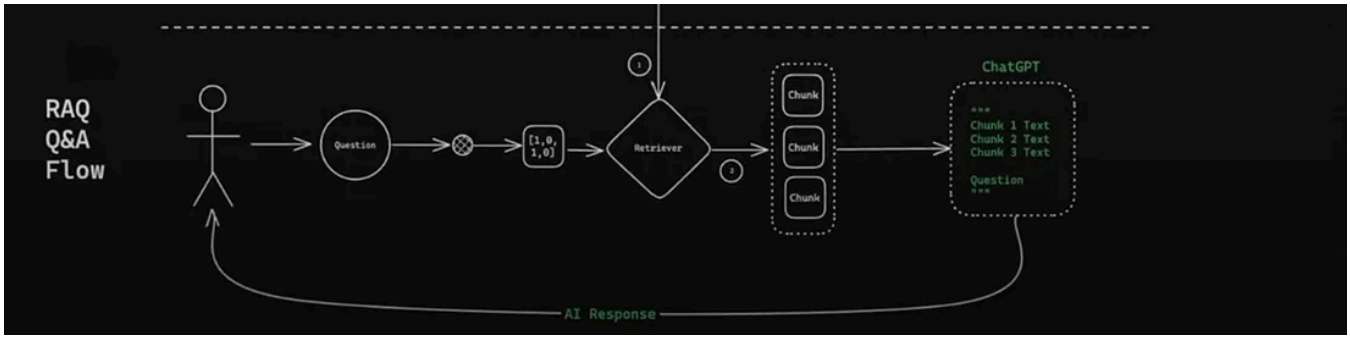
- The large text document is divided into smaller “chunks” of text. For instance, if you have a PDF document, it could be split into several sections or paragraphs, each considered a chunk.

Embedding Generation:

- Each chunk is then processed by a Language Model (LLM) embedder, which converts the text into an embedding — a numerical representation of the text. This embedding captures the semantic meaning of the text and is represented as a vector (a list of numbers). For example, different words like “Dog,” “Cat,” and “House” might be converted into vectors like [1,2,4,1], [1,2,3,2], and [0,3,7,9], respectively.

Vector Store:

- The embeddings (numerical representations) of all the chunks are stored in a vector store. This is a database optimized for storing and retrieving high-dimensional vectors. The vector store is used to quickly find relevant chunks based on a query.

