Medium Search Write

How to Build AI Agents with LangGraph: A Step-by-Step Guide

Lore Van Oudenhove · Follow
11 min read · Sep 6, 2024

673 16

Introduction

In the world of AI, retrieval-augmented generation (RAG) systems have become common tools for handling simple queries and generating contextually relevant responses. However, as the demand for more sophisticated AI applications grows, there's a need for systems that go beyond these retrieval capabilities. Enter AI agents — autonomous entities capable of performing complex, multi-step tasks, maintaining state across interactions, and dynamically adapting to new information. **LangGraph**, a powerful extension of the LangChain library, is designed to help developers build these advanced AI agents by enabling stateful, multi-actor applications with cyclic computation capabilities.

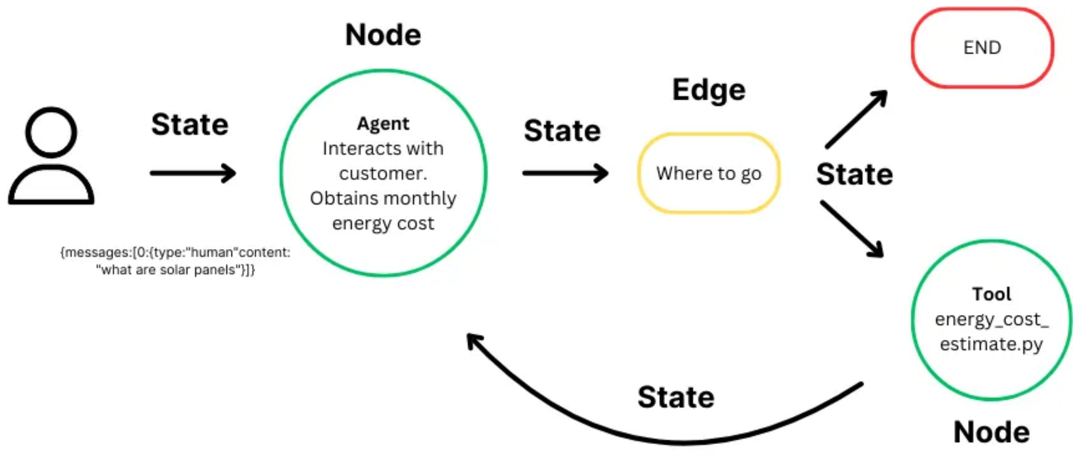
In this article, we'll explore how LangGraph transforms AI development and provide a step-by-step guide on how to build your own AI agent using an example that computes energy savings for solar panels. This example will showcase how LangGraph's unique features can create intelligent, adaptable, and real-world-ready AI systems.

What is LangGraph?

LangGraph is an advanced library built on top of LangChain, designed to enhance your Large Language Model (LLM) applications by introducing cyclic computational capabilities. While LangChain allows the creation of Directed Acyclic Graphs (DAGs) for linear workflows, LangGraph takes this a step further by enabling the addition of cycles, which are essential for

Top highlight

developing complex, agent-like behaviors. These behaviors allow LLMs to continuously loop through a process, dynamically deciding what action to take next based on evolving conditions.


Node
State
Agent
Interacts with customer. Obtains monthly energy cost
{messages:[0:{type:"human"content: "what are solar panels"}]}
State
Edge
Where to go
State
END
Tool
energy_cost_estimate.py
State
Node

Langraph: nodes, states, and edges.

At the heart of LangGraph is the concept of a **stateful graph**:

- **State**: Represents the context or memory that is maintained and updated as the computation progresses. It ensures that each step in the graph can access relevant information from previous steps, allowing for dynamic decision-making based on accumulated data throughout the process.
- **Nodes**: Serve as the building blocks of the graph, representing individual computation steps or functions. Each node performs a specific task, such as processing inputs, making decisions, or interacting with external systems. Nodes can be customized to execute a wide range of operations within the workflow.
- **Edges**: Connect nodes within the graph, defining the flow of computation from one step to the next. They support conditional logic, allowing the path of execution to change based on the current state and facilitate the movement of data and control between nodes, enabling complex, multi-step workflows.

