

OFFICIAL COURSE STUDY GUIDE

LangChain & LangGraph: RAG to Multi-Agent

Go from RAG pipelines to production multi-agent systems.

AI & AGENTIC ENGINEERING · ADVANCED · 2 MODULES · 7 HR 37 MIN

Course by Pradip Nichite · Curated and published by MegaCourse

MEGACOURSE.APP

LangChain & LangGraph: RAG to Multi-Agent: The Complete Study Guide

2026 Edition

Published by MegaCourse · megacourse.app

Learn Anything. Achieve Everything.

MegaCourse curates the world's best free courses into structured learning paths with study guides, practice material, and certificates. This guide accompanies the MegaCourse course "LangChain & LangGraph: RAG to Multi-Agent", taught by Pradip Nichite. It is designed to be read alongside the video lessons and to serve as a standalone reference for review.

Study guide text copyright © 2026 MegaCourse. Course video content remains the property of its creators. You may print this guide for personal study. Redistribution or resale is not permitted.

Enroll, track progress, and earn your certificate at

megacourse.app/courses/langchain-and-langgraph-rag-to-multi-agent

Contents

<i>About This Course</i>	<i>4</i>
CHAPTER 1 RAG Pipelines and Agent Fundamentals	6
CHAPTER 2 Production RAG Agents and Long-Term Memory	13
<i>Glossary</i>	<i>19</i>
<i>Answer Key</i>	<i>21</i>
<i>Where to Go Next</i>	<i>23</i>

INTRODUCTION

About This Course

This study guide supports the MegaCourse "LangChain & LangGraph: RAG to Multi-Agent" course. This course teaches you to build sophisticated AI applications, moving from basic RAG pipelines to complex multi-agent systems. You will gain practical skills for developing production-ready LLM applications.

The field of AI and agentic engineering is rapidly evolving. Understanding how to orchestrate large language models (LLMs) into robust applications is a valuable skill. This course provides a clear path to mastering these techniques using LangChain and LangGraph.

MegaCourse carefully curates the best free video content and adds structure, learning outcomes, and study materials. This guide complements the video lessons, offering a structured approach to learning. It helps you consolidate concepts and practice your skills effectively.

Course Facts

FORMAT	2 modules · 11 lessons
VIDEO INSTRUCTION	About 7 hr 37 min
LEVEL	Advanced
CATEGORY	AI & Agentic Engineering
INSTRUCTOR	Pradip Nichite
CERTIFICATE	Included with course completion on MegaCourse
COURSE HOME	megacourse.app/courses/langchain-and-langgraph-rag-to-multi-agent

What You Will Learn

- Build advanced RAG systems using LangChain.
- Orchestrate complex multi-agent workflows with LangGraph.
- Incorporate tools, memory, and state management into agents.
- Ship production-ready large language model applications.

Who This Course Is For

This course is designed for developers and engineers who want to build advanced applications with large language models. If you aim to create robust RAG systems or orchestrate multi-agent workflows, this course will serve your goals.

Before starting, you should have a solid understanding of Python. Basic knowledge of large language models and their concepts will also be beneficial. This foundation ensures you can fully engage with the advanced topics covered.

Before You Start

- Possess a working knowledge of Python.
- Understand basic concepts of large language models.

How to Use This Guide

This study guide works alongside the video lessons on megacourse.app. We recommend watching each video lesson first to grasp the core concepts. Then, use this guide to review the material, work through examples, and solidify your understanding. Consistent practice is key to mastering these topics.

Pace yourself according to your schedule. The course has 2 modules and 11 lessons, totaling about 7 hours and 37 minutes of instruction. Break down your study sessions into manageable chunks. Focus on one lesson at a time before moving to the next.

This guide includes summaries, exercises, and a glossary. Use the exercises to test your comprehension and apply what you have learned. Refer to the glossary for quick definitions of technical terms. These tools are here to reinforce your learning and prepare you for building your own projects.

CHAPTER 1

RAG Pipelines and Agent Fundamentals

This chapter provides a foundational understanding of Retrieval Augmented Generation (RAG) systems and introduces the concept of agentic architectures using LangChain. You will learn to construct robust RAG pipelines, from basic document processing to integrating with vector databases for semantic search.

