✨ Member-only story

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

Devvrat Rana · Follow

14 min read · Nov 25, 2024

Listen Share ••• More

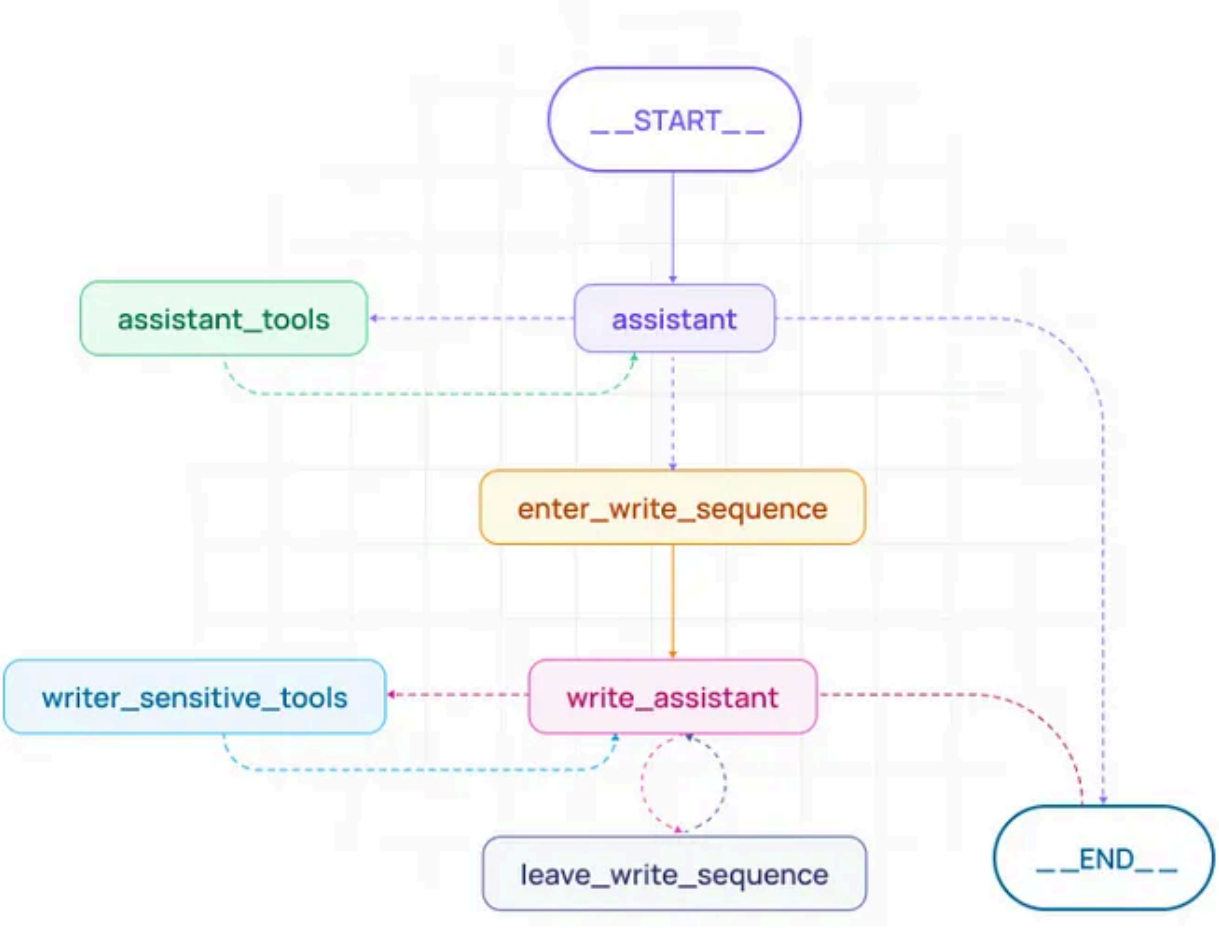


Image Credit: https://www.langchain.com/langgraph

Introduction:

Building complex agent runtimes in the **LangChain** ecosystem often requires a sophisticated approach to managing workflows and dependencies i.e **LangGraph.**

LangGraph— a powerful extension built on top of **LangChain**, designed to seamlessly integrate with its ecosystem while introducing a game-changing feature: effortless creation of cyclical graphs.