LangGraph redefines AI development by seamlessly managing graph structure, state, and coordination, empowering the creation of sophisticated, multi-actor applications. With automatic state management, LangGraph ensures that context is preserved across interactions, enabling your AI to respond intelligently to changing inputs. Its streamlined agent coordination guarantees precise execution and efficient information exchange, letting you focus on crafting innovative workflows rather than technical intricacies. LangGraph's flexibility allows for the development of tailored, high-performance applications, while its scalability and fault tolerance ensure your systems remain robust and reliable, even at the enterprise level.

Step-by-step Guide

Now that we have a solid understanding of what LangGraph is and how it enhances AI development, let's dive into a practical example. In this scenario, we'll build an AI agent designed to calculate potential energy savings for solar panels based on user input. This agent can be implemented as a lead generation tool on a solar panel seller's website, where it interacts with potential customers, offering personalized savings estimates. By gathering key data such as monthly electricity costs, this AI agent helps educate customers on the financial benefits of solar energy while simultaneously qualifying leads for follow-up by the sales team. This example showcases the power of LangGraph in creating intelligent, dynamic systems that can automate complex tasks and drive business value.

Step 1: Import Necessary Libraries

We start by importing all the essential Python libraries and modules required for the project.

```python
from langchain_core.tools import tool
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import Runnable
from langchain_aws import ChatBedrock
import boto3
from typing import Annotated
from typing_extensions import TypedDict
from langgraph.graph.message import AnyMessage, add_messages
from langchain_core.messages import ToolMessage
from langchain_core.runnables import RunnableLambda
from langgraph.prebuilt import ToolNode
from langgraph.prebuilt import tools_condition
```

These imports set the foundation for utilizing LangChain, LangGraph, and AWS services to build our AI assistant.

Step 2: Define the Tool for Calculating Solar Savings

Next, we define a tool that will handle the computation of energy savings based on the monthly electricity cost provided by the user.

```
@tool
def compute_savings(monthly_cost: float) -> float:
    """
    Tool to compute the potential savings when switching to solar energy based o

    Args:
        monthly_cost (float): The user's current monthly electricity cost.

    Returns:
        dict: A dictionary containing:
            - 'number_of_panels': The estimated number of solar panels required.
            - 'installation_cost': The estimated installation cost.
            - 'net_savings_10_years': The net savings over 10 years after instal
    """
    def calculate_solar_savings(monthly_cost):
        # Assumptions for the calculation
        cost_per_kWh = 0.28
        cost_per_watt = 1.50
        sunlight_hours_per_day = 3.5
        panel_wattage = 350
        system_lifetime_years = 10

        # Monthly electricity consumption in kWh
        monthly_consumption_kWh = monthly_cost / cost_per_kWh

        # Required system size in kW
        daily_energy_production = monthly_consumption_kWh / 30
        system_size_kW = daily_energy_production / sunlight_hours_per_day

        # Number of panels and installation cost
        number_of_panels = system_size_kW * 1000 / panel_wattage
        installation_cost = system_size_kW * 1000 * cost_per_watt

        # Annual and net savings
        annual_savings = monthly_cost * 12
        total_savings_10_years = annual_savings * system_lifetime_years
        net_savings = total_savings_10_years - installation_cost

        return {
            "number_of_panels": round(number_of_panels),
            "installation_cost": round(installation_cost, 2),
            "net_savings_10_years": round(net_savings, 2)
        }

    # Return calculated solar savings
    return calculate_solar_savings(monthly_cost)
```

This function processes the user's monthly electricity cost and returns a detailed estimate of the solar panel system's benefits, including the number of panels required, installation costs, and net savings over ten years. For simplicity, we have made a few assumptions in the calculations, such as the average cost per kilowatt-hour and average sunlight hours. However, in a more advanced version of this AI agent, we could gather this information directly from the user, tailoring the estimates more precisely to their unique circumstances.