The chapter also covers the LangChain Expression Language (LCEL) for building efficient and composable chains. Finally, it explores how LangChain agents can interact with external tools, such as SQL databases, and offers an initial look into LangGraph for orchestrating complex, multi-agent workflows.

Building Foundational RAG Systems

Covers: Langchain RAG Course: From Basics to Production-Ready RAG Chatbot · Using Langchain and Open Source Vector DB Chroma for Semantic Search with OpenAI's LLM | Code · Chatbot Answering from Your Own Knowledge Base: Langchain, ChatGPT, Pinecone, and Streamlit: | Code · Building a Document-based Question Answering System with LangChain, Pinecone, and LLMs like GPT-4.

RAG systems enhance large language models (LLMs) by providing them with external, relevant information. This approach mitigates issues like hallucination and allows LLMs to answer questions based on specific, up-to-date knowledge bases. A typical RAG pipeline involves several key steps. First, documents are loaded from various sources. These documents are then split into smaller, manageable chunks. Text splitting is crucial; chunks must be small enough to fit within an LLM's context window, yet large enough to retain semantic meaning. Overlapping chunks can help preserve context across splits.

Next, these text chunks are transformed into numerical representations called embeddings using an embedding model. These embeddings capture the semantic meaning of the text. The embeddings are then stored in a vector database, such as Chroma or Pinecone. Vector databases are optimized for efficient similarity search. When a user asks a question, the question itself is also converted into an embedding. This query embedding is used to search the vector database for the most semantically similar document chunks. This process is known as retrieval.

The retrieved chunks, along with the original user query, are then passed to an LLM. The LLM uses this augmented context to generate a more informed and accurate answer. This combination of retrieval and generation is what defines RAG.

Here is a basic example of setting up a RAG chain with LangChain:

```
from langchain_community.document_loaders import TextLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.embeddings import OpenAIEmbeddings
from langchain_community.vectorstores import Chroma
from langchain_openai import ChatOpenAI
```

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_core.output_parsers import StrOutputParser

# 1. Load Documents
loader = TextLoader("your_document.txt")
docs = loader.load()

# 2. Split Documents
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
splits = text_splitter.split_documents(docs)

# 3. Create Embeddings and Store in Vector DB
embeddings = OpenAIEmbeddings()
vectorstore = Chroma.from_documents(documents=splits, embedding=embeddings)
retriever = vectorstore.as_retriever()

# 4. Define LLM and Prompt
llm = ChatOpenAI(model="gpt-4o")
prompt = ChatPromptTemplate.from_messages([
    ("system", "Answer the question based only on the provided context: {context}"),
    ("user", "{question}")
])

# 5. Build the RAG Chain
rag_chain = (
    {"context": retriever, "question": RunnablePassthrough()}
    | prompt
    | llm
    | StrOutputParser()
)

# Example Usage
# response = rag_chain.invoke("What is the main topic of the document?")
# print(response)
```

Common mistakes in RAG implementation include using an inappropriate chunk size, which can lead to fragmented context or irrelevant information. Another error is selecting a vector store that does not scale with your data volume or query load. Ensuring your embedding model aligns with your domain and data type is also important for retrieval quality. For practice, try creating a RAG system using a local text file. Experiment with different `chunk_size` and `chunk_overlap` values. Observe how these changes affect the quality of the retrieved context and the final answer generated by the LLM. You can also swap out Chroma for Pinecone by initializing `PineconeVectorStore.from_documents` instead. This provides direct experience with different vector database integrations.

LangChain Expression Language (LCEL) for Chain Construction

Covers: LangChain Expression Language (LCEL) | Langchain Tutorial | Code

The LangChain Expression Language, or LCEL, offers a powerful and declarative way to compose complex chains in LangChain. It provides a flexible interface for chaining components such as prompts, language

models, retrievers, and output parsers. LCEL is built on the concept of `Runnable` objects, which are callable components that can be combined using the pipe operator (`|`). This approach offers several significant advantages. It enables streaming intermediate results, supports asynchronous execution, allows for parallel processing of components, and facilitates the implementation of fallbacks. These features are crucial for building production-ready LLM applications that are performant and resilient.

LCEL makes chains explicit and easy to understand. A chain defined with LCEL clearly shows the flow of data from one component to the next. For instance, in a RAG pipeline, you can define how the user's question first goes to a retriever, then the retrieved context and question go to a prompt, which then feeds into an LLM, and finally an output parser processes the LLM's response.

Key LCEL components include `RunnablePassthrough`, which passes its input directly to the next component or allows injecting specific keys into a dictionary. `RunnableLambda` allows you to wrap any Python function into a runnable component, providing flexibility for custom logic.

Consider this example of a simple LCEL chain:

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough

# Define the LLM
llm = ChatOpenAI(model="gpt-4o", temperature=0)

# Define a prompt template
prompt = ChatPromptTemplate.from_template("Tell me a short story about {topic}.")

# Define an output parser
parser = StrOutputParser()

# Construct the chain using LCEL
story_chain = (
    {"topic": RunnablePassthrough()} # Input is passed as 'topic'
    | prompt
    | llm
    | parser
)

# Invoke the chain
# print(story_chain.invoke("a brave knight"))
```

