RAG Agent with LangGraph - Comprehensive Code Explanation

Overview

This code implements a sophisticated RAG (Retrieval-Augmented Generation) system using LangGraph that can intelligently route questions between local vectorstore retrieval and web search, with multiple quality control mechanisms.

1. Imports and Setup

```python
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import WebBaseLoader
from langchain_community.vectorstores import Chroma
from langchain.prompts import PromptTemplate
from typing import List
from langgraph.graph import END, StateGraph, START
from typing_extensions import TypedDict
from langchain_community.chat_models import ChatOllama
from langchain_ollama.llms import OllamaLLM
from langchain_core.output_parsers import JsonOutputParser
from langchain_core.output_parsers import StrOutputParser
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain import hub
from pprint import pprint
from dotenv import load_dotenv
import os
load_dotenv()
from langchain_openai import OpenAIEmbeddings,ChatOpenAI
```

Purpose: Import all necessary components for:

- Document processing and vectorstore management
- LangGraph workflow orchestration
- Multiple LLM providers (Ollama, OpenAI)
- Web search capabilities
- Environment variable management

2. Document Loading and Processing

```python
urls = [
    "https://lilianweng.github.io/posts/2023-06-23-agent/",
    "https://lilianweng.github.io/posts/2023-03-15-prompt-engineering/",
    "https://lilianweng.github.io/posts/2023-10-25-adv-attack-llm/",
]

docs = [WebBaseLoader(url).load() for url in urls]
docs_list = [item for sublist in docs for item in sublist]

text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=250, chunk_overlap=0
)
doc_splits = text_splitter.split_documents(docs_list)
```

Purpose:

- Load three specific blog posts about LLM agents, prompt engineering, and adversarial attacks
- Flatten the nested list structure from multiple documents
- Split documents into 250-token chunks with no overlap for efficient retrieval

3. Vector Store Initialization

```python
def initialize_vectorstore(doc_splits):
    persist_directory = "chroma_db"

    if not os.path.exists(persist_directory):
        print("Chroma DB does not exist. Creating a new database...")
        vectorstore = Chroma.from_documents(
            documents=doc_splits,
            collection_name="ragChroma",
            embedding=OpenAIEmbeddings(),
            persist_directory=persist_directory
        )
        vectorstore.persist()
    else:
        print("Chroma DB exists. Loading from the existing database...")
        vectorstore = Chroma(persist_directory,OpenAIEmbeddings())

    return vectorstore.as_retriever()

retriever = initialize_vectorstore(doc_splits)
```

Purpose:

- Create or load a persistent Chroma vector database
- Use OpenAI embeddings for semantic similarity search
- Return a retriever interface for document querying

4. LLM Configuration

```python
llm = OllamaLLM(model=os.environ['MODEL'], format="json", temperature=0)
# llm = ChatOpenAI(model='gpt-4o-mini')  # Alternative option
```

Purpose: Configure the main LLM (using Ollama) with:

- JSON output format for structured responses
- Temperature 0 for deterministic outputs
- Model name from environment variable

5. Question Router

```python
prompt = PromptTemplate(
    template="""You are an expert at routing a user question to a vectorstore or web search. \n
    Use the vectorstore for questions on LLM  agents, prompt engineering, and adversarial attacks. \n
    You do not need to be stringent with the keywords in the question related to these topics. \n
    Otherwise, use web-search. Give a binary choice 'web_search' or 'vectorstore' based on the question. \n
    Return the a JSON with a single key 'datasource' and no premable or explanation. \n
    Question to route: {question}""",
    input_variables=["question"],
)

question_router = prompt | llm | JsonOutputParser()
```

Purpose:

- Intelligently route questions to appropriate data sources
- Uses the loaded documents' topics (agents, prompt engineering, adversarial attacks) for vectorstore
- Returns structured JSON: `{"datasource": "vectorstore"}` or `{"datasource": "web_search"}`

6. Document Relevance Grader

```python
prompt = PromptTemplate(
    template="""You are a grader assessing relevance of a retrieved document to a user question. \n
    Here is the retrieved document: \n\n {document} \n\n
    Here is the user question: {question} \n
    If the document contains keywords related to the user question, grade it as relevant. \n
    It does not need to be a stringent test. The goal is to filter out erroneous retrievals. \n
    Give a binary score 'yes' or 'no' score to indicate whether the document is relevant to the question. \n
    Provide the binary score as a JSON with a single key 'score' and no premable or explanation.""",
    input_variables=["question", "document"],
)

retrieval_grader = prompt | llm | JsonOutputParser()
```

Purpose:

- Quality control for retrieved documents
- Filters out irrelevant documents before generation
- Returns: `{"score": "yes"}` or `{"score": "no"}`

