✨ Member-only story

Building a Multi-Agent RAG System with LangGraph and Mistral LLM: A Step-by-Step Guide

Devvrat Rana · Follow

16 min read · Dec 31, 2024

Listen Share More

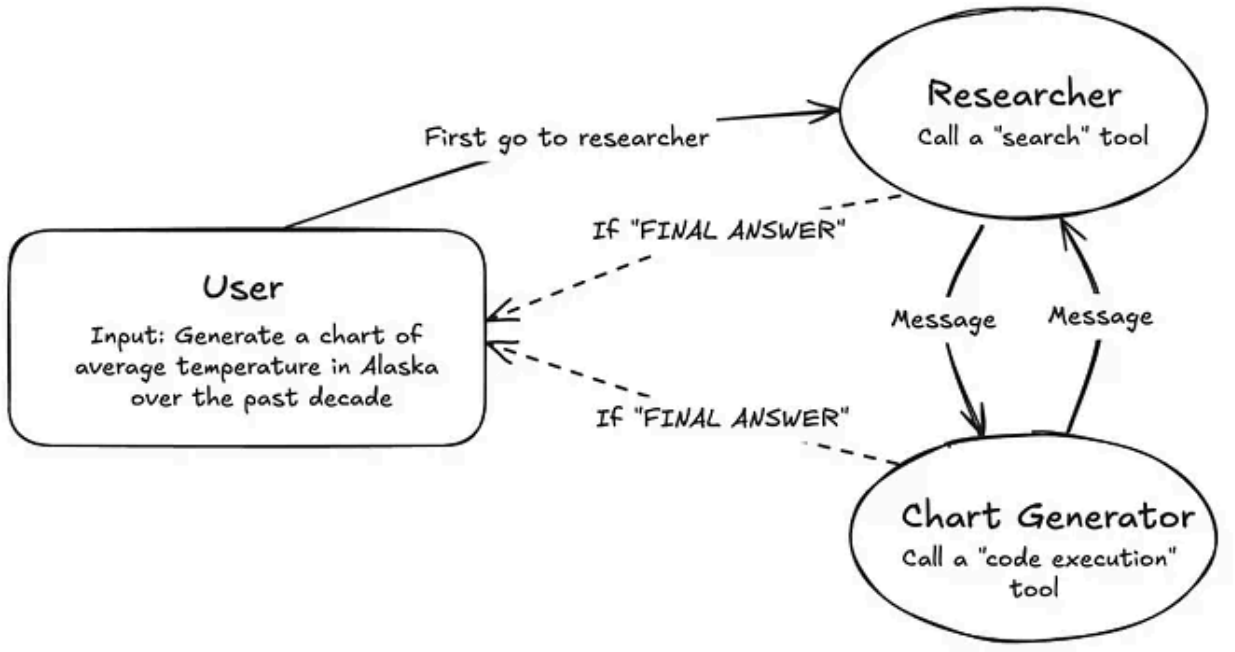


Image Credit to : https://langchain-ai.github.io/langgraph/tutorials/multi_agent/multi-agent-collaboration/

The rise of **Retrieval-Augmented Generation (RAG)** has significantly enhanced the capabilities of Large Language Models (LLMs) by combining their generative power with external knowledge retrieval. However, as tasks grow more complex, single

RAG systems may fall short of handling sophisticated workflows and decision-making processes. This is where **Multi-Agent Systems** come into play.

In this blog, we will explore how to build a **Multi-Agent RAG System** that leverages collaboration between specialized agents to perform more advanced tasks efficiently. By using **LangGraph**, a powerful orchestration framework built on **LangChain**, we can seamlessly coordinate interactions between these agents, enabling structured workflows, improved retrieval strategies, and more autonomous problem-solving.

What You'll Learn:

1. Understanding the need for **Multi-Agent RAG** systems.
2. Designing **specialized agents for distinct roles** (e.g., retrieval, summarization, decision-making).
3. Orchestrating agent interactions using **LangGraph** to achieve dynamic workflows.
4. Practical implementation of a **Multi-Agent RAG** pipeline with step-by-step code examples.