In a RAG context, LCEL allows you to build sophisticated data flows. For example, you might have a chain where the user's query first goes to a retriever, but also a separate branch where the query is used to check a cache. The results from both branches could then be combined. LCEL simplifies the construction of such branching and merging logic. It promotes modularity, allowing you to define and test individual components before composing them into larger systems. This modularity is particularly beneficial for debugging and maintaining complex RAG pipelines. When an issue arises, you can isolate the problematic component more easily.

To practice with LCEL, take the RAG chain from the previous section. Refactor the prompt, LLM, and output parser into a standalone LCEL chain that takes a dictionary with `context` and `question` as input. Then, integrate this new chain with the retriever using the `RunnablePassthrough` component to ensure the question is passed through correctly. This exercise will solidify your understanding of how to build and combine runnable components effectively.

Natural Language to SQL with LangChain Agents

Covers: Mastering Natural Language to SQL with LangChain and LangSmith | NL2SQL | With Code · LangChain, SQL Agents & OpenAI LLMs: Query Database Using Natural Language | Code

LangChain agents extend the capabilities of LLMs by giving them access to external tools. While a basic LLM can generate text, an agent can decide which tools to use and in what order to achieve a specific goal. This allows LLMs to perform actions beyond text generation, such as querying databases, searching the web, or interacting with APIs. The agent's core loop involves observation, thought, and action. It observes the current state, thinks about the next best step, and then takes an action using an available tool.

Natural Language to SQL (NL2SQL) is a prime application for LangChain agents. Instead of writing complex SQL queries manually, users can express their data requests in plain English. The agent then translates these requests into executable SQL queries, runs them against a database, and interprets the results. This significantly lowers the barrier for data access and analysis for non-technical users.

A LangChain SQL agent typically consists of a language model, a set of SQL-specific tools, and a prompt that guides the LLM in its decision-making process. The tools allow the agent to inspect the database schema, execute SQL queries, and retrieve table information.

Here is a conceptual outline of how a SQL agent operates:

1. **User Query:** A user asks a question like "Show me the total sales for each product category last month."
2. **Agent Thought:** The LLM, acting as the agent's brain, receives the query. It reasons about which tools it needs to use. It might first use a tool to list available tables and their schemas to understand the database structure.
3. **Tool Use (Schema Inspection):** The agent executes a tool to get schema details.
4. **Agent Thought (Query Generation):** Based on the schema and the user's query, the LLM constructs a SQL query.
5. **Tool Use (SQL Execution):** The agent executes the generated SQL query using a database execution tool.
6. **Agent Thought (Result Interpretation):** The LLM receives the SQL query results. It then interprets these results and formats them into a human-readable answer.
7. **Final Answer:** The agent presents the answer to the user.

Setting up a SQL agent involves connecting to a database, often using `SQLAlchemy`, and then creating a `SQLDatabaseToolkit` which provides the necessary tools for the agent.

```
from langchain_community.utilities import SQLDatabase
from langchain_community.agent_toolkits import SQLDatabaseToolkit
from langchain_openai import ChatOpenAI
from langchain.agents import create_sql_agent

# 1. Establish database connection
# For SQLite, use: sqlite:///your_database.db
# For other databases, adjust the connection string.
db = SQLDatabase.from_uri("sqlite:///Chinook.db")

# 2. Initialize the LLM
llm = ChatOpenAI(model="gpt-4o", temperature=0)

# 3. Create a SQL Toolkit
toolkit = SQLDatabaseToolkit(db=db, llm=llm)

# 4. Create the SQL Agent
agent_executor = create_sql_agent(llm=llm, toolkit=toolkit, verbose=True)

# 5. Invoke the agent with a natural language query
# response = agent_executor.invoke({"input": "List the total number of tracks for each genre."})
# print(response)
```