Cyclical graphs are essential for designing dynamic and iterative agent workflows, enabling more advanced and adaptable runtime behaviors. With **LangGraph**, developers can now simplify the process of designing these graphs, unlocking new possibilities for creating intelligent agents.

In this blog post, we'll take you through:
- The **motivations** behind **LangGraph's** creation.
- The **core functionality** it brings to the **LangChain** ecosystem.

- Two example **agent runtimes** implemented using **LangGraph**.
- Common runtime modifications based on real-world feedback and how to implement them.

Whether you're an experienced **LangChain** developer or just exploring **agent-based** AI solutions, **LangGraph** offers innovative tools to elevate your workflows. Let's dive in!

What is LangGraph

LangGraph, an advanced library within the **LangChain** ecosystem, offers a streamlined approach to designing and managing multi-agent LLM applications. By modeling workflows as cyclical graphs, **LangGraph** empowers developers to coordinate the interactions of multiple LLM agents, facilitating seamless communication and efficient execution of complex tasks.

While **LangChain** specializes in creating linear workflows using Directed Acyclic Graphs (DAGs), LangGraph expands this functionality by introducing support for cycles within workflows. This enhancement enables the development of more sophisticated and adaptive systems, reflecting the dynamic behavior of intelligent agents that can revisit tasks and make decisions based on new or evolving information.

Motivation

One of **LangChain**'s key strengths is its ability to simplify the creation of custom chains, supported by tools like the **LangChain Expression Language**. However, until now, the framework lacked a straightforward way to introduce cycles into these workflows, which traditionally operate as **Directed Acyclic Graphs (DAGs)** — a common structure in **data orchestration** frameworks.

A recurring pattern we've observed in complex LLM applications is the need for cycles in the runtime. These cycles often leverage LLMs to dynamically reason about the next step in the process. This functionality — essentially running an LLM in a loop — unlocks the ability to use LLMs for reasoning tasks, making them a crucial component of intelligent, adaptable systems often referred to as **agents**.

Why Agentic Behavior Matters

Consider a typical **retrieval-augmented generation (RAG)** application. In the standard setup, a retriever fetches documents, which are then passed to an LLM to generate a response. While effective in many cases, this approach struggles **when**

the retriever fails to return meaningful results. In such situations, it would be ideal for the LLM to recognize the issue, refine the query, and attempt retrieval again. By introducing a loop, the system becomes more flexible, capable of handling vague or non-predefined use cases effectively.

This agentic behavior can be distilled into a simple, yet ambitious, loop:
1. The LLM determines what action to take or what response to provide.
2. The system executes the action and returns to step 1.
This loop continues until a final response is generated. It's the foundational logic behind tools like the core `**AgentExecutor**` and projects like **AutoGPT.** While simple in structure, it's ambitious because it delegates nearly all decision-making and reasoning to the LLM.

The Need for More Control
In practice, as we've worked with the community and enterprises to deploy agents, we've found that more controlled workflows are often required. For instance:
- You might need an agent to always execute a specific tool first.
- You may require more precise control over tool usage.
- Different prompts might be necessary based on the agent's current state.

These controlled workflows are often best modeled as *state machines*. State machines provide looping capabilities to handle ambiguous inputs while allowing developers to guide how the loops are structured.

LangGraph introduces a way to build these state machines as graphs. By enabling developers to easily define cyclical workflows, **LangGraph** bridges the gap between flexibility and control. It combines the adaptability of agentic loops with the precision of state machine logic, empowering developers to create more robust and dynamic multi-agent applications.