By the end of this blog, you'll be able to build and orchestrate an advanced, modular, and scalable **Multi-Agent RAG system** capable of solving real-world challenges. Let's dive in! 🚀

RAG Systems: Revolutionizing Contextual Understanding and Content Generation

Overview of RAG Systems

A **Retrieval-Augmented Generation (RAG)** system enhances content generation and context understanding by combining retrieval-based mechanisms with generative models. These systems are designed to leverage the strengths of both components to produce outputs that are more accurate, informative, and contextually relevant.

Key characteristics of RAG systems include:

- **Hybrid Approach**: Integration of retrieval mechanisms like document retrieval and passage ranking with generative processes.
- **External Knowledge Access**: Utilizing sources such as knowledge graphs and large text corpora to retrieve relevant information for generative enhancement.

- **Context Enrichment:** Incorporating retrieved knowledge into generative outputs to produce diverse, factual, and coherent content.
- **Targeted Applications:** Ideal for tasks like question answering, dialogue systems, and content generation requiring factual accuracy and contextual understanding.

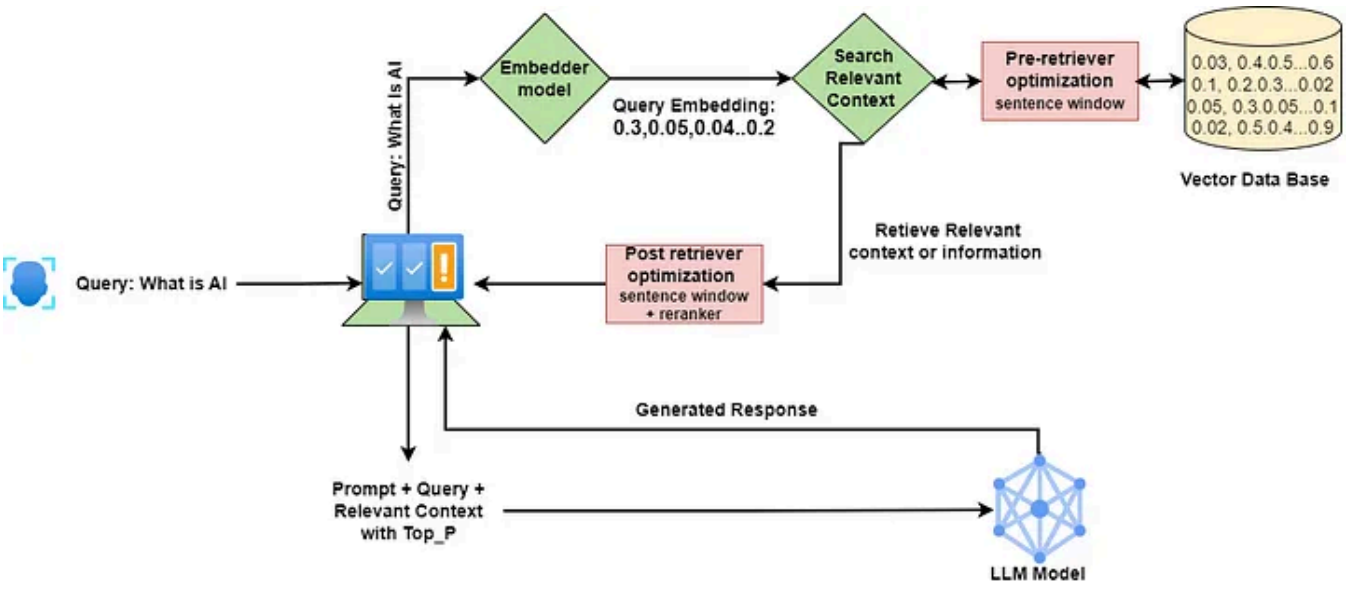


Image Credit to Author

Challenges in Enterprise Adoption

The initial implementations, often referred to as “naive RAG,” were insufficient for addressing the complex needs of large enterprises. These organizations manage vast repositories of structured and unstructured data, facing three critical challenges:

1. **Accuracy:** Enterprises, especially in industries like finance and healthcare, demand zero tolerance for errors or hallucinations.
2. **Relevancy:** Precise information retrieval is essential to avoid querying unnecessary data, ensuring efficiency and minimizing resource usage.
3. **Latency:** Enterprise-grade systems require near-instantaneous responses, with RAG pipelines needing to process and return results in under a second, even as model inference adds additional network latency.