Common challenges with NL2SQL agents include ensuring the agent accurately understands complex database schemas, handling ambiguous natural language queries, and preventing the generation of unsafe or inefficient SQL. Security is also a concern; agents should be carefully constrained to prevent unauthorized data access or malicious operations. For practice, set up an in-memory SQLite database with a few tables and some sample data. Then, use a LangChain SQL agent to query this database using natural language. Try asking questions that require joins or aggregations to test the agent's reasoning capabilities.

Introduction to LangGraph for Multi-Agent Workflows

Covers: LangGraph RAG Agent Tutorial | Basics to Advanced Multi-Agent AI Chatbot | With Code

While individual LangChain agents can perform complex tasks, many real-world problems benefit from coordinating multiple agents or requiring more intricate control flow. This is where LangGraph becomes indispensable. LangGraph is a library built on LangChain that allows you to construct stateful, multi-actor applications with LLMs. It provides a framework for defining workflows as graphs, where nodes represent actions or agents, and edges represent transitions between states. This explicit graph structure gives developers fine-grained control over the execution flow, enabling complex patterns like conditional routing, cycles, and human-in-the-loop interactions.

The core concept in LangGraph is the `StateGraph`. You define the state that your graph will manage, and then you add nodes and edges to define how that state evolves. Each node in the graph performs a specific task, such as calling an LLM, using a tool, or performing a custom computation. Edges define the transitions between these nodes. Conditional edges allow the graph to make decisions based on the current state or the output of a node, routing the execution path dynamically.

Consider a sophisticated RAG system. Instead of a single linear chain, you might want an initial agent to decide if a query even requires retrieval. If it is a simple factual question, the LLM might answer directly. If it requires external knowledge, a retrieval agent is invoked. After retrieval, another agent might re-rank documents or refine the query. This kind of dynamic decision-making and iterative process is difficult to implement with simple chains but is a natural fit for LangGraph.

Here is a basic illustration of how you might define a simple graph:

```
from typing import TypedDict, Annotated, List
from langchain_core.messages import BaseMessage
from langgraph.graph import StateGraph, START, END

# Define the state for our graph
class AgentState(TypedDict):
    messages: Annotated[List[BaseMessage], lambda x, y: x + y]
    # Add other state variables as needed

# Define a node function
def call_llm(state: AgentState):
    # This node would typically call an LLM
    print("Calling LLM...")
    return {"messages": ["LLM response message"]}

# Define another node function
def call_tool(state: AgentState):
    # This node would typically call a tool
    print("Calling Tool...")
    return {"messages": ["Tool output message"]}

# Build the graph
workflow = StateGraph(AgentState)

# Add nodes
workflow.add_node("llm_node", call_llm)
workflow.add_node("tool_node", call_tool)

# Add edges (simple sequence for demonstration)
workflow.add_edge(START, "llm_node")
workflow.add_edge("llm_node", "tool_node")
workflow.add_edge("tool_node", END)

# Compile the graph
app = workflow.compile()

# Example usage
# final_state = app.invoke({"messages": ["Initial user query"]})
# print(final_state)
```

LangGraph's explicit state management and control flow are its primary strengths. It allows you to model complex decision-making processes, implement loops for iterative refinement, and even incorporate human feedback at specific points in the workflow. This structured approach is essential for building resilient and scalable multi-agent systems that can handle a wide range of tasks and adapt to dynamic

conditions. Understanding LangGraph lays the groundwork for orchestrating the advanced multi-agent systems covered in later modules. For practice, try to expand the simple graph example. Add a conditional edge that decides whether to call the `tool_node` based on the content of the `llm_node`'s output. This will help you grasp how to introduce dynamic routing into your agent workflows.

KEY TAKEAWAYS

- RAG systems combine external knowledge retrieval with LLM generation to produce more accurate and grounded answers.
- A RAG pipeline involves document loading, splitting, embedding, vector storage, retrieval, and LLM synthesis.
- LangChain Expression Language (LCEL) provides a declarative and efficient way to compose complex, production-ready LLM chains.
- LangChain agents extend LLMs with tools, enabling them to perform actions like querying databases or interacting with external APIs.
- Natural Language to SQL (NL2SQL) is a practical application of agents, allowing users to query databases using plain English.
- LangGraph is a framework for building stateful, multi-actor LLM applications with explicit control flow, crucial for complex agentic workflows.