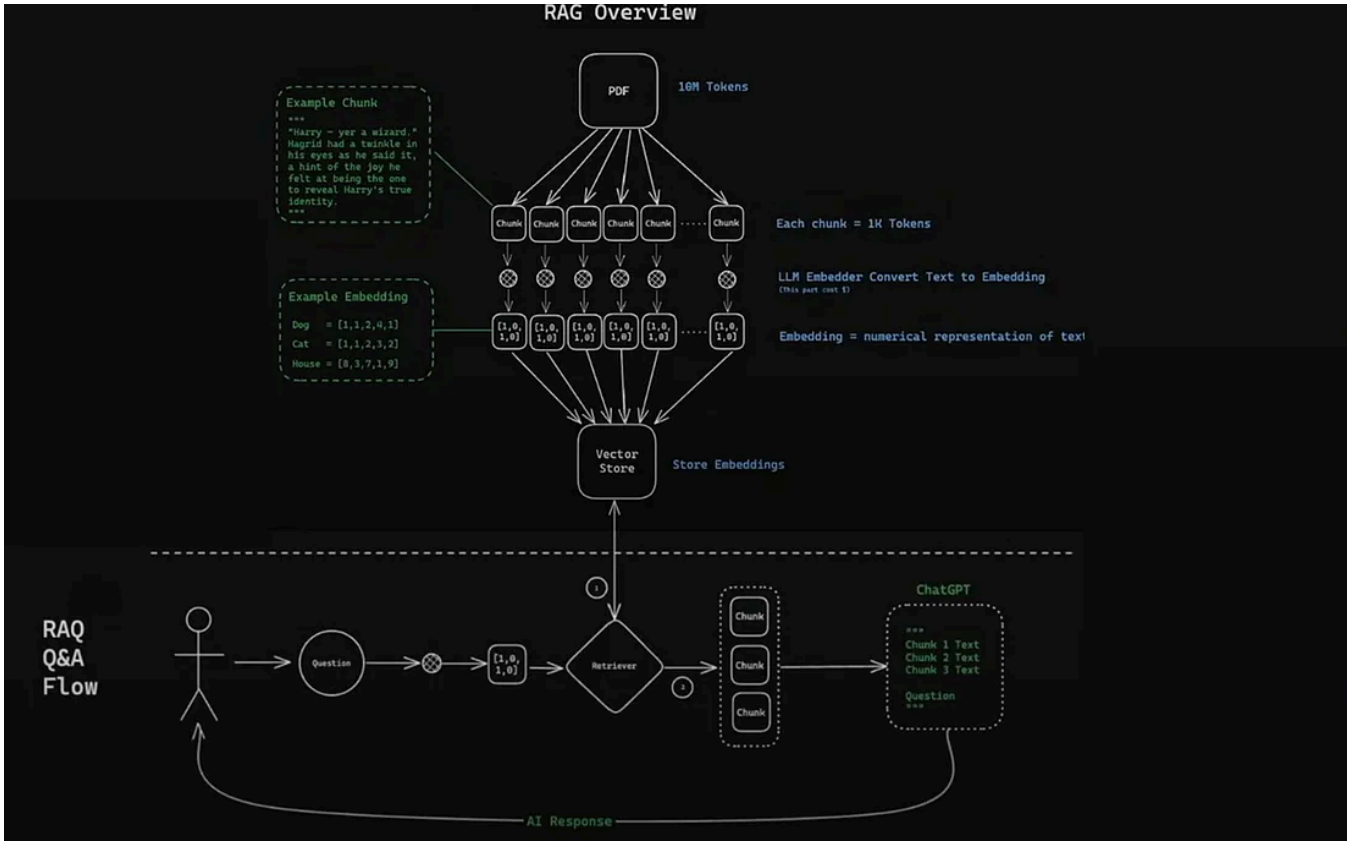


RAQ Q&A Flow

- **Question Input:** The process begins with a user posing a question (RAQ — Retrieval-Augmented Question Answering).

- **Retrieval Step:** The question is processed by a retriever model, which checks the stored chunks of text (from the vector store) to find the most relevant information. This step is based on similarity scoring (e.g., scores between 0.0 to 1.0).
- **Chunk Retrieval:** The retriever pulls out the most relevant chunks of text that are likely to contain the answer to the user's question.
- **AI Processing:** These retrieved chunks, along with the original question, are passed to an AI model like ChatGPT. The AI model processes the input and generates an appropriate response.
- **Response to User:** Finally, the AI's response is sent back to the user, completing the Q&A flow.

Combining the entire Q&A Flow along with the Vector Store looks like this.



This entire process combines retrieval and generation to create an intelligent system that can answer questions based on a vast amount of stored knowledge.

Let us look at basic example code to illustrate all the above functionalities.

Import Libraries:

```
import os

from langchain.text_splitter import CharacterTextSplitter
from langchain_community.document_loaders import TextLoader
from langchain_community.vectorstores import Chroma
from langchain_openai import OpenAIEmbeddings
```

These lines import necessary modules and classes. The `os` module is for file path manipulations, while `CharacterTextSplitter`, `TextLoader`, `Chroma`, and `OpenAIEmbeddings` are LangChain components used for text processing and vector storage.

Define Directories:

```
# Define the directory containing the text file and the persistent directory
current_dir = os.path.dirname(os.path.abspath(__file__))
file_path = os.path.join(current_dir, "books", "odyssey.txt")
persistent_directory = os.path.join(current_dir, "db", "chroma_db")
```

These lines define paths: `current_dir` sets the base directory, `file_path` points to the text file `odyssey.txt`, and `persistent_directory` specifies where the Chroma vector store will be stored.

Check Vector Store Existence:

```
# Check if the Chroma vector store already exists
if not os.path.exists(persistent_directory):
    print("Persistent directory does not exist. Initializing vector store...")
```

This checks if the vector store directory exists. If not, it prints a message and proceeds to initialize the store.

Verify Text File Existence:

```
    # Ensure the text file exists
    if not os.path.exists(file_path):
        raise FileNotFoundError(
            f"The file {file_path} does not exist. Please check the path."
        )
```

This ensures that the text file `odyssey.txt` is present. If it's missing, it raises an error with a descriptive message.

Load Text Content:

```
# Read the text content from the file
loader = TextLoader(file_path)
documents = loader.load()
```

`TextLoader` is used to read the contents of the file into a `documents` object, preparing it for further processing.