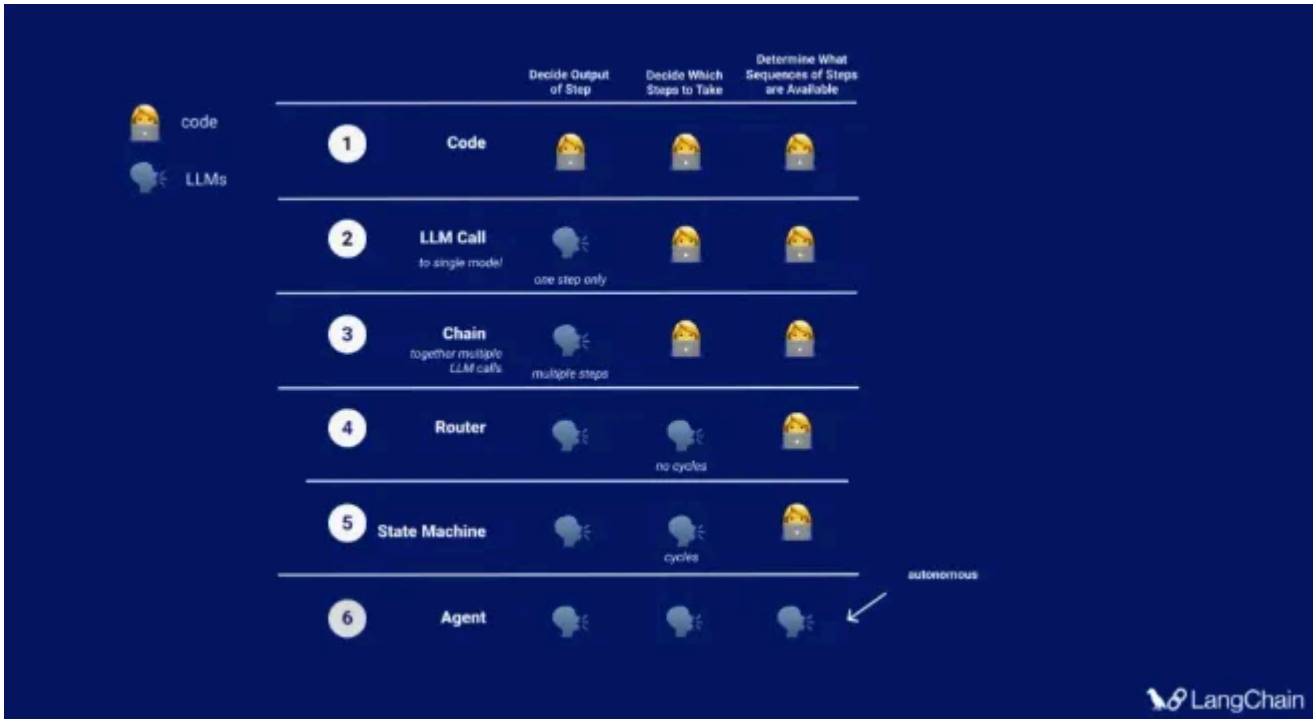


image credit: https://blog.langchain.dev/langgraph/

Lang Graph Functionality

StateGraph

The **StateGraph** class represents a graph that facilitates dynamic state management over time. When initializing this class, you provide a state definition — a central object that evolves as nodes in the graph perform operations on its attributes. These operations are returned as key-value pairs, updating the state incrementally.

State attributes can be updated in two distinct ways:
1. **Override:** An attribute's value is completely replaced by the new value returned by a node. This approach is ideal when you need to update an attribute with a fresh value.
2. **Additive Updates:** An attribute's value is extended by appending new information. This is particularly useful when the attribute represents cumulative data, such as a list of actions, where each node contributes additional entries.