Step 3: Set Up State Management and Error Handling

Effective state management and error handling are crucial for building robust AI systems. Here, we define utilities to manage errors and maintain the conversation's state.

```
def handle_tool_error(state) -> dict:
    """
    Function to handle errors that occur during tool execution.

    Args:
        state (dict): The current state of the AI agent, which includes messages

    Returns:
        dict: A dictionary containing error messages for each tool that encounte
    """
    # Retrieve the error from the current state
    error = state.get("error")

    # Access the tool calls from the last message in the state's message history
    tool_calls = state["messages"][-1].tool_calls

    # Return a list of ToolMessages with error details, linked to each tool call
    return {
        "messages": [
            ToolMessage(
                content=f"Error: {repr(error)}\n please fix your mistakes.",  #
                tool_call_id=tc["id"],  # Associate the error message with the c
            )
            for tc in tool_calls  # Iterate over each tool call to produce indiv
        ]
    }

def create_tool_node_with_fallback(tools: list) -> dict:
    """
    Function to create a tool node with fallback error handling.

    Args:
        tools (list): A list of tools to be included in the node.

    Returns:
        dict: A tool node that uses fallback behavior in case of errors.
    """
    # Create a ToolNode with the provided tools and attach a fallback mechanism
    # If an error occurs, it will invoke the handle_tool_error function to manag
    return ToolNode(tools).with_fallbacks(
        [RunnableLambda(handle_tool_error)],  # Use a lambda function to wrap th
        exception_key="error"  # Specify that this fallback is for handling erro
    )
```

These functions ensure that any errors encountered during the tool's execution are handled gracefully, providing helpful feedback to the user.

Step 4: Define the State and Assistant Class

In this step, we'll define how the AI agent manages its state (the ongoing context of the conversation) and ensure it responds appropriately to the user's input and tool output.

To do this, we create a `State` class using Python's `TypedDict` to define the structure of the messages that will be passed around. The state will hold messages, including input from the user and output from the agent or tools.

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

Next, we create the Assistant class, which is responsible for running the AI agent, interacting with the tools, and managing the flow of the conversation. The Assistant invokes the tools, ensures they return appropriate results, and handles any re-prompts or errors that may occur during execution. Its core functionality includes invoking the Runnable, which defines the process of calling the LLM and tools like `compute_savings`, and then monitoring the results. If the agent fails to return a valid response or if a tool doesn't provide meaningful data, the Assistant re-prompts the user or requests clarification. It continues to loop through the Runnable until a valid output is obtained, ensuring smooth execution and effective responses.

```
class Assistant:
    def __init__(self, runnable: Runnable):
        # Initialize with the runnable that defines the process for interacting
        self.runnable = runnable

    def __call__(self, state: State):
        while True:
            # Invoke the runnable with the current state (messages and context)
            result = self.runnable.invoke(state)

            # If the tool fails to return valid output, re-prompt the user to cl
            if not result.tool_calls and (
                not result.content
                or isinstance(result.content, list)
                and not result.content[0].get("text")
            ):
                # Add a message to request a valid response
                messages = state["messages"] + [("user", "Respond with a real ou
                state = {**state, "messages": messages}
            else:
                # Break the loop when valid output is obtained
                break

        # Return the final state after processing the runnable
        return {"messages": result}
```

This setup is essential for maintaining the flow of conversation and ensuring that the assistant responds appropriately based on the context.

Step 5: Set Up the LLM with AWS Bedrock

In this step, we configure the Large Language Model (LLM) using AWS Bedrock, which will power the AI assistant's language capabilities. AWS Bedrock allows us to access advanced LLMs such as Anthropic's Claude. To interact with AWS services, you need to have your **AWS credentials configured**. This means you must either have your AWS credentials set in your environment (through the AWS CLI or environment variables) or use a credentials file that AWS SDKs can access. Without proper AWS configuration, the assistant won't be able to connect to AWS services like Bedrock for running the LLM.

```
def get_bedrock_client(region):
    return boto3.client("bedrock-runtime", region_name=region)

def create_bedrock_llm(client):
    return ChatBedrock(model_id='anthropic.claude-3-sonnet-20240229-v1:0', clien

llm = create_bedrock_llm(get_bedrock_client(region='us-east-1'))
```

This integration ensures that the assistant can effectively interpret and respond to user inputs.

