

2025

Lesson 12

LangGraph

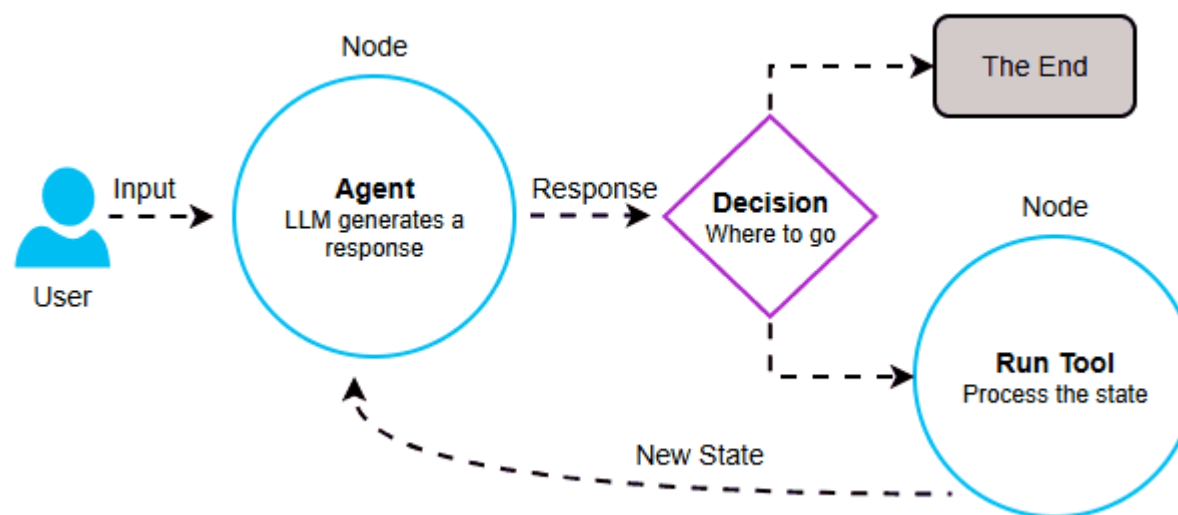




LangGraph and Graph

Introduction To LangGraph

- LangGraph is an advanced library built on top of LangChain.
- It is designed to enhanced Large Language Model (LLM) application by implementing cyclic computation capabilities i.e., a graph.
- LangChain allows the creation of Directed Acyclic Graph (DAGs) for linear workflows (sequential chain), LangGraph takes this a step further by enabling the addition of cycles.
- This addition is essential for developing complex, agent-like behaviours allowing LLM to continuously loop through a process, dynamically deciding what action to take next based on changing conditions.