You determine whether an attribute is overridden or updated additively during the initial state definition. This flexibility makes **StateGraph** a powerful tool for managing evolving workflows in a structured and dynamic manner.

```
from langgraph.graph import StateGraph
from typing import TypedDict, List, Annotated
import Operator
```

```
class State(TypedDict):
    input: str
    all_actions: Annotated[List[str], operator.add]


graph = StateGraph(State)
```

Nodes

After creating a **StateGraph**, you can add nodes using the **graph.add_node(name, value)** syntax.

- The **name** parameter is a string that identifies the node, allowing you to reference it when defining edges.
- The **value** parameter is either a function or an **LCEL** runnable. This callable will process the state object.

The function or **LCEL** runnable should:
1. Accept a dictionary in the same format as the **State** object as its input.
2. Output a dictionary containing the keys of the **State** object to update, along with their corresponding values.

This design ensures seamless integration and efficient state updates within the graph.

```
graph.add_node("model", model)
graph.add_node("tools", tool_executor)
```

There is also a special `END` node that is used to represent the end of the graph. It is important that your cycles be able to end eventually!

```
from langgraph.graph import END
```

Edges

After adding nodes to your **StateGraph**, you can define edges to establish the graph's structure. There are three main types of edges:

1. Starting Edge
The starting edge connects the graph's entry point to a specific node. This ensures that the designated node is the first one executed when input is passed to the graph.

```
graph.set_entry_point("model")
```

2. Normal Edges
Normal edges define a strict sequence where one node is always executed after another. For instance, in a basic agent runtime, you might want the "model" node to be executed after the "tools" node.

```
graph.add_edge("tools", "model")
```

3. Conditional Edges
Conditional edges enable dynamic execution paths based on a function's output (often powered by an LLM). These edges are ideal for creating decision-making workflows.

To create a conditional edge, you need:
1. **The upstream node:** The node whose output determines the next step.
2. **A function:** This function decides which node to call next. It should return a string corresponding to a downstream node.
3. **A mapping:** This maps the possible outputs of the function (step 2) to the appropriate downstream nodes.

Example (Pseudocode):
In this example, after the "model" node runs, the graph either exits or calls the "tools" node based on the function **should_continue**:

```
graph.add_conditional_edge(
    "model",
    should_continue,
    {
        "end": END,
        "continue": "tools"
    }
)
```

This flexible edge structure allows **StateGraph** to handle both linear and dynamic workflows, enabling robust and adaptive runtime behaviors.

Compile

Once the graph is fully defined, it can be compiled into a runnable object. This process transforms the graph definition into a functional entity that can execute workflows.

The resulting runnable object supports all the standard methods available in LangChain runnables, such as **`.invoke`**, **`.stream`**, **`.astream_log`**, and more. This ensures seamless integration, allowing the graph to be used just like any other LangChain chain or runnable.

```
app = graph.compile()
```

Agent Executor

We've rebuilt the canonical LangChain `AgentExecutor` using LangGraph, enabling the use of existing LangChain agents while making it easier to customize the internal workings of the `AgentExecutor`.

By default, the state of this graph includes familiar concepts from LangChain agents, such as:
- **input:** The initial input to the agent.
- **chat_history:** A record of past interactions.
- **intermediate_steps:** The steps taken by the agent during execution.
- **agent_outcome:** Represents the most recent result produced by the agent.

This redesign combines the power of LangGraph with the flexibility of LangChain agents, allowing for more adaptable and customizable agent workflows.

```
from typing import TypedDict, Annotated, List, Union
from langchain_core.agents import AgentAction, AgentFinish
from langchain_core.messages import BaseMessage
import operator


class AgentState(TypedDict):
   input: str
   chat_history: list[BaseMessage]
   agent_outcome: Union[AgentAction, AgentFinish, None]
   intermediate_steps: Annotated[list[tuple[AgentAction, str]], operator.add]
```

Chat Agent Executor

A growing trend in AI development is the rise of “chat” models that operate on a list of messages. These models, often equipped with advanced features like function calling, are particularly well-suited for creating agent-like experiences. When working with such models, representing the state of an agent as a list of messages becomes a natural and intuitive approach.