CHECK YOUR UNDERSTANDING

Answer these before moving on. Model answers are in the Answer Key at the back of this guide.

1. Describe the core components of a RAG system and their respective roles.
2. What are two key benefits of using LangChain Expression Language (LCEL) for building LLM applications?
3. Explain how a LangChain agent differs from a simple LangChain chain.
4. What is the primary purpose of a SQL agent in LangChain?
5. How does LangGraph enhance the capabilities of single LangChain agents?

CHAPTER 2

Production RAG Agents and Long-Term Memory

This chapter guides you through the essential steps for deploying advanced Retrieval-Augmented Generation (RAG) agents in a production environment. You will learn how to host a vector database, specifically Chroma DB, on a cloud server like AWS EC2, ensuring your RAG system has a robust and scalable knowledge base.

The chapter then moves into building a production-ready RAG agent using LangGraph and FastAPI. We cover how to manage conversation sessions, maintain history, and integrate with your hosted vector database. Finally, you will explore how to add sophisticated long-term memory capabilities to your AI agents using Memo, allowing them to retain information and context across extended interactions.

Hosting Chroma DB on AWS EC2 for Production RAG

Covers: Hosting Chroma DB on AWS EC2: Server Setup and Client Connection Tutorial

Deploying a RAG system for production requires a robust and accessible vector database. While local installations are convenient for development, a cloud-hosted solution provides scalability, reliability, and remote access for your applications. This section details how to set up Chroma DB on an AWS EC2 instance.

Setting Up Your EC2 Instance

First, you need an EC2 instance. Log into the AWS Management Console and navigate to EC2. Launch a new instance. Choose an appropriate Amazon Machine Image (AMI), such as Ubuntu Server 22.04 LTS. Select an instance type that meets your performance needs, for example, a `t2.medium` or `t3.medium` for initial deployments. Configure instance details, storage, and tags as needed. The most critical step for database hosting is network configuration.

Configuring Security Groups

Security groups act as virtual firewalls. For Chroma DB, you need to allow inbound traffic on specific ports. Create a new security group or modify an existing one. Add an inbound rule for port 22 (SSH) from your IP address or a restricted range to allow secure shell access. Crucially, add another inbound rule for port 8000, which is the default port Chroma DB uses for its HTTP server. Set the source for port 8000 to `Anywhere` (`0.0.0.0/0`) if your client application will connect from various locations, or restrict it to specific IP ranges for tighter security. Review your settings and launch the instance.

Installing and Running Chroma DB on EC2

Once your EC2 instance is running, connect to it via SSH using the provided `.pem` key. Update your package lists and install Docker, which is a common and recommended way to run services like Chroma DB.

```
sudo apt update
sudo apt install docker.io -y
sudo systemctl start docker
sudo systemctl enable docker
```

After Docker is installed, you can pull the Chroma DB Docker image and run it. The `-p 8000:8000` flag maps the container's port 8000 to the host's port 8000, making it accessible from outside the instance. The `--name chromadb` assigns a memorable name to the container.

```
sudo docker pull chromadb/chroma
sudo docker run -d -p 8000:8000 --name chromadb chromadb/chroma
```

Verify that Chroma DB is running by checking Docker processes:

```
sudo docker ps
```

Client Connection to Remote Chroma DB

With Chroma DB running on your EC2 instance, your local LangChain application can connect to it. You will need the public IP address or public DNS name of your EC2 instance. In your Python code, use `chromadb.HttpClient` and specify the host and port.

```
import chromadb
from langchain_community.vectorstores import Chroma
from langchain_community.embeddings import OpenAIEmbeddings

# Replace with your EC2 instance's public IP or DNS name
CHROMA_HOST = "your_ec2_public_ip"
CHROMA_PORT = 8000

# Initialize the Chroma client
client = chromadb.HttpClient(host=CHROMA_HOST, port=CHROMA_PORT)

# Example: Connect to a collection or create one
collection_name = "my_rag_collection"

# Ensure the collection exists or create it
try:
    collection = client.get_or_create_collection(name=collection_name)
except Exception as e:
    print(f"Error getting/creating collection: {e}")
    # Handle error, e.g., ensure Chroma DB is running and accessible
```

```
# You can then use this client with LangChain's Chroma integration
# For example, to load an existing vector store or create a new one
vectorstore = Chroma(
    client=client,
    collection_name=collection_name,
    embedding_function=OpenAIEmbeddings()
)

print(f"Successfully connected to Chroma DB at {CHROMA_HOST}:{CHROMA_PORT}")
```