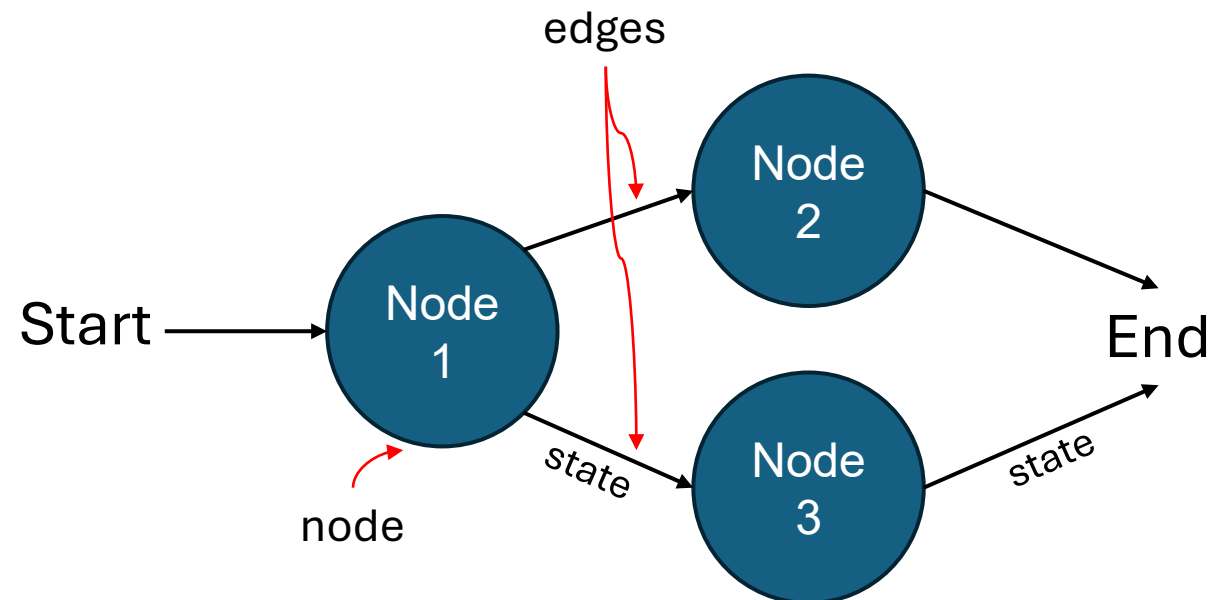


Graph

- A graph is a structured representation of **nodes** (or vertices) and **edges** (or connections) used to model workflows, data flows or relationship within a Large Language Model (LLM)-based system.
- The components in a graph are:
 - Nodes:
 - Represent discrete tasks, **functions** or prompts.
 - Example:
 - Send a prompt to an LLM.
 - Perform a specific computation or operation.
 - Edges
 - Represent the flow of data or control between nodes.
 - Define how the output of one node is passed as the input to another, creating a logical sequence of tasks.
 - Conditional edges are used when want to optionally route between nodes. It is implemented as a function.

Graph

- State
 - The state of the graph is defined using a state schema.
 - It is defined either using **TypedDict** class from Python's typing module or **Pydantic** model.
 - Maintains and updates the memory as the process advances.
 - Contains relevant information from earlier steps for decision making.
- Directed Graph
 - LangGraph workflows are typically represented as directed graphs, where the direction of edges indicates the flow of execution or data (e.g., from one LLM query to a subsequent operation).



LangGraph Architecture

- In a LangGraph-style architecture, **nodes** represent individual tasks or operations.
- **Edges** define the flow of information between these tasks. Every node is a function.
- Each node in the graph encapsulates a specific piece of functionality. It takes an input, performs a computation or action and produces an output.
- Edge as data flow. The connections between nodes represent the flow of data or the results from one function to the next.
- A **state** schema serves as a blueprint or structure for defining and validating the data that a workflow state holds during execution. It ensures consistency and prevents errors by enforcing rules on the data that each state in the workflow processes.

Schema Definitions

Pydantic

```
from pydantic import BaseModel
from typing import List, Optional

class StateSchema(BaseModel):
    user_query: str
    current_step: str
    results: List[str] = []
    error: Optional[str] = None
```

- user_query: The input provided by the user (required, type str)
- current_step: Tracks which node in the graph is currently processing the data
- results: A list to accumulate intermediate or final results (**default** is an empty list)
- error: An optional field to handle errors

Vs

TypedDict

```
from typing import TypedDict, List,
Optional

class StateSchema(TypedDict):
    user_query: str
    current_step: str
    results: List[str]
    error: Optional[str]
```

- user_query: A required string representing the user's input
- current_step: A string indicating the current stage of the workflow
- results: A list of strings to store intermediate or final results
- error: An optional string to capture error messages

Building A Simple Graph

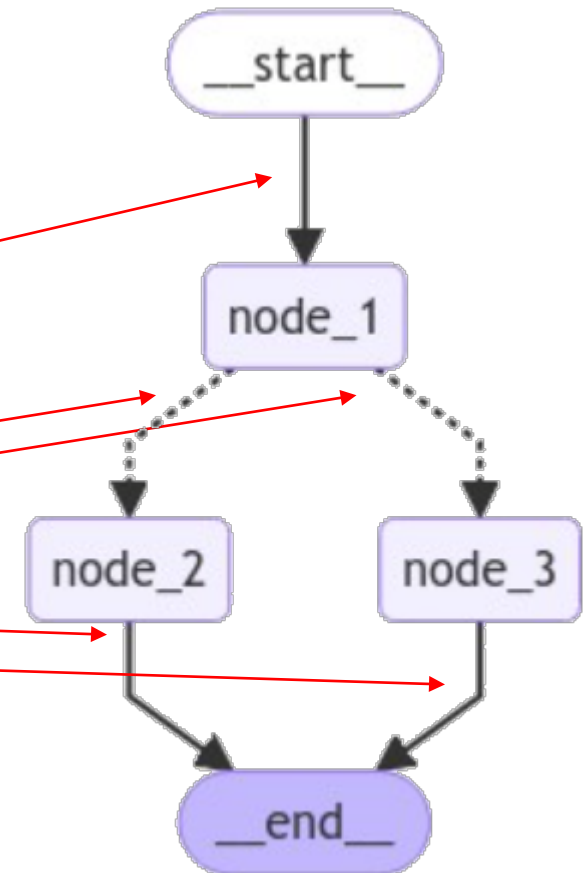
```
from langgraph.graph import StateGraph, START, END

# Build graph
builder = StateGraph(State)
builder.add_node("node_1", node_1)
builder.add_node("node_2", node_2)
builder.add_node("node_3", node_3)

# Logic
builder.add_edge(START, "node_1")
builder.add_conditional_edges("node_1", decide_mood)
builder.add_edge("node_2", END)
builder.add_edge("node_3", END)

# Add
graph = builder.compile()

# View
display(Image(graph.get_graph().draw_mermaid_png()))
```