To support this, we’ve developed an agent runtime specifically designed for these models. The runtime takes a list of messages as input, and the nodes within the graph progressively add to this list over time, enabling seamless and dynamic state management.

```
from typing import TypedDict, Annotated, Sequence
import operator
from langchain_core.messages import BaseMessage


class AgentState(TypedDict):
    messages: Annotated[Sequence[BaseMessage], operator.add]
```

Time to Build a Chatbot with LangGraph

Installation

```
%%capture --no-stderr
%pip install -U langgraph langsmith langchain-openai langchain_anthropic tavily
```

Let's start by creating a simple chatbot using **LangGraph**. This chatbot will reply to user messages and help you understand the basic ideas behind **LangGraph**. By the end of this section, you'll have a working chatbot.

First, we'll create a **`StateGraph`**, which acts as the backbone of the chatbot. It works like a **"state machine,"** where you add nodes to represent the LLM and functions the chatbot can use. Then, you'll connect these nodes with edges to decide how the chatbot moves between tasks.

```
from typing import Annotated

from typing_extensions import TypedDict

from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages


class State(TypedDict):
    # Messages have the type "list". The `add_messages` function
    # in the annotation defines how this state key should be updated
    # (in this case, it appends messages to the list, rather than overwriting t
    messages: Annotated[list, add_messages]


graph_builder = StateGraph(State)
```

Our graph is now ready to perform two important tasks:

1. Each node can take the current State as input and return updates to it.
2. Updates to messages are added to the existing list instead of replacing it, thanks to the **`add_messages`** function used with the **`Annotated`** syntax.

Next, we'll add a **"chatbot"** node. Nodes are like small units of work and are usually just regular Python functions.

```
from langchain_openai import OpenAI, ChatOpenAI
from langchain_anthropic import ChatAnthropic

#llm = ChatAnthropic(model="claude-3-5-sonnet-20240620",anthropic_api_key=api_k

llm = ChatOpenAI(openai_api_key=api_key)


def chatbot(state: State):
    return {"messages": [llm.invoke(state["messages"])]}


# The first argument is the unique node name
# The second argument is the function or object that will be called whenever
# the node is used.
graph_builder.add_node("chatbot", chatbot)
```

The chatbot node function works by taking the current State as input and returning a dictionary with an updated “messages” list. This is the basic way all LangGraph node functions work.

The **`add_messages`** function in our State ensures that the LLM’s response messages are added to the existing messages in the State, instead of replacing them.

Now, let’s add an entry point. This tells the graph where to start its process each time it runs.

```
graph_builder.add_edge(START, "chatbot")
```

Similarly, set a **finish** point. This instructs the graph **“any time this node is run, you can exit.”**

```
graph_builder.add_edge("chatbot", END)
```

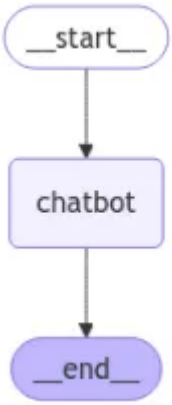
Finally, we’ll want to be able to run our graph. To do so, call **“compile()”** on the graph builder. This creates a **“CompiledGraph”** we can use invoke on our state.

```
graph = graph_builder.compile()
```

You can visualize the graph using the **get_graph** method and one of the **“draw” methods, like draw_ascii or draw_png.** The draw methods each require additional dependencies.

```
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
```



Now let’s run the chatbot!