Step 6: Define the Assistant's Workflow

Now that we have set up the LLM and tools, the next step is to define the AI assistant's workflow. This involves creating a template for the conversation, specifying the tools that the assistant will use, and configuring how the AI agent will respond to user input and trigger different functions (like calculating solar savings). The workflow is essentially the logic that governs how the assistant interacts with users, gathers information, and calls tools to provide results.

The first part of the workflow involves **creating a prompt template** that defines how the assistant will communicate with the user. The prompt helps guide the AI assistant in determining what to ask the user, how to respond based on the input, and when to trigger tools like `compute_savings`.

In this case, the assistant needs to ask the user for their monthly electricity cost to calculate solar panel savings. Here's how we define the conversation:

```
primary_assistant_prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            '''You are a helpful customer support assistant for Solar Panels Bel
            You should get the following information from them:
            - monthly electricity cost
            If you are not able to discern this info, ask them to clarify! Do no

            After you are able to discern all the information, call the relevant
            ''',
        ),
        ("placeholder", "{messages}"),
    ]
)
```

- **`system` message:** This message acts as a guide for the AI agent, instructing it to ask the user for their monthly electricity cost and not make guesses if the information is unclear. The assistant will keep prompting the user until it gathers the required data.
- **`placeholder`:** This placeholder allows the assistant to inject the messages from the conversation dynamically, creating an ongoing dialogue where the user's input influences the next steps.

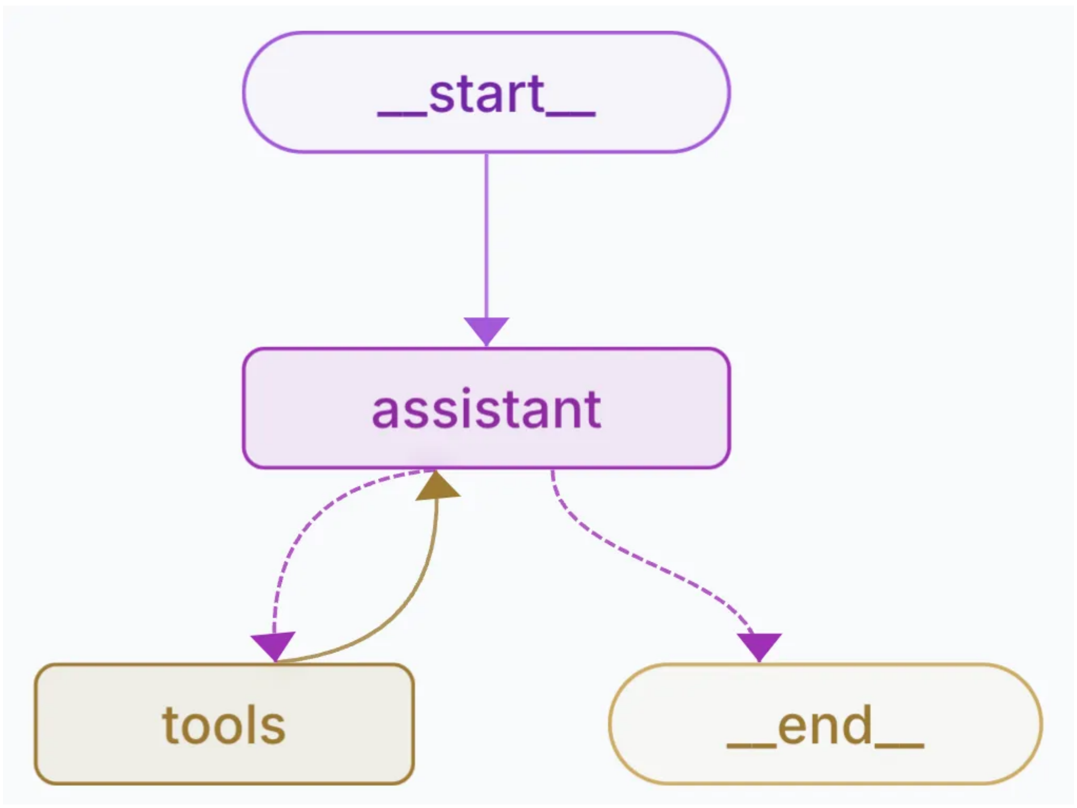
Next, we define the tools that the assistant will use during the interaction, with the primary tool being `compute_savings`, which calculates potential savings based on the user's monthly electricity cost. After specifying the tools in the list, we bind them to the assistant's workflow using the `llm.bind_tools()` method. This step ensures that the AI assistant can access and trigger the tools as needed during the conversation, creating a seamless interaction between the user and the assistant.

```
# Define the tools the assistant will use
part_1_tools = [
    compute_savings
]

# Bind the tools to the assistant's workflow
part_1_assistant_runnable = primary_assistant_prompt | llm.bind_tools(part_1_too
```