Split Document into Chunks:

```
# Split the document into chunks
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
docs = text_splitter.split_documents(documents)
```

The document is split into chunks using `CharacterTextSplitter` . Each chunk is 1000 characters long, with no overlap, facilitating better handling and processing of the text data.

Display Document Information:

```
# Display information about the split documents
print("\n--- Document Chunks Information ---")
```

```
print(f"Number of document chunks: {len(docs)}")
print(f"Sample chunk:\n{docs[0].page_content}\n")
```

This prints the number of chunks and shows a sample chunk to provide insight into how the document was divided.

Create Embeddings:

```
# Create embeddings
print("\n--- Creating embeddings ---")
embeddings = OpenAIEmbeddings(
    model="text-embedding-3-small"
)  # Update to a valid embedding model if needed
print("\n--- Finished creating embeddings ---")
```

An `OpenAIEmbeddings` object is created to generate embeddings for each chunk. The specific model used is `text-embedding-3-small`, though it should be replaced with an available and appropriate model if necessary.

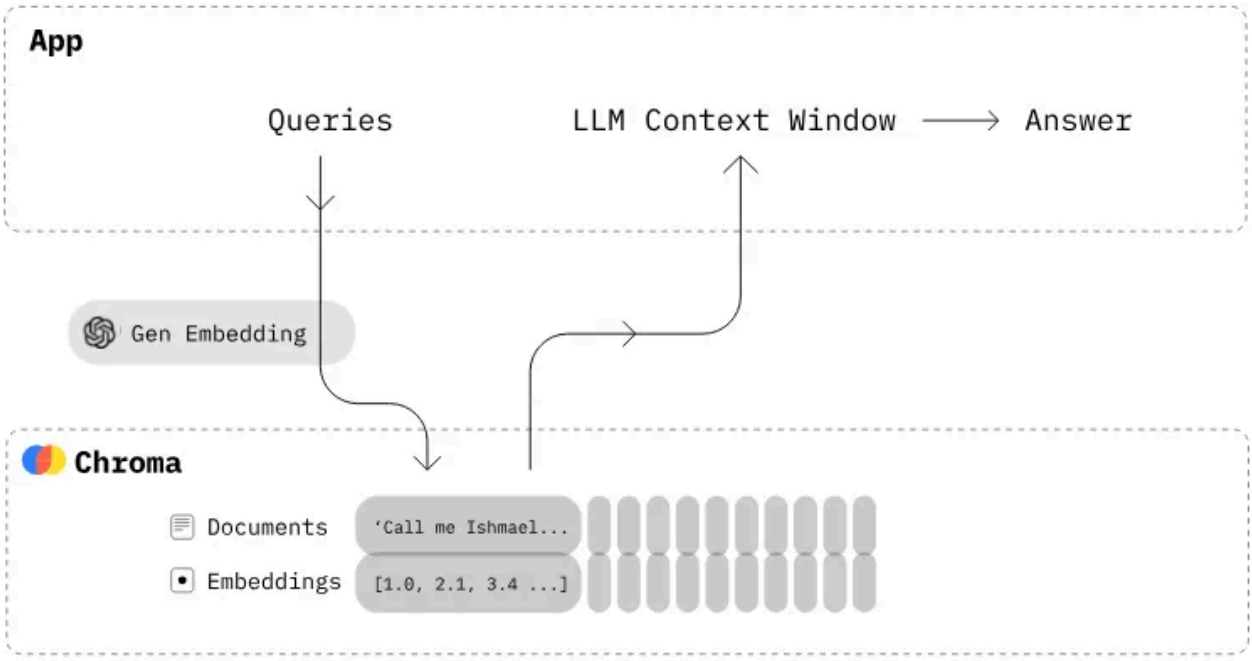
Create and Persist Vector Store:

```
# Create the vector store and persist it automatically
print("\n--- Creating vector store ---")
db = Chroma.from_documents(
    docs, embeddings, persist_directory=persistent_directory)
print("\n--- Finished creating vector store ---")
```

The code creates a vector store using `Chroma`, which stores the document embeddings and persists them in the specified directory, making them available for future queries and operations.

A quick look at the Chroma:

Chroma is the AI-native open-source vector database. Chroma makes it easy to build LLM apps by making knowledge, facts, and skills pluggable for LLMs.