If you’re new to RAG, I suggest starting with my previous blog, *“Creating a Chatbot Using Open-Source LLM and RAG Technology with LangChain and Flask,”* to understand the fundamentals of RAG.

The Path to Enterprise-Ready RAG Applications

Recent advancements are addressing these challenges, marking a turning point in the development of production-ready RAG systems for enterprise use. The emergence of new principles and innovations signals a transformative era in enterprise software development.

By overcoming barriers of **accuracy**, **relevancy**, and **latency**, RAG systems are poised to unlock the full potential of enterprise AI applications, delivering robust, efficient, and contextually aware solutions across industries.

Understanding AI Agents and Their Role in Automation

An AI agent is a software entity designed to perform tasks on behalf of a user, automating processes, making decisions, and intelligently interacting with its environment. These agents can exist as purely software-based systems or as physical entities.

How AI Agents Work

AI agents operate by:

1. **Perceiving the Environment**: Using sensors or other input mechanisms to gather information.
2. **Processing Information**: Leveraging algorithms or models to interpret and analyze the input data.
3. **Taking Action**: Acting through actuators or other means to achieve specific goals.

Core Concept: Dynamic Decision-Making

The foundation of AI agents lies in their ability to dynamically choose a sequence of actions to achieve a desired outcome. Unlike static workflows, such as those in LangChain's hardcoded chains, agents utilize a language model as a reasoning engine. This enables them to determine not only the optimal actions but also the most effective order to execute them.

Agents in LangChain

LangChain provides a versatile framework for building and utilizing AI agents, offering features such as:

- **A Standardized Interface**: Simplifying the development and deployment of agents.

- **Pre-Built Agents**: Ready-to-use solutions for common tasks.
- **End-to-End Examples**: Demonstrating practical implementations.

These agents are designed to integrate seamlessly with various tools, making them ideal for applications like:

- Grounded question-answering.
- API interactions.
- Context-driven actions.

This flexibility enables agents to adapt effectively to diverse scenarios and environments, making them powerful tools for solving complex, context-sensitive problems.

If you're a beginner, I recommend starting with my previous blog, *"Understanding LangChain Agents: A Beginner's Guide to How LangChain Agents Work,"* to grasp the basics of agents.

Core Components of an AI Agent:

1. **LLM**: Functions as the core reasoning engine, assigned a specific role and task.
2. **Memory**: Includes both short-term (for immediate context) and long-term memory (for historical data).
3. **Planning**: Involves advanced reasoning mechanisms such as reflection, self-critique, and query routing.
4. **Tools**: External utilities like calculators, web search, or APIs enhance functionality.

The ReAct Framework: Combining Reasoning and Action

One widely adopted framework for AI agents is **ReAct** (*Reason* + *Act*), designed for handling complex, sequential tasks. A ReAct agent integrates routing, query planning, and tool usage into a unified system, maintaining state in memory for multi-step interactions.

The **ReAct Process** follows these iterative steps:

1. **Thought**: The agent analyzes the user query and reasons about the next action to take.
2. **Action**: Based on reasoning, the agent executes a specific action (e.g., using a tool).
3. **Observation**: The agent evaluates the feedback from its action to guide subsequent steps.

This cycle continues until the agent completes the task and delivers a response to the user. The ReAct framework exemplifies how AI agents can effectively combine reasoning, action, and memory to solve complex tasks.

What is Agentic RAG?

Agentic RAG refers to an AI agent-driven implementation of Retrieval-Augmented Generation (RAG). By integrating AI agents into the RAG pipeline, this approach orchestrates its components and extends functionality beyond simple information retrieval and generation. This enhancement addresses the limitations of traditional, non-agentic RAG systems.

How Does Agentic RAG Work?

Agents can be incorporated into various stages of the RAG pipeline, but the concept typically refers to their use in the **retrieval component**. In this setup, retrieval becomes “agentic” through the inclusion of specialized retrieval agents with access to diverse tools, such as:

- **Vector Search Engines**: Performing searches over vector indices, as seen in standard RAG pipelines.
- **Web Search**: Accessing real-time information from the internet.
- **Calculators**: Solving mathematical or quantitative queries.
- **APIs**: Programmatic access to applications like email, chat programs, or other software.

Capabilities of a RAG Agent:

1. Deciding whether or not to retrieve information.

2. Choosing the appropriate tool for retrieving relevant data.
3. Formulating the retrieval query.
4. Evaluating retrieved content and determining if re-retrieval is necessary.