Step 7: Build the Graph Structure

In this step, we construct the **graph structure** for the AI assistant using LangGraph, which controls how the assistant processes user input, triggers tools, and moves between stages. The graph defines **nodes** for the core actions (like invoking the assistant and tool) and **edges** that dictate the flow between these nodes.



AI agent designed to calculate potential energy savings for solar panels.

Each **node** in LangGraph represents an operational step, such as interacting with the user or executing a tool. We define two key nodes for this AI assistant:

- **Assistant Node:** Manages the conversation flow, asking the user for their electricity cost and handling responses.
- **Tool Node:** Executes the tool (e.g., `compute_savings`) to calculate the user's solar panel savings.

```
builder = StateGraph(State)
builder.add_node("assistant", Assistant(part_1_assistant_runnable))
builder.add_node("tools", create_tool_node_with_fallback(part_1_tools))
```

Edges define how the flow moves between nodes. Here, the assistant starts the conversation, then transitions to the tool once the required input is collected, and returns to the assistant after the tool's execution.

```
builder.add_edge(START, "assistant")  # Start with the assistant
builder.add_conditional_edges("assistant", tools_condition)  # Move to tools aft
builder.add_edge("tools", "assistant")  # Return to assistant after tool executi
```

We use **MemorySaver** to ensure the graph retains the conversation state across different steps. This allows the assistant to remember the user's input, ensuring continuity in multi-step interactions.

```
memory = MemorySaver()
graph = builder.compile(checkpointer=memory)
```

Step 8: Running the Assistant

Finally, you can run the assistant by initiating the graph and starting the conversation.

```
# import shutil
import uuid
```

```
# Let's create an example conversation a user might have with the assistant
tutorial_questions = [
    'hey',
    'can you calculate my energy saving',
    "my montly cost is $100, what will i save"
]

thread_id = str(uuid.uuid4())

config = {
    "configurable": {
        "thread_id": thread_id,
    }
}

_printed = set()
for question in tutorial_questions:
    events = graph.stream(
        {"messages": ("user", question)}, config, stream_mode="values"
    )
    for event in events:
        _print_event(event, _printed)
```

Conclusion

By following these steps, you have successfully created an AI assistant using LangGraph that can calculate solar panel energy savings based on user inputs. This tutorial demonstrates the power of LangGraph in managing complex, multi-step processes and highlights how to leverage advanced AI tools to solve real-world challenges efficiently. Whether you're developing AI agents for customer support, energy management, or other applications, LangGraph provides the flexibility, scalability, and robustness needed to bring your ideas to life.

Interested in visualizing and testing this agent using LangGraph Studio? Check out my recent article and YouTube video:

LangGraph Studio: Visualizing and Testing AI Agents with LangChain

A guide to using LangGraph Studio for visualizing, testing, and developing AI agents with LangChain. Includes setup and…

medium.com