Common connection issues include incorrect security group rules, the Chroma DB container not running, or an incorrect IP address. Always double-check your security group inbound rules and ensure the Docker container is active and accessible on port 8000.

Building Production RAG Agents with FastAPI and Long-Term Memory

Covers: Integrating LangGraph RAG Agent with FastAPI | Production Setup with Sessions, History, Vector DB · LangGraph + Mem0
Integration: Add Long-Term Memory to Your AI Agents

Moving a LangGraph RAG agent from a script to a production environment requires an API layer, session management, and often, long-term memory. FastAPI is an excellent choice for building performant and easy-to-use APIs for your agents.

Integrating LangGraph with FastAPI

FastAPI allows you to expose your LangGraph agent as a web service. You define endpoints that receive user input, process it through the agent, and return a response. The agent's state, including conversation history, needs careful management across requests.

```
from fastapi import FastAPI, Request
from pydantic import BaseModel
from typing import List, Dict, Any
from langchain_core.messages import BaseMessage, HumanMessage, AIMessage
from langchain_community.chat_message_histories import ChatMessageHistory
from langchain_core.runnables.history import RunnableWithMessageHistory
# Assume you have your LangGraph agent defined as 'rag_agent_graph'
# and a remote Chroma DB client 'vectorstore' from the previous section.

app = FastAPI()

# In-memory storage for session history (for demonstration)
# In production, use Redis, a database, or Mem0 for persistence.
session_history: Dict[str, ChatMessageHistory] = {}

class ChatInput(BaseModel):
    session_id: str
    message: str

@app.post("/chat")
async def chat_endpoint(input: ChatInput):
```

```

user_message = input.message

if session_id not in session_history:
    session_history[session_id] = ChatMessageHistory()

# Create a runnable with history
with_message_history = RunnableWithMessageHistory(
    rag_agent_graph, # Your pre-defined LangGraph agent
    lambda session_id: session_history[session_id],
    input_messages_key="messages",
    history_messages_key="history"
)

# Invoke the agent with the current message and session history
response = with_message_history.invoke(
    {"messages": [HumanMessage(content=user_message)]},
    config={"configurable": {"session_id": session_id}}
)

# The response from LangGraph will be a list of messages, get the last one
agent_response = response["messages"][-1].content if response and response["messages"]
else "No response."

return {"session_id": session_id, "response": agent_response}

```

To run this FastAPI application, save it as `main.py` and use Uvicorn:

```
uvicorn main:app --reload --host 0.0.0.0 --port 8001
```

This setup demonstrates session management using an in-memory dictionary. For real production, `session_history` should be replaced with a persistent store like Redis, a database, or Memo, which we discuss next. The `RunnableWithMessageHistory` wrapper automatically manages adding messages to the history before invoking the agent and retrieving it for context.

Adding Long-Term Memory with Mem0

While session history maintains context for a single conversation, agents often benefit from long-term memory that persists across sessions and provides deeper contextual understanding. Memo is a memory layer designed to give AI agents persistent and context-aware memory. It allows agents to remember facts, preferences, and past interactions over extended periods.

Integrating Memo with your LangGraph agent provides a more sophisticated memory solution than simple `ChatMessageHistory`. Memo can store and retrieve information based on relevance, providing a richer context to the agent's decisions and responses.

Mem0 Integration Steps

1. **Install Memo:** Add the Memo library to your project.

```
pip install mem0
```

2. Initialize Memo: Create a Memo client. You might need an API key if using a hosted Memo service.

```
from mem0 import Mem0
m0 = Mem0(api_key="YOUR_MEM0_API_KEY") # Or configure for local/self-hosted
```

3. Integrate with LangGraph: You can integrate Memo into your LangGraph agent in several ways. A common pattern is to add a node to your graph that retrieves relevant memories before the LLM generates a response, or to store new memories after a response. Memo can also be used as a tool the agent decides to use.