Agentic RAG Architectures

1. Single-Agent RAG (Router)

In its simplest form, an agentic RAG system acts as a **router**.

- The agent routes queries to multiple external knowledge sources, such as vector databases, web searches, or APIs (e.g., Slack or email).
- The agent decides which source to query based on the context of the task.

This setup enables flexible retrieval from diverse tools or data sources but remains limited to a single agent managing reasoning, retrieval, and generation.

If you're new to LangGraph, check out **"Introduction to LangGraph: Building Chatbots and Simplifying Cyclical Graphs in Agent Runtimes"** for detailed insights.

2. Multi-Agent RAG Systems

To overcome the limitations of single-agent systems, **multi-agent RAG architectures** chain multiple specialized agents.

- **Master Agent**: Acts as a coordinator, delegating retrieval tasks to specialized agents.
- **Specialized Agents**:
- **Internal Data Retrieval Agent**: Fetches proprietary or internal information.
- **Personal Data Agent**: Retrieves data from personal accounts like email or messaging platforms.
- **Public Data Agent**: Handles web searches or public information retrieval.

This architecture enhances scalability and efficiency by leveraging multiple agents to handle distinct tasks, resulting in more robust and contextually rich outputs.

It's time to implement Multi-Agent RAG using LangGraph!:

1. Installation:

```
%%capture --no-stderr
%pip install -U langchain_community langchain_experimental matplotlib langgraph
%pip install -qU duckduckgo-search langchain-community
%pip install -qU langchain_mistralai
```

2. Environment Setup:

```
import os
os.environ["MISTRAL_API_KEY"] = mistral_api_key  # Replace with your actual key
```

3. Define tools:

```
from typing import Annotated
from langchain_core.tools import tool
from langchain_experimental.utilities import PythonREPL
from langchain.agents import initialize_agent, load_tools, AgentType
from langchain_community.tools import DuckDuckGoSearchRun



search_tool = DuckDuckGoSearchRun()

# Warning: This executes code locally, which can be unsafe when not sandboxed

repl = PythonREPL()

@tool
def python_repl_tool(
    code: Annotated[str, "The python code to execute to generate your chart."],
):
    """Use this to execute python code. If you want to see the output of a valu
    you should print it out with `print(...)`. This is visible to the user."""
    try:
        result = repl.run(code)
    except BaseException as e:
        return f"Failed to execute. Error: {repr(e)}"
    result_str = f"Successfully executed:\n\`\`\`python\n{code}\n\`\`\`\nStdout
    return (
```

```
            result_str + "\n\nIf you have completed all tasks, respond with FINAL A
        )
```

Code Explanation:

```
from typing import Annotated
from langchain_core.tools import tool
from langchain_experimental.utilities import PythonREPL
from langchain.agents import initialize_agent, load_tools, AgentType
from langchain_community.tools import DuckDuckGoSearchRun
```

- **`Annotated`** : Used to annotate the type and purpose of inputs for better clarity and validation.

LangChain modules:

- `DuckDuckGoSearchRun` : Search tools for retrieving data.
- `tool` : A decorator to define custom tools in LangChain.
- `PythonREPL` : A utility for running Python code locally.
- `initialize_agent` and `AgentType` : For creating and managing agents that combine tools and language models.
- **Core Tool Purpose:** Allows users to execute Python code safely and retrieve search results from the web

```
search_tool = DuckDuckGoSearchRun()
```

`DuckDuckGoSearchRun` : Instantiates a search tool that fetches search results from DuckDuckGo. This can be integrated into an agent for retrieving online information.

```
repl = PythonREPL()
```

- `PythonREPL` : Creates a local Python execution environment. This is particularly useful for dynamic computations or data visualization in workflows.
- **Warning:** Executing arbitrary code can pose security risks if not sandboxed.

```
@tool
def python_repl_tool(
    code: Annotated[str, "The python code to execute to generate your chart."],
):
    """Use this to execute python code. If you want to see the output of a valu
    you should print it out with `print(...)`. This is visible to the user."""
    try:
        result = repl.run(code)
    except BaseException as e:
        return f"Failed to execute. Error: {repr(e)}"
    result_str = f"Successfully executed:\n```python\n{code}\n```\nStdout: {res
    return result_str + "\n\nIf you have completed all tasks, respond with FINA
```

Purpose: Defines a tool named `python_repl_tool` to execute Python code in the REPL.

