Google Drive Integration with RAG System

Goal:

Automatically download a `.txt` file from Google Drive, embed it using OpenAI, store it in Qdrant, and build a RAG system to generate answers and send them via email.

Libraries Used

Library	Purpose
`dotenv`	Loads OpenAI & Gmail credentials securely from `.env`
`langchain_core.documents.Document`	Wraps text chunks for ingestion
`langchain.text_splitter.RecursiveCharacterTextSplitter`	Splits long text into overlapping chunks
`qdrant_client`	Python client to communicate with Qdrant
`qdrant_client.http.models.VectorParams`, `Distance`	Defines embedding vector parameters
`langchain_community.vectorstores.Qdrant`	Interfaces LangChain with Qdrant
`langgraph.graph.StateGraph`	Builds LangGraph DAG flow
`langchain_core.runnables.RunnableLambda`	Turns Python functions into LangGraph nodes
`langchain_openai.ChatOpenAI`	Calls GPT-3.5-Turbo for answering
`langchain_openai.OpenAIEmbeddings`	Converts text chunks into embedding vectors
`yagmail`	Sends email replies using Gmail
`googleapiclient` + `google-auth`	Downloads `.txt` file from Google Drive
`os`	Loads environment variables
`io`	Handles byte streams for file download

Step-by-Step Breakdown

Step 1: Load Environment Variables

```
python
CopyEdit
from dotenv import load_dotenv
load_dotenv()
```

Loads:

- `OPENAI_API_KEY`
- `EMAIL_USER`
- `EMAIL_PASSWORD`

Step 2: Download `.txt` File from Google Drive

```
python
CopyEdit
def download_txt_from_drive(file_id, destination):
    ...
```

- Uses Google Drive API
- Authenticates via `credentials.json` + `token.json`
- Downloads a text file and saves locally

Step 3: Load and Split the Text File

```
python
CopyEdit
def load_txt_as_documents(txt_file):
    with open(txt_file, 'r', encoding='utf-8') as f:
        return f.read()

raw_text = load_txt_as_documents("document.txt")
```

```
splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=20
0)
texts = splitter.split_text(raw_text)
documents = [Document(page_content=chunk) for chunk in texts]
```

- Loads raw file content
- Splits into overlapping chunks for better semantic retrieval

Step 4: Convert Chunks to Embeddings

```
python
CopyEdit
embedding_function = OpenAIEmbeddings()
```

- Uses OpenAI's model to turn each chunk into a `1536-d` vector

Step 5: Initialize Qdrant

```
python
CopyEdit
qdrant_client = QdrantClient(host="localhost", port=6333)
qdrant_client.recreate_collection(
    collection_name="rag_txt_collection",
    vectors_config=VectorParams(size=1536, distance=Distance.COSINE),
)
```

- Connects to a **Docker-hosted** Qdrant instance
- Uses cosine similarity

Step 6: Upload Embeddings to Vector Store

```
python
CopyEdit
db = Qdrant(
    client=qdrant_client,
    collection_name="rag_txt_collection",
    embeddings=embedding_function
)
db.add_documents(documents)
```

- Stores chunks and vectors in Qdrant

Step 7: Define Shared State for LangGraph

```
python
CopyEdit
class GraphState(TypedDict):
    question: str
    context: str
    answer: str
    recipient: str
```

- Shared state across all nodes

Step 8: Retrieve Node

```
python
CopyEdit
def retrieve(state: GraphState):
    retriever = db.as_retriever()
    docs = retriever.invoke(state["question"])
    context = "\n\n".join([doc.page_content for doc in docs])
```

```
    return {"question": state["question"], "context": context}
```

- Finds top relevant chunks based on similarity

Step 9: Generate Answer Using GPT

```
python
CopyEdit
llm = ChatOpenAI(model="gpt-3.5-turbo")

def generate(state: GraphState):
    prompt = f"Answer the question using this context:\n\n{state['context']}\n\n
Question: {state['question']}"
    response = llm.invoke(prompt)
    return {
        "question": state["question"],
        "context": state["context"],
        "answer": response.content
    }
```

- Uses GPT-3.5 to produce an informed response

Step 10: Build the LangGraph Workflow

```
python
CopyEdit
graph = StateGraph(GraphState)
graph.add_node("retrieve", RunnableLambda(retrieve))
graph.add_node("generate", RunnableLambda(generate))
graph.add_node("send_email", RunnableLambda(send_email))

graph.set_entry_point("retrieve")
graph.add_edge("retrieve", "generate")
graph.add_edge("generate", "send_email")
```

```
graph.add_edge("send_email", END)

app = graph.compile()
```

- Constructs a 3-node LangGraph:
  `retrieve → generate → send_email`

Step 11: Run the Application

```
python
CopyEdit
inputs = {
    "question": "I have an issue setting a different delivery address up",
    "recipient": "shahzain0066@gmail.com"
}

result = app.invoke(inputs)
print("Final Answer:", result["answer"])
```

- Inputs a natural question and email
- Outputs GPT response and sends it automatically