When the agent receives a user query, you can first query Memo for relevant past interactions or facts. This retrieved information can then be injected into the prompt provided to the LLM, giving the agent a more informed context.

```
# Example of retrieving memories before invoking the LLM
def retrieve_memories(state: Dict[str, Any]) -> Dict[str, Any]:
    user_message = state["messages"][-1].content
    # Query Mem0 for relevant memories based on the user_message
    relevant_memories = m0.get_memories(query=user_message, limit=3)
    # Format memories for injection into the prompt
    memory_context = "\n".join([mem.data.content for mem in relevant_memories if mem.data and
                                mem.data.content])
    state["memory_context"] = memory_context
    return state

# Add this function as a node in your LangGraph workflow before the LLM call.
# The LLM prompt would then include {memory_context} in its template.
```

Similarly, after the agent generates a response, you can store new information or updates to existing memories in Memo. This ensures the agent learns and adapts over time, providing a truly long-term memory solution that goes beyond the current session's chat history.

```
# Example of storing new memories after a response
def store_new_memory(state: Dict[str, Any]) -> Dict[str, Any]:
    user_message = state["messages"][-2].content if len(state["messages"]) >= 2 else ""
    agent_response = state["messages"][-1].content
    combined_text = f"User: {user_message}\nAgent: {agent_response}"
    m0.add_memory(data={"content": combined_text, "session_id": state["session_id"]})
    return state

# This can be another node in your graph or a callback after the agent completes.
```

By combining FastAPI for API exposure, LangGraph for orchestration, a remote vector database for RAG, and Memo for long-term memory, you build a powerful, production-ready multi-agent system capable of complex, stateful, and context-aware interactions.

KEY TAKEAWAYS

- Hosting a vector database like Chroma DB on AWS EC2 provides scalability and remote access for production RAG applications.
- Proper security group configuration on AWS EC2 is crucial to allow client connections to Chroma DB while maintaining security.
- FastAPI is an effective framework for exposing LangGraph RAG agents as web services, enabling production deployment.
- Managing conversation history and sessions is vital for stateful interactions in a production RAG agent.
- Memo offers a robust solution for adding long-term, context-aware memory to AI agents, allowing them to retain information across sessions.
- Integrating Memo involves retrieving relevant memories to inform agent responses and storing new information for future use.

CHECK YOUR UNDERSTANDING

Answer these before moving on. Model answers are in the Answer Key at the back of this guide.

1. Why is it beneficial to host Chroma DB on a cloud service like AWS EC2 for a production RAG application?
2. What is the primary purpose of configuring security groups when setting up Chroma DB on AWS EC2?
3. How does FastAPI contribute to deploying a LangGraph RAG agent in a production environment?
4. Explain the difference between short-term session history and long-term memory provided by tools like Memo in an AI agent.
5. Describe one method to integrate Memo into a LangGraph agent workflow.

REFERENCE

Glossary

Agent

An LLM augmented with tools, enabling it to reason about a task, decide which tools to use, and execute actions to achieve a goal.

Aws ec2

Amazon Elastic Compute Cloud, a web service that provides resizable compute capacity in the cloud.

Chroma db

An open-source vector database that simplifies embedding storage and retrieval for AI applications.

Embedding

A numerical representation of text, images, or other data that captures its semantic meaning in a high-dimensional space.

Fastapi

A modern, fast web framework for building APIs with Python, based on standard Python type hints.

Langchain expression language (lcel)

A declarative syntax in LangChain for composing runnable components into complex, efficient, and scalable chains.

Langgraph

A library for building stateful, multi-actor LLM applications by defining workflows as directed graphs with nodes and edges.

Mem0

A memory layer for AI agents that provides long-term, context-aware memory persistence across multiple interactions and sessions.

Retrieval augmented generation (rag)

An architecture that improves LLM responses by retrieving relevant information from an external knowledge base before generating an answer.

Security group

A virtual firewall for your EC2 instance to control inbound and outbound traffic.

Session history

The record of messages and interactions within a single, continuous conversation session with an AI agent.

Tool

A function or utility that an agent can call to interact with the external world, such as a database query tool or a web search tool.

Vector database

A database optimized for storing and querying high-dimensional vectors (embeddings) based on their similarity.