Input:

- `code` (Annotated as a string): The Python code to execute.

Functionality:

- Attempts to execute the provided Python code using `repl.run(code)` .
- Captures and formats any standard output or errors.

Output:

- A success message with executed code and output.
- Error message in case of execution failure.

4. Create graph

Now that we've defined our tools and made some helper functions, will create the individual agents below and tell them how to talk to each other using LangGraph.

Define Agent Nodes

We now need to define the nodes.

First, we'll create a utility to create a system prompt for each agent.

```
def make_system_prompt(suffix: str) -> str:
 return (
 "You are a helpful AI assistant, collaborating with other assistants."
 " Use the provided tools to progress towards answering the question."
 " If you are unable to fully answer, that's OK, another assistant with differe
 " will help where you left off. Execute what you can to make progress."
 " If you or any of the other assistants have the final answer or deliverable,"
 " prefix your response with FINAL ANSWER so the team knows to stop."
 f"\n{suffix}"
 )
```

Use **LangGraph** to define a multi-agent workflow where two agents collaborate to perform research and generate charts:

```
from typing import Literal
from langchain_core.messages import BaseMessage, HumanMessage
from langgraph.prebuilt import create_react_agent
from langgraph.graph import MessagesState, END
from langgraph.types import Command
from langchain_mistralai import ChatMistralAI

llm = ChatMistralAI(
    model="mistral-large-latest",
    temperature=0,
    max_retries=2,
    #mistral_api_key=os.environ.get("MISTRAL_API_KEY") # Pass the API Key
)

def get_next_node(last_message: BaseMessage, goto: str):
    if "FINAL ANSWER" in last_message.content:
        # Any agent decided the work is done
```

```
research_agent = create_react_agent(
    llm,
    tools=[search_tool],
    state_modifier=make_system_prompt(
        "You can only do research. You are working with a chart generator colle
    ),
)


def research_node(
    state: MessagesState,
) -> Command[Literal["chart_generator", END]]:
    result = research_agent.invoke(state)
    goto = get_next_node(result["messages"][-1], "chart_generator")
    # wrap in a human message, as not all providers allow
    # AI message at the last position of the input messages list
    result["messages"][-1] = HumanMessage(
        content=result["messages"][-1].content, name="researcher"
    )
    return Command(
        update={
            # share internal message history of research agent with other agent
            "messages": result["messages"],
        },
        goto=goto,
    )


# Chart generator agent and node
# NOTE: THIS PERFORMS ARBITRARY CODE EXECUTION, WHICH CAN BE UNSAFE WHEN NOT SA

chart_agent = create_react_agent(
    llm,
    [python_repl_tool],
    state_modifier=make_system_prompt(
        "You can only generate charts. You are working with a researcher collea
    ),
)

def chart_node(state: MessagesState) -> Command[Literal["researcher", END]]:
    result = chart_agent.invoke(state)
    goto = get_next_node(result["messages"][-1], "researcher")
    # wrap in a human message, as not all providers allow
    # AI message at the last position of the input messages list
    result["messages"][-1] = HumanMessage(
        content=result["messages"][-1].content, name="chart_generator"
    )
    return Command(
        update={
            # share internal message history of chart agent with other agents
            "messages": result["messages"],
        },
```

```
        goto=goto,
    )
```

Explanation of code:

```
from typing import Literal
from langchain_core.messages import BaseMessage, HumanMessage
from langgraph.prebuilt import create_react_agent
from langgraph.graph import MessagesState, END
from langgraph.types import Command
from langchain_mistralai import ChatMistralAI
```

A. Imports

- `Literal` : Used for specifying fixed string values in type hints.

LangChain Modules:

- `BaseMessage` , `HumanMessage` : Represent and handle conversational messages.
- `ChatMistralAI` : LLM integration for Mistral AI.
- **LangGraph Modules:**
- `create_react_agent` : Creates agents with tools and a specific behavior.
- `MessagesState` : Tracks message history.
- `Command` : Defines the output of a node in the graph.
- `END` : Represents the terminal state of the workflow.