7. RAG Generation Chain

```python
prompt = hub.pull("rlm/rag-prompt")

def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)

rag_chain = prompt | llm | StrOutputParser()
```

Purpose:

- Uses a pre-built RAG prompt from LangChain Hub
- Formats multiple documents into a single context string
- Generates natural language responses

8. Hallucination Grader

```python
prompt = PromptTemplate(
    template="""You are a grader assessing whether an answer is grounded in / supported by a set of facts. \n
    Here are the facts:
    \n ------- \n
    {documents}
    \n ------- \n
    Here is the answer: {generation}
    Give a binary score 'yes' or 'no' score to indicate whether the answer is grounded in / supported by a set of facts. \n
    Provide the binary score as a JSON with a single key 'score' and no preamble or explanation.""",
    input_variables=["generation", "documents"],
)

hallucination_grader = prompt | llm | JsonOutputParser()
```

Purpose:

- Checks if generated answers are factually grounded in source documents
- Prevents hallucinations by validating against provided context
- Returns: `{"score": "yes"}` or `{"score": "no"}`

9. Answer Quality Grader

```python
prompt = PromptTemplate(
    template="""You are a grader assessing whether an answer is useful to resolve a question. \n
    Here is the answer:
    \n ------- \n
    {generation}
    \n ------- \n
    Here is the question: {question}
    Give a binary score 'yes' or 'no' to indicate whether the answer is useful to resolve a question. \n
    Provide the binary score as a JSON with a single key 'score' and no preamble or explanation.""",
    input_variables=["generation", "question"],
)

answer_grader = prompt | llm | JsonOutputParser()
```

Purpose:

- Evaluates if the generated answer actually addresses the user's question
- Quality control for answer relevance and completeness
- Returns: `{"score": "yes"}` or `{"score": "no"}`

10. Question Rewriter

```python
re_write_prompt = PromptTemplate(
    template="""You a question re-writer that converts an input question to a better version that is optimized \n
     for vectorstore retrieval. Look at the initial and formulate an improved question. \n
     Here is the initial question: \n\n {question}. Improved question with no preamble: \n """,
    input_variables=["generation", "question"],
)

question_rewriter = re_write_prompt | llm | StrOutputParser()
```

Purpose:

- Optimizes questions for better vectorstore retrieval
- Reformulates unclear or poorly structured questions
- Used when initial retrieval fails or produces poor results

11. Web Search Tool

```python
web_search_tool = TavilySearchResults(k=3)
```

Purpose:

- Provides web search capability using Tavily
- Returns top 3 search results for questions outside the vectorstore domain

12. Graph State Definition

```python
class GraphState(TypedDict):
    """
    Represents the state of our graph.

    Attributes:
        question: question
        generation: LLM generation
        documents: list of documents
    """

    question: str
    generation: str
    documents: List[str]
```

Purpose:

- Defines the state structure passed between nodes
- Tracks question, generated answer, and relevant documents throughout the workflow

13. Node Functions

Retrieve Node

```python
def retrieve(state):
    print("---RETRIEVE---")
    question = state["question"]
    documents = retriever.get_relevant_documents(question)
    return {"documents": documents, "question": question}
```

Purpose: Fetches relevant documents from the vectorstore based on the question

Generate Node

python

```python
def generate(state):
    print("---GENERATE---")
    question = state["question"]
    documents = state["documents"]
    generation = rag_chain.invoke({"context": documents, "question": question})
    return {"documents": documents, "question": question, "generation": generation}
```

Purpose: Generates answers using the RAG chain with retrieved documents as context

Grade Documents Node

python

```python
def grade_documents(state):
    print("---CHECK DOCUMENT RELEVANCE TO QUESTION---")
    question = state["question"]
    documents = state["documents"]

    filtered_docs = []
    for d in documents:
        score = retrieval_grader.invoke(
            {"question": question, "document": d.page_content}
        )
        grade = score["score"]
        if grade == "yes":
            print("---GRADE: DOCUMENT RELEVANT---")
            filtered_docs.append(d)
        else:
            print("---GRADE: DOCUMENT NOT RELEVANT---")
            continue
    return {"documents": filtered_docs, "question": question}
```

Purpose: Filters out irrelevant documents before generation

Transform Query Node

python

```python
def transform_query(state):
    print("---TRANSFORM QUERY---")
    question = state["question"]
    documents = state["documents"]
    better_question = question_rewriter.invoke({"question": question})
    return {"documents": documents, "question": better_question}
```

Purpose: Rewrites questions for better retrieval results

Web Search Node

python

```python
def web_search(state):
    print("---WEB SEARCH---")
    question = state["question"]
    docs = web_search_tool.invoke({"query": question})
    web_results = "\n".join([d["content"] for d in docs])
    web_results = Document(page_content=web_results)
    return {"documents": web_results, "question": question}
```