REFERENCE

Answer Key

Chapter 1: RAG Pipelines and Agent Fundamentals

1. Describe the core components of a RAG system and their respective roles.

A RAG system comprises a document loader, text splitter, embedding model, vector store, retriever, and an LLM. The loader ingests data, the splitter breaks it into chunks, and the embedding model converts chunks into numerical embeddings. The vector store saves these embeddings for efficient search. The retriever finds relevant chunks for a query, and the LLM synthesizes an answer using the query and retrieved context.

2. What are two key benefits of using LangChain Expression Language (LCEL) for building LLM applications?

LCEL offers several benefits, including enhanced composability, allowing developers to easily combine and reuse components. It also provides built-in support for streaming, asynchronous execution, and parallel processing. These features contribute to building more performant and robust LLM applications.

3. Explain how a LangChain agent differs from a simple LangChain chain.

A simple LangChain chain executes a predefined sequence of steps. An agent, however, uses an LLM as a reasoning engine to dynamically decide which tools to use and in what order to accomplish a task. This allows agents to handle more complex and open-ended problems by interacting with external environments.

4. What is the primary purpose of a SQL agent in LangChain?

The primary purpose of a SQL agent in LangChain is to enable natural language interaction with databases. It translates user queries expressed in plain English into executable SQL commands. The agent then runs these queries, retrieves results, and presents them back to the user in a human-readable format.

5. How does LangGraph enhance the capabilities of single LangChain agents?

LangGraph enhances agent capabilities by providing a framework for orchestrating stateful, multi-actor workflows. It allows for complex control flow, including conditional routing and cycles, which is difficult with single agents or linear chains. This enables the creation of more sophisticated and adaptive multi-agent systems.

Chapter 2: Production RAG Agents and Long-Term Memory

1. Why is it beneficial to host Chroma DB on a cloud service like AWS EC2 for a production RAG application?

Hosting Chroma DB on AWS EC2 offers several advantages for production. It provides scalability, allowing the database to grow with your application's needs. It also ensures high availability and

reliability, as cloud infrastructure is designed for uptime. Also, it enables remote access for multiple client applications, which is essential for distributed systems.

2. What is the primary purpose of configuring security groups when setting up Chroma DB on AWS EC2?

Security groups act as a virtual firewall for your EC2 instance. Their primary purpose is to control inbound and outbound network traffic. For Chroma DB, configuring security groups correctly is crucial to allow client applications to connect to port 8000 while restricting access to other ports for security, such as SSH (port 22) only from trusted IPs.

3. How does FastAPI contribute to deploying a LangGraph RAG agent in a production environment?

FastAPI provides a framework to expose the LangGraph RAG agent as a web API. This allows external applications to interact with the agent over HTTP. It simplifies handling requests, managing routes, and serializing responses, making the agent accessible and usable within a larger system architecture.

4. Explain the difference between short-term session history and long-term memory provided by tools like Memo in an AI agent.

Short-term session history refers to the conversation context within a single, ongoing interaction. It allows the agent to remember what was said recently in the current chat. Long-term memory, as provided by Memo, extends beyond a single session, allowing the agent to retain facts, preferences, and past interactions over extended periods, providing deeper contextual understanding across multiple, separate conversations.

5. Describe one method to integrate Memo into a LangGraph agent workflow.

One method to integrate Memo is by adding a dedicated node to the LangGraph workflow. This node would execute before the LLM generation step. It queries Memo for relevant memories based on the current user input, then injects these retrieved memories into the prompt given to the LLM, enriching the context for the agent's response.

CLOSING

Where to Go Next

After completing this course, continue to practice your skills by building your own projects. Start with small applications that extend the concepts learned here. Experiment with different LLMs, tools, and agent configurations. Real-world application is the best way to solidify your understanding.

Consider applying these techniques to solve a specific problem you or your team faces. Building a custom RAG system for internal documentation or developing an agent to automate a workflow are excellent next steps. Share your projects and seek feedback from the community.

To further expand your expertise, explore topics such as fine-tuning LLMs, advanced prompt engineering, or deploying LLM applications at scale. The principles of agentic design learned here form a strong foundation for many advanced AI engineering challenges.



MEGACOURSE

Learn Anything. Achieve Everything.

Track your progress, take the quizzes, and earn your certificate for this course at
megacourse.app/courses/langchain-and-langgraph-rag-to-multi-agent