B. LLM Initialization

```
llm = ChatMistralAI(
    model="mistral-large-latest",
    temperature=0,
    max_retries=2,
)
```

`ChatMistralAI` :

- Integrates Mistral's language model with LangChain.
- `temperature=0` : Ensures deterministic outputs.
- `max_retries=2` : Configures retry attempts for API calls.
- `mistral_api_key` : (commented out) Placeholder for API key.

C. Helper Function

```
def get_next_node(last_message: BaseMessage, goto: str):
    if "FINAL ANSWER" in last_message.content:
        return END
    return goto
```

Determines the next node based on the last message:

- If the message contains `"FINAL ANSWER"` , the workflow ends (`END`).
- Otherwise, proceeds to the specified next node.

D. Research Agent and Node

Research Agent

```
research_agent = create_react_agent(
    llm,
    tools=[search_tool],
    state_modifier=make_system_prompt(
        "You can only do research. You are working with a chart generator colle
    ),
)
```

Agent Purpose: Focused solely on research.

Tools:

`search_tool` : Performs research tasks, such as retrieving information.

State Modifier:

- Ensures the agent adheres to the role of a “researcher.”

Research Node

```
def research_node(state: MessagesState) -> Command[Literal["chart_generator", E
    result = research_agent.invoke(state)
    goto = get_next_node(result["messages"][-1], "chart_generator")
    result["messages"][-1] = HumanMessage(
        content=result["messages"][-1].content, name="researcher"
    )
    return Command(
        update={"messages": result["messages"]},
        goto=goto,
    )
```

- Executes the research agent’s task using the current state.
- Converts the last AI-generated message into a human message (to maintain compatibility with tools).

Outputs:

- `update` : Updates the shared message history.
- `goto` : Specifies the next node ("chart_generator" or `END`).

E. Chart Generator Agent and Node

Chart Generator Agent

```
chart_agent = create_react_agent(
    llm,
    [python_repl_tool],
    state_modifier=make_system_prompt(
        "You can only generate charts. You are working with a researcher collea
    ),
)
```

Agent Purpose: Responsible for generating charts.

Tools:

- `python_repl_tool` : Executes Python code to generate charts.

State Modifier:

- Limits the agent to chart generation tasks.

Agent Purpose: Responsible for generating charts.

Tools:

- `python_repl_tool` : Executes Python code to generate charts.

State Modifier:

- Limits the agent to chart generation tasks.

Chart Node

```
def chart_node(state: MessagesState) -> Command[Literal["researcher", END]]:
    result = chart_agent.invoke(state)
    goto = get_next_node(result["messages"][-1], "researcher")
    result["messages"][-1] = HumanMessage(
        content=result["messages"][-1].content, name="chart_generator"
    )
    return Command(
        update={"messages": result["messages"]},
        goto=goto,
    )
```

- Executes the chart generator's task using the current state.
- Converts the last AI-generated message into a human message.

Outputs:

- `update` : Updates the shared message history.
- `goto` : Specifies the next node ("researcher" or `END`).

F. Workflow Summary

Collaborative Nodes:

- **Research Node**: Gathers information using the research agent.
- **Chart Node**: Generates charts based on the information provided.

State Management:

- Message history is shared between nodes to ensure context is preserved.

Termination:

- The workflow ends when any agent outputs a message containing `"FINAL ANSWER"`.

Key Points

- **Modularity**: Each agent and node has a specific role.
- **Collaboration**: Agents share message history to maintain coherence.
- **Dynamic Execution**: The use of Python code allows for dynamic chart generation, but poses potential risks if not sandboxed.
- **Customizability**: The workflow can be extended by adding more nodes or tools.

5. Define the Graph

We can now put it all together and define the graph!

```
from langgraph.graph import StateGraph, START

workflow = StateGraph(MessagesState)
workflow.add_node("researcher", research_node)
workflow.add_node("chart_generator", chart_node)

workflow.add_edge(START, "researcher")
graph = workflow.compile()
```

Code Explanation:

```
from langgraph.graph import StateGraph, START
```

- **`StateGraph`** : A class that represents a directed graph, where nodes define tasks and edges define transitions between tasks.
- **`START`** : A predefined constant representing the starting point of the workflow.

```
workflow = StateGraph(MessagesState)
```

`StateGraph(MessagesState)` :

- Initializes a new graph.
- Uses `MessagesState` to track the shared message history across nodes.

```
workflow.add_node("researcher", research_node)
workflow.add_node("chart_generator", chart_node)
```

Nodes:

- Represent tasks or agents within the workflow.
- Each node is assigned a name and a corresponding function:
- `"researcher"` → `research_node` : Handles research tasks.
- `"chart_generator"` → `chart_node` : Handles chart generation tasks.

```
workflow.add_edge(START, "researcher")
```

Edges:

- Define transitions between nodes.
- Here, the edge connects `START` to the `"researcher"` node, making `"researcher"` the first task executed in the workflow.

```
graph = workflow.compile()
```

`workflow.compile()`:

- Converts the defined graph into an executable form.
- Ensures all nodes and edges are valid and ready for execution.

How the Workflow Functions

Start Node:

- Execution begins at the `START` node.
- Transitions to the `"researcher"` node as per the defined edge.

Research Node (`research_node`):

- Executes the research task (e.g., gathering data or performing a search).
- Determines the next step (either transitioning to `"chart_generator"` or ending the workflow).

Chart Generator Node (`chart_node`):

- Executes the chart generation task (e.g., creating charts using Python code).
- Determines whether to loop back to `"researcher"` or end the workflow.

End:

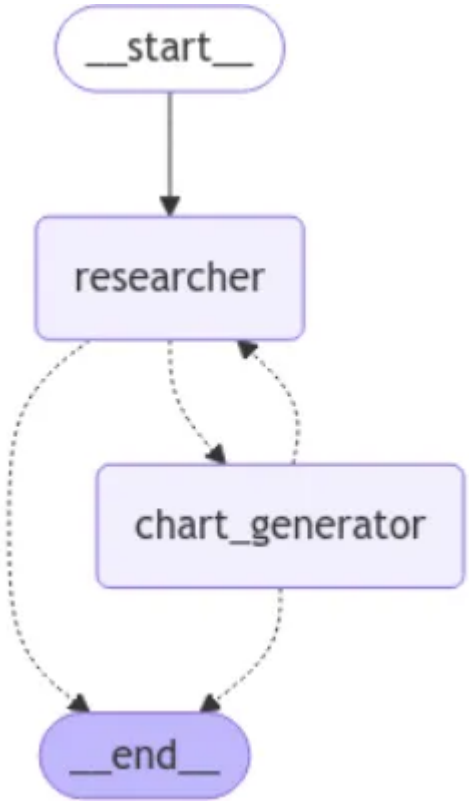
- The workflow terminates when a node returns `END`.

6. Visualize a state graph

```
from IPython.display import Image, display

try:
    display(Image(graph.get_graph().draw_mermaid_png()))
except Exception:
    # This requires some extra dependencies and is optional
    pass
```

This code is used to **visualize a state graph** (created using LangGraph) in a Jupyter Notebook environment by rendering it as a **PNG image**



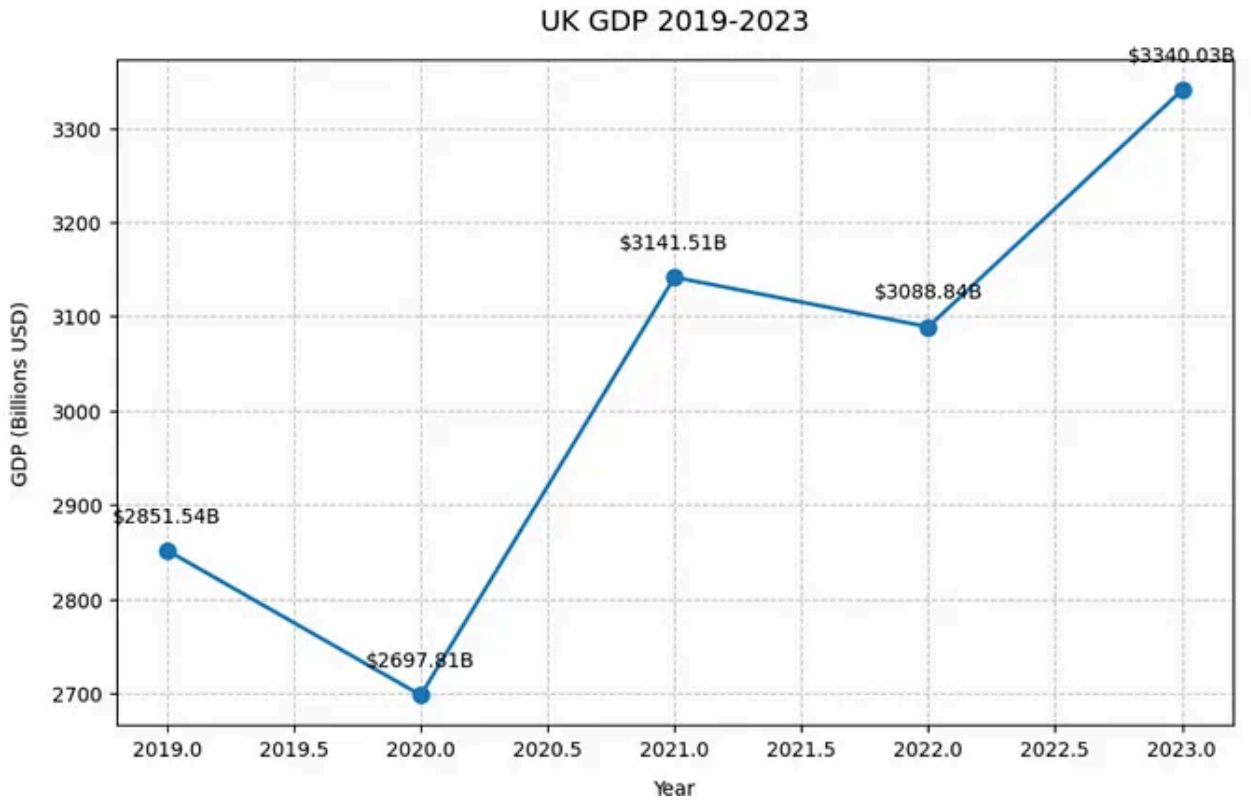
7. Invoke the Multi Agentic RAG

```
events = graph.stream(
    {
        "messages": [
            (
                "user",
                "First, get the UK's GDP over the past 5 years, then make a lir
                "Once you make the chart, finish.",
            )
        ],
    },
    # Maximum number of steps to take in the graph
```

```
        {"recursion_limit": 10},
    )
    for s in events:
        print(s)
        print("----")
```

Output:

```
/usr/local/lib/python3.10/dist-packages/langchain_community/utilities/duckduckg
  ddgs_gen = ddgs.text(
{'researcher': {'messages': [HumanMessage(content="First, get the UK's GDP over
----
```



Agentic RAG systems redefine the traditional RAG pipeline by adding reasoning, decision-making, and adaptability, making them ideal for complex, real-world applications requiring diverse knowledge sources and nuanced retrieval strategies.

My Other Blog links:

1. Creating a Chatbot Using Open-Source LLM and RAG Technology with Lang Chain and Flask

2. Build a Chatbot with Advance RAG System: with LlamaIndex, OpenSource LLM, Flask and LangChain

3. How to build Chatbot with advance RAG system with by using LlamaIndex and OpenSource LLM with Flask...

4. How to develop a chatbot using the open-source LLM Mistral-7B, Lang Chain Memory, ConversationChain, and Flask.

5. Understanding LangChain Agents: A Beginner's Guide to How LangChain Agents Work

6. Building Custom tools for LLM Agent by using Lang Chain

7. Deploying Llama 3.1 8b LLM Locally with Ollama to build a RAG-Powered Chatbot using LlamaIndex and Flask

8. Introduction to LangGraph: Building Chatbots and Simplifying Cyclical Graphs in Agent Runtimes

Conclusion

In this blog, we explored the process of building a **Multi-Agent RAG System** using **LangGraph** and the powerful **Mistral LLM**. By leveraging LangGraph's state graph capabilities, we created a seamless workflow that allows agents to collaborate effectively. The modular design of LangGraph enabled us to define roles for each agent — such as research and chart generation — while ensuring they communicate and share context through a unified message state.

Integrating **Mistral LLM** brought robust language understanding and generation capabilities to the system, enabling it to handle complex queries and tasks with precision. This combination of tools demonstrates how **multi-agent** architectures can be built efficiently to solve real-world problems, from information retrieval to dynamic visualization.

LangGraph and Mistral LLM open up exciting possibilities for developing intelligent, collaborative systems. Whether you're working on research assistants,