Chroma gives you the tools to:

- store embeddings and their metadata
- embed documents and queries
- search embeddings

Chroma prioritizes:

- simplicity and developer productivity
- it also happens to be very quick

Else Statement for Existing Store:

```
else:
    print("Vector store already exists. No need to initialize.")
```

Here is the complete code for execution.

```
import os

from langchain.text_splitter import CharacterTextSplitter
```

```
from langchain_community.document_loaders import TextLoader
from langchain_community.vectorstores import Chroma
from langchain_openai import OpenAIEmbeddings

# Define the directory containing the text file and the persistent directory
current_dir = os.path.dirname(os.path.abspath(__file__))
file_path = os.path.join(current_dir, "books", "odyssey.txt")
persistent_directory = os.path.join(current_dir, "db", "chroma_db")

# Check if the Chroma vector store already exists
if not os.path.exists(persistent_directory):
    print("Persistent directory does not exist. Initializing vector store...")

    # Ensure the text file exists
    if not os.path.exists(file_path):
        raise FileNotFoundError(
            f"The file {file_path} does not exist. Please check the path."
        )

    # Read the text content from the file
    loader = TextLoader(file_path)
    documents = loader.load()

    # Split the document into chunks
    text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
    docs = text_splitter.split_documents(documents)

    # Display information about the split documents
    print("\n--- Document Chunks Information ---")
    print(f"Number of document chunks: {len(docs)}")
    print(f"Sample chunk:\n{docs[0].page_content}\n")

    # Create embeddings
    print("\n--- Creating embeddings ---")
    embeddings = OpenAIEmbeddings(
        model="text-embedding-3-small"
    )  # Update to a valid embedding model if needed
    print("\n--- Finished creating embeddings ---")

    # Create the vector store and persist it automatically
    print("\n--- Creating vector store ---")
    db = Chroma.from_documents(
        docs, embeddings, persist_directory=persistent_directory)
    print("\n--- Finished creating vector store ---")

else:
    print("Vector store already exists. No need to initialize.")
```

This code is a comprehensive workflow that prepares a text document for semantic search or other vector-based operations by transforming it into embeddings and storing them persistently for quick access.

What if I want to query a question from the document?

Let us assume that we have a question like

query = "Who is Odysseus' wife?"

The `query` variable contains the user's question, which will be used to search for relevant information in the vector store.

```
# Retrieve relevant documents based on the query
retriever = db.as_retriever(
    search_type="similarity_score_threshold",
    search_kwargs={"k": 3, "score_threshold": 0.4},
)
relevant_docs = retriever.invoke(query)
```

The `retriever` object is configured to search the vector store for documents similar to the query. It uses a similarity score threshold of 0.4, aiming to retrieve the top 3 (`k: 3`) documents that exceed this threshold in similarity to the query.

Display the Relevant Results:

```
# Display the relevant results with metadata
print("\n--- Relevant Documents ---")
for i, doc in enumerate(relevant_docs, 1):
    print(f"Document {i}:\n{doc.page_content}\n")
    if doc.metadata:
        print(f"Source: {doc.metadata.get('source', 'Unknown')}\n")
```

This loop iterates through the relevant documents retrieved by the query. It prints out the content of each document, along with any metadata associated with the document, such as the source. This helps provide context and traceability for the retrieved information.

The complete code with the query accessed is shown below.

```
import os
from dotenv import load_dotenv
from langchain_community.vectorstores import Chroma
from langchain_openai import OpenAIEmbeddings

# Load environment variables from .env
load_dotenv()

# Define the persistent directory
current_dir = os.path.dirname(os.path.abspath(__file__))
persistent_directory = os.path.join(current_dir, "db", "chroma_db")

# Define the embedding model
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")

# Load the existing vector store with the embedding function
db = Chroma(persist_directory=persistent_directory,
            embedding_function=embeddings)

# Define the user's question
query = "Who is Odysseus' wife?"

# Retrieve relevant documents based on the query
retriever = db.as_retriever(
    search_type="similarity_score_threshold",
    search_kwargs={"k": 3, "score_threshold": 0.4},
)
relevant_docs = retriever.invoke(query)

# Display the relevant results with metadata
print("\n--- Relevant Documents ---")
for i, doc in enumerate(relevant_docs, 1):
    print(f"Document {i}:\n{doc.page_content}\n")
    if doc.metadata:
        print(f"Source: {doc.metadata.get('source', 'Unknown')}\n")
```

This code effectively sets up a system to query a pre-existing vector database of text documents and retrieve relevant information based on semantic similarity. It leverages LangChain's vector store and embedding functionalities to perform efficient text searches. I hope with this you might have got a pretty decent overview on how the RAG is being used for certain business scenario.

References: