Jira-Integrated Retrieval-Augmented Generation (RAG) System

Objective

This system:

- Reads a `.txt` document
- Splits and embeds it using OpenAI
- Stores the embeddings in Qdrant (a vector database)
- Uses LangGraph to:
    - Retrieve relevant context from the vector store
    - Generate answers using GPT-3.5
    - Create a **Jira ticket** automatically if the user question is issue-related

Libraries Used and Their Purpose

Library	Purpose
`os`	Access environment variables like API keys
`dotenv`	Load environment variables from `.env` file
`langchain_core.documents.Document`	Wrap text chunks into structured documents
`RecursiveCharacterTextSplitter`	Break long text into overlapping chunks
`OpenAIEmbeddings`	Generate vector embeddings from text using OpenAI
`ChatOpenAI`	Use GPT-3.5 to generate responses
`qdrant_client`	Communicate with Qdrant vector DB
`VectorParams`, `Distance`	Define vector size and similarity metric
`Qdrant` (LangChain wrapper)	Integrates Qdrant with LangChain APIs
`StateGraph`, `RunnableLambda`	Create a LangGraph pipeline (like a DAG)

Library	Purpose
`jira`	Connect to Jira to create issues/tasks automatically

Step-by-Step Code Breakdown

Step 1: Load Environment Variables

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

- Reads keys like `OPENAI_API_KEY`, `JIRA_URL`, `JIRA_EMAIL`, etc., from a `.env` file.
- Keeps sensitive data **secure and separate** from your code.

Step 2: Load and Chunk Text File

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

- Loads a plain `.txt` file from your local system.
- Encodes it in UTF-8 (standard text encoding).

```
python
CopyEdit
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
texts = text_splitter.split_text(raw_text)
```

```
documents = [Document(page_content=chunk) for chunk in texts]
```

- `RecursiveCharacterTextSplitter` breaks the raw text into chunks of 1000 characters, with a 200-character **overlap** for context continuity.
- Each chunk is wrapped in a `Document` class for structured processing.

Step 3: Create Embeddings with OpenAI

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

- Converts each chunk into a **1536-dimensional embedding vector** using OpenAI's embedding model (e.g., `text-embedding-ada-002` by default).
- These embeddings help in semantic search.

Step 4: Initialize and Populate Qdrant

```
python
CopyEdit
qdrant_client = QdrantClient(host="localhost", port=6333)
```

- Connects to a **Qdrant server running locally on Docker** at port 6333.

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

- Re-initializes the Qdrant collection with:
    - **Vector size = 1536**
    - **Distance metric = Cosine similarity** (great for text similarity)

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

- Wraps the raw Qdrant client with LangChain's `Qdrant` wrapper.
- Embeds and uploads all document chunks to the Qdrant collection.

Step 5: Define LangGraph Shared State

```
python
CopyEdit
from typing import TypedDict

class GraphState(TypedDict):
    question: str
    context: str
    answer: str
```

- This is a **state structure** passed between LangGraph nodes.
- Keeps track of:
    - User's `question`
    - Retrieved `context`

    - Generated `answer`

Step 6: Define the Retrieval Node

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

- Takes the user's question → finds similar documents using Qdrant
- Joins the most relevant chunks into one big `context` string

Step 7: Define the Generate Node

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

- Calls **OpenAI's GPT-3.5-Turbo** model to generate answers

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

    # Auto-create Jira ticket if issue-related
    trigger_keywords = ["issue", "problem", "bug", "error", "fail", "help", "suppo
rt"]
    if any(word in state["question"].lower() for word in trigger_keywords):
        create_jira_ticket(
            summary=f"User Support Request: {state['question']}",
            description=f"""Auto-created from RAG system.\n\nQuestion:\n{state
['question']}\n\nAnswer:\n{answer}"""
        )

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

- Generates a contextual answer.
- If the question seems like a support request (matches keywords), it calls the Jira ticket function.

Step 8: Jira Ticket Function

```
python
CopyEdit
from jira import JIRA

def create_jira_ticket(summary: str, description: str):
    options = {"server": os.getenv("JIRA_URL")}
    jira = JIRA(
        options,
        basic_auth=(os.getenv("JIRA_EMAIL"), os.getenv("JIRA_API_TOKEN"))
    )
    issue_dict = {
        'project': {'key': os.getenv("JIRA_PROJECT_KEY")},
        'summary': summary,
        'description': description,
        'issuetype': {'name': 'Task'},
    }
    issue = jira.create_issue(fields=issue_dict)
    print(f"Created Jira issue: {issue.key}")
    return issue.key
```

- Uses the `jira` Python library to:
  - Authenticate to your Jira project
  - Create a new **Task** issue with the question and generated answer

Step 9: Define LangGraph Flow

```
python
CopyEdit
graph = StateGraph(GraphState)
graph.add_node("retrieve", RunnableLambda(retrieve))
graph.add_node("generate", RunnableLambda(generate))
graph.set_entry_point("retrieve")
graph.add_edge("retrieve", "generate")
graph.add_edge("generate", END)
app = graph.compile()
```

- Builds a 2-node LangGraph:

  ```
  sql
  CopyEdit
  retrieve → generate → END
  ```

Step 10: Run the Pipeline

```
python
CopyEdit
inputs = {"question": "I have an issue setting a different delivery address up"}
result = app.invoke(inputs)
print(result['answer'])
```

- Inputs a natural language question
- Gets a GPT-based answer
- Creates a Jira support ticket if it contains a trigger keyword like "issue"

Summary

Step	Action
1. Load	Read `.txt` content
2. Split	Chunk text into overlapping segments
3. Embed	Convert to vector embeddings with OpenAI
4. Store	Push to Qdrant collection
5. Query	Retrieve relevant chunks with semantic search
6. Answer	Generate GPT-3.5 answer using retrieved context
7. Jira	Auto-create ticket if query is an issue