Tip: You can exit the chat loop at any time by typing “quit”, “exit”, or “q”.

```
def stream_graph_updates(user_input: str):
    for event in graph.stream({"messages": [("user", user_input)]}):
        for value in event.values():
            print("Assistant:", value["messages"][-1].content)


while True:
    try:
        user_input = input("User: ")
        if user_input.lower() in ["quit", "exit", "q"]:
            print("Goodbye!")
```

```
        break
```



```
        stream_graph_updates(user_input)
        break
```

Now our simple Chatbot is ready now time to integrate some tools like **Search Engine** with it so that our chatbot can find relevant answer from internet and provide better response

So, we are going to integrate tavily search api with our chatbot.

```
from langchain_community.tools.tavily_search import TavilySearchResults

tool = TavilySearchResults(max_results=2)
tools = [tool]
tool.invoke("What's a 'node' in LangGraph?")
```

Next, we'll start defining our graph. The following is all the same as in Part 1, except we have added bind_tools on our LLM. This lets the LLM know the correct JSON format to use if it wants to use our search engine.

```
from typing import Annotated

from langchain_anthropic import ChatAnthropic
from langchain_openai import OpenAI, ChatOpenAI
from typing_extensions import TypedDict

from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages


class State(TypedDict):
    messages: Annotated[list, add_messages]


graph_builder = StateGraph(State)
```

```
llm = ChatOpenAI()
# Modification: tell the LLM which tools it can call
llm_with_tools = llm.bind_tools(tools)


def chatbot(state: State):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}


graph_builder.add_node("chatbot", chatbot)
```

Next, we need to create a function to run tools when they are called. We'll do this by adding the tools to a new node.

In the example below, we create a `**BasicToolNode**`. This node checks the latest message in the state and runs tools if it finds any `**tool_calls**`. It uses the **tool-calling** feature supported by LLMs like Anthropic, OpenAI, Google Gemini, and others.

Later, we'll use **LangGraph's** prebuilt `**ToolNode**` to make things faster, but building it ourselves first will help us understand how it works.

```
import json

from langchain_core.messages import ToolMessage


class BasicToolNode:
    """A node that runs the tools requested in the last AIMessage."""

    def __init__(self, tools: list) -> None:
        self.tools_by_name = {tool.name: tool for tool in tools}

    def __call__(self, inputs: dict):
        if messages := inputs.get("messages", []):
            message = messages[-1]
        else:
            raise ValueError("No message found in input")
        outputs = []
        for tool_call in message.tool_calls:
            tool_result = self.tools_by_name[tool_call["name"]].invoke(
                tool_call["args"]
            )
            outputs.append(
                ToolMessage(
                    content=json.dumps(tool_result),
```

```
                    name=tool_call["name"],
                    tool_call_id=tool_call["id"],
                )
            )
        return {"messages": outputs}


tool_node = BasicToolNode(tools=[tool])
graph_builder.add_node("tools", tool_node)
```

With the tool node added, we can set up conditional edges.

Edges control how the graph moves from one node to the next. Conditional edges use “if” statements to decide the next node based on the current state of the graph. These functions take the current state as input and return a string (or list of strings) that tells the graph which node(s) to go to next.

In the example below, we define a router function called **`route_tools`**. This function checks if the chatbot’s output includes any **`tool_calls`**. We add this function to the graph using **`add_conditional_edges`**, telling the graph to use it after the chatbot node finishes to decide the next step.

- If tool calls are present, the graph routes to the tools node.
- If not, it ends the process (**`END`**).