Purpose: Performs web search and formats results as documents

14. Edge Functions (Decision Logic)

Route Question

python

```python
def route_question(state):
    print("---ROUTE QUESTION---")
    question = state["question"]
    source = question_router.invoke({"question": question})
    if source["datasource"] == "web_search":
        print("---ROUTE QUESTION TO WEB SEARCH---")
        return "web_search"
    elif source["datasource"] == "vectorstore":
        print("---ROUTE QUESTION TO RAG---")
        return "vectorstore"
```

Purpose: Initial routing decision between vectorstore and web search

Decide to Generate

```python
def decide_to_generate(state):
    print("---ASSESS GRADED DOCUMENTS---")
    filtered_documents = state["documents"]

    if not filtered_documents:
        print("---DECISION: ALL DOCUMENTS ARE NOT RELEVANT TO QUESTION, TRANSFORM QUERY---")
        return "transform_query"
    else:
        print("---DECISION: GENERATE---")
        return "generate"
```

Purpose: Decides whether to generate an answer or rewrite the question based on document relevance

Grade Generation

python

```python
def grade_generation_v_documents_and_question(state):
    print("---CHECK HALLUCINATIONS---")
    question = state["question"]
    documents = state["documents"]
    generation = state["generation"]

    score = hallucination_grader.invoke(
        {"documents": documents, "generation": generation}
    )
    grade = score["score"]

    if grade == "yes":
        print("---DECISION: GENERATION IS GROUNDED IN DOCUMENTS---")
        score = answer_grader.invoke({"question": question, "generation": generation})
        grade = score["score"]
        if grade == "yes":
            print("---DECISION: GENERATION ADDRESSES QUESTION---")
            return "useful"
        else:
            print("---DECISION: GENERATION DOES NOT ADDRESS QUESTION---")
            return "not useful"
    else:
        print("---DECISION: GENERATION IS NOT GROUNDED IN DOCUMENTS, RE-TRY---")
        return "not supported"
```

Purpose: Multi-step quality control:

1. Check for hallucinations
2. Check if answer addresses the question
3. Route to appropriate next step

15. Workflow Construction

python

```python
workflow = StateGraph(GraphState)

# Define the nodes
workflow.add_node("web_search", web_search)
workflow.add_node("retrieve", retrieve)
workflow.add_node("grade_documents", grade_documents)
workflow.add_node("generate", generate)
workflow.add_node("transform_query", transform_query)

# Build graph
workflow.add_conditional_edges(
    START,
    route_question,
    {
        "web_search": "web_search",
        "vectorstore": "retrieve",
    },
)
workflow.add_edge("web_search", "generate")
workflow.add_edge("retrieve", "grade_documents")
workflow.add_conditional_edges(
    "grade_documents",
    decide_to_generate,
    {
        "transform_query": "transform_query",
        "generate": "generate",
    },
)
workflow.add_edge("transform_query", "retrieve")
workflow.add_conditional_edges(
    "generate",
    grade_generation_v_documents_and_question,
    {
        "not supported": "generate",
        "useful": END,
        "not useful": "transform_query",
    },
)

app = workflow.compile()
```

Purpose: Constructs the complete workflow graph with all nodes and decision points

Overall Agent Design

This RAG agent implements a sophisticated **multi-step reasoning and quality control system**:

1. Intelligent Routing

- Questions are initially routed to either vectorstore or web search based on topic relevance
- The system knows its domain expertise (LLM agents, prompt engineering, adversarial attacks)

2. Multi-Layer Quality Control

- **Document Relevance**: Filters out irrelevant retrieved documents
- **Hallucination Detection**: Ensures answers are grounded in source material
- **Answer Quality**: Verifies answers actually address the question

3. Adaptive Retrieval

- If documents aren't relevant, the system rewrites the question and tries again
- If answers aren't grounded or useful, it can retry generation or rewrite the question

4. Fallback Mechanisms

- Web search for out-of-domain questions
- Question rewriting for improved retrieval
- Multiple retry loops for quality improvement

5. Workflow Path Examples

Successful RAG Path: `START → route_question → retrieve → grade_documents → generate → quality_check → END`

Question Rewriting Path: `START → route_question → retrieve → grade_documents → transform_query → retrieve → generate → END`

Web Search Path: `START → route_question → web_search → generate → quality_check → END`

Retry Path: `START → route_question → retrieve → grade_documents → generate → quality_check → transform_query → retrieve → generate → END`

This design ensures high-quality, reliable answers through multiple validation steps and adaptive behavior when initial attempts fail.