Jira Integration

▨ Status	Done

🧠 LangChain RAG System with Qdrant + JIRA Integration

An intelligent **Retrieval-Augmented Generation (RAG)** system that:

- Answers user questions based on custom document knowledge.
- Stores document vectors in **Qdrant**.
- Uses **OpenAI GPT-3.5 Turbo** to generate answers.
- Automatically creates **JIRA support tickets** for issue-related questions.

🧰 Libraries & Tools Overview

Library	Purpose
os	Interacts with system environment variables.
dotenv (`python-dotenv`)	Loads environment variables from `.env` file.
typing.TypedDict	Defines structured types for LangGraph state handling.
langchain_core.documents.Document	Represents individual chunks of text (documents) to be processed by LLMs.
langchain.text_splitter.RecursiveCharacterTextSplitter	Splits long documents into smaller overlapping chunks for effective retrieval.
langchain_community.vectorstores.Qdrant	LangChain's integration with the Qdrant vector store.

Library	Purpose
qdrant_client	Python client to interact with the local or remote Qdrant DB.
langchain_openai.OpenAIEmbeddings	Generates embeddings (vectors) from raw text using OpenAI API.
langchain_openai.ChatOpenAI	Wrapper around OpenAI's Chat models like GPT-3.5 Turbo.
langgraph.graph.StateGraph	Constructs a LangGraph (stateful computation flow).
langchain_core.runnables.RunnableLambda	Wraps a Python function into a runnable node in the graph.
jira.JIRA	Python library for creating and managing JIRA issues.

🔧 Environment Setup

🔷 Required `.env` Variables

Create a `.env` file in your project root:

```
env
CopyEdit
OPENAI_API_KEY=your_openai_api_key
JIRA_URL=https://your-domain.atlassian.net
JIRA_EMAIL=your_email@example.com
JIRA_API_TOKEN=your_jira_api_token
JIRA_PROJECT_KEY=PROJECTKEY
```

> ⚠️ All credentials are required for the system to work end-to-end.

📥 Data Loading and Preprocessing

✅ Load Text and Chunk it

```
python
CopyEdit
def load_txt_as_documents(txt_file: str):
    with open(txt_file, 'r', encoding='utf-8') as f:
        raw_text = f.read()
    splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=
200)
    chunks = splitter.split_text(raw_text)
    return [Document(page_content=chunk) for chunk in chunks]
```

- Input: `DOCUNMENT.txt`
- Output: List of `Document` objects (each ~1000 characters with overlap of 200)

📦 Vector Database Setup (Qdrant)

✅ Initialize Qdrant Vector DB

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

✅ Recreate Collection

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

> The size 1536 corresponds to OpenAI's embedding dimension (for text-embedding-ada-002).

✅ Store Embeddings

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

🧾 JIRA Ticket System

✅ Function: `create_jira_ticket`

Automatically creates a JIRA issue (type: Task) using provided summary and description:

```
python
CopyEdit
def create_jira_ticket(summary: str, description: str):
    ...
```

🔷 Trigger Conditions

If any of the following keywords are found in the question:

```
python
CopyEdit
["issue", "problem", "bug", "error", "fail", "help", "support"]
```

A support ticket is created in JIRA.

🧠 LangGraph Workflow

✅ State Definition

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

✅ Node 1: Retrieve Context

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

✅ Node 2: Generate Answer

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

Also triggers JIRA creation if the question contains issue-related keywords.

🔄 Graph Flow

```
text
CopyEdit
[ Entry → retrieve ] → [ generate ] → [ END ]
```

✅ Graph Compilation

```
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()
```

🚀 Running the Pipeline

✅ Entry Point

```
python
CopyEdit
if __name__ == "__main__":
    user_input = "I have an issue setting a different delivery address up"
    result = app.invoke({"question": user_input})
    print("\n🧠 Answer:\n", result["answer"])
```

If issue is detected:

```
yaml
CopyEdit
🧾 Created Jira issue: PROJECTKEY-123
```

💡 Customization Ideas

- Add support for `.pdf` or `.docx` files using LangChain loaders.
- Use metadata from documents (e.g., titles) for smarter ticket descriptions.
- Deploy via FastAPI or Streamlit for a web interface.
- Extend ticket creation to include priority/assignee from LLM output.

📦 Requirements

```
nginx
CopyEdit
langchain
langgraph
qdrant-client
openai
python-dotenv
```

```
jira
```

Install via:

```
bash
CopyEdit
pip install -r requirements.txt
```

🏁 Conclusion

This project is a complete demonstration of how to integrate **RAG**, **LLMs**, **vector search**, and **issue tracking systems** like JIRA. Ideal for:

- **Enterprise knowledge base bots**
- **Internal support automation**
- **Smart document QA systems**