Later, we’ll replace this with LangGraph’s prebuilt **`tools_condition`** for simplicity, but creating it ourselves first helps make the process clearer.

```
from typing import Literal


def route_tools(
    state: State,
):
    """
    Use in the conditional_edge to route to the ToolNode if the last message
    has tool calls. Otherwise, route to the end.
    """
    if isinstance(state, list):
        ai_message = state[-1]
    elif messages := state.get("messages", []):
        ai_message = messages[-1]
    else:
```

```
            raise ValueError(f"No messages found in input state to tool_edge: {stat
        if hasattr(ai_message, "tool_calls") and len(ai_message.tool_calls) > 0:
            return "tools"
        return END


    # The `tools_condition` function returns "tools" if the chatbot asks to use a t
    # it is fine directly responding. This conditional routing defines the main age
    graph_builder.add_conditional_edges(
        "chatbot",
        route_tools,
        # The following dictionary lets you tell the graph to interpret the conditi
        # It defaults to the identity function, but if you
        # want to use a node named something else apart from "tools",
        # You can update the value of the dictionary to something else
        # e.g., "tools": "my_tools"
        {"tools": "tools", END: END},
    )
    # Any time a tool is called, we return to the chatbot to decide the next step
    graph_builder.add_edge("tools", "chatbot")
    graph_builder.add_edge(START, "chatbot")
    graph = graph_builder.compile()
```

Notice that conditional edges start from a single node. This tells the graph “any time the ‘chatbot’ node runs, either go to ‘tools’ if it calls a tool, or end the loop if it responds directly.

Like the prebuilt **tools_condition**, our function returns the **END** string if no tool calls are made. When the graph transitions to **END**, it has no more tasks to complete and ceases execution. Because the condition can return **END**, we don’t need to explicitly set a **finish_point** this time. Our graph already has a way to finish!

Let’s visualize the graph we’ve built. The following function has some additional dependencies to run that are unimportant for this tutorial.

```
    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
```

```
while True:
    try:
        user_input = input("User: ")
        if user_input.lower() in ["quit", "exit", "q"]:
            print("Goodbye!")
            break

        stream_graph_updates(user_input)
    except:
        # fallback if input() is not available
        user_input = "What do you know about LangGraph?"
        print("User: " + user_input)
        stream_graph_updates(user_input)
        break
```

You've built a conversational agent in LangGraph that can use a search engine to fetch up-to-date information when needed. This allows it to handle a broader range of user questions. To see all the steps your agent took, check out the LangSmith trace.

However, our chatbot still can't remember previous interactions, which limits its ability to have smooth, multi-turn conversations. In our next blog will build more advance chatbot and multi RAG system with LangGraph

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

Conclusion

LangGraph opens up exciting possibilities for designing and managing complex workflows in AI applications. By enabling the creation of cyclical graphs, it extends the functionality of the **LangChain** ecosystem, making it easier to build dynamic and adaptable systems like chatbots and agent runtimes.

In this blog, we explored how to get started with **LangGraph**, from understanding its core concepts to implementing **nodes, edges, and state** management. We even built a simple **chatbot** to demonstrate its practical use. **LangGraph** not only simplifies the process of creating agent workflows but also enhances flexibility, allowing developers to tackle more complex, real-world scenarios.

As you dive deeper into **LangGraph**, you'll discover its potential to streamline development and unlock new possibilities in AI-driven applications. Whether you're building robust agent systems or improving **conversational agents**, **LangGraph** is a powerful tool to have in your toolkit.

Feel free to connect with me on LinkedIn

References

🚀 **LangGraph Quick Start**

Build language agents as graphs

langchain-ai.github.io

LangGraph

Deploy your LLM app instantly with LangServe.

www.langchain.com

Tavily Search | 🦜🔗 LangChain

Tavily's Search API is a search engine built specifically for AI agents (LLMs), delivering real-time, accurate, and...

python.langchain.com

LangGraph Tutorial: A Comprehensive Guide for Beginners

Learn LangGraph with this beginner-friendly tutorial featuring Python code examples.

blog.futuresmart.ai

LangGraph

TL;DR: LangGraph is module built on top of LangChain to better enable creation of cyclical graphs, often needed for...

blog.langchain.dev

https://www.datacamp.com/tutorial/langgraph-tutorial

Langgraph | Langgraph Tutorial | Agents | Graphs | Introduction To Langgraph

Follow

Written by Devvrat Rana