Build A Simple Graph

```
import random
from typing import Literal

def decide_mood(state) -> Literal["node_2", "node_3"]:

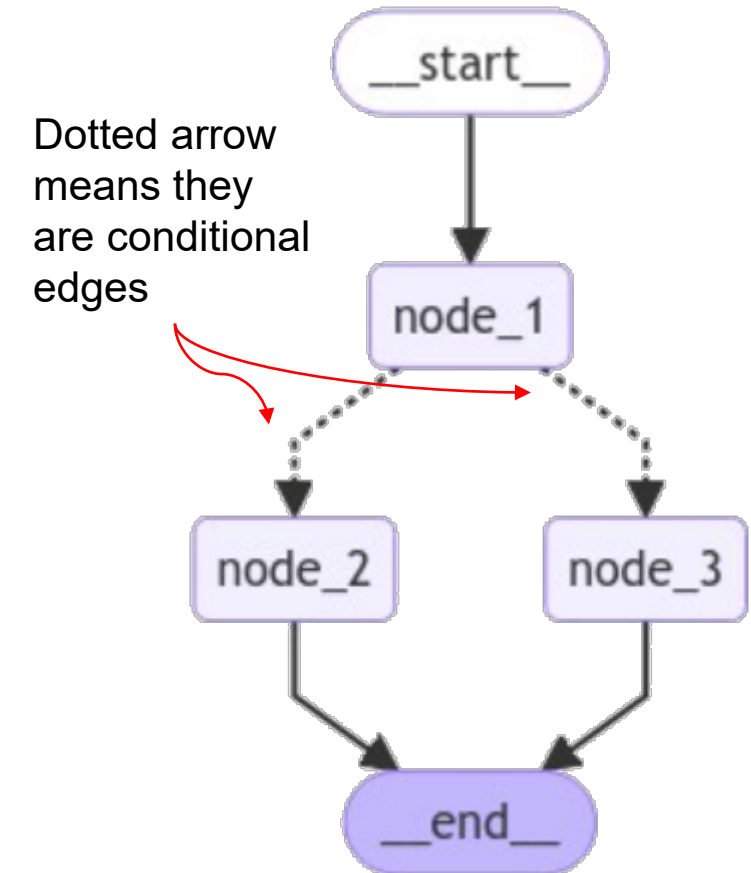
    # Often, we will use state to decide on the next node to visit
    user_input = state['graph_state']

    # Here, let's just do a 50 / 50 split between nodes 2, 3
    if random.random() < 0.5:

        # 50% of the time, we return Node 2
        return "node_2"

    # 50% of the time, we return Node 3
    return "node_3"
```

Arbitrarily example of
a conditional node





Agent

Agent

- An agent is an autonomous entity that can execute tasks, interact with various components, and make decisions based on the given instructions or data.
 - Agents can perform tasks independently, such as querying data, executing code, or interacting with APIs.
 - An agent typically works to achieve a specific objective, such as answering a question, processing input, or generating content.
 - Based on the data they receive and the rules they are programmed with, agents can decide on the next steps in a workflow.
- The ReAct agent is an advanced AI agent design pattern that integrates **Reasoning and Acting** in a unified workflow.
 - Reasoning: The agent generates thoughts to reason through a problem step-by-step.
 - Acting: The agent performs actions to interact with the environment or retrieve necessary information.

Agentic AI

- A concept first introduced by Andrew Ng (a prominent figure in AI).
- Agentic AI surpasses the reactive capabilities of Generative AI by acting autonomously, planning ahead and adapting to user's needs.
- Key features of Agentic AI
 - Autonomy:
 - Operates independently, making decisions based on goals, constraints, and environmental inputs.
 - Requires minimal human oversight for task execution.
 - Goal-Oriented Behaviour:
 - Works toward specific objectives or goals defined by the user or its programming.
 - Breaks down complex tasks into smaller, manageable steps.
 - Contextual Understanding
 - Uses natural language understanding (NLU) to process user instructions or environmental data.
 - Maintains context over time, especially in multi-step processes or dialogues.

Agentic AI

- Reasoning and Problem-Solving
 - Employs logic, heuristics, or LLMs for decision-making.
 - Can evaluate multiple solutions or approaches to select the most effective one.
- Tool Integration and Action Execution
 - Interacts with external tools, APIs, databases, and environments.
 - Capable of performing actions like querying a database, making an API call, or writing and executing code.
- Dynamic Planning and Adaptation
 - Plans actions dynamically, updating its strategy based on real-time feedback or new information.
 - Adapts to unexpected challenges or changes in the environment.
- Multi-agent collaboration
 - Multiple AI agents collaborate to achieve better outcomes than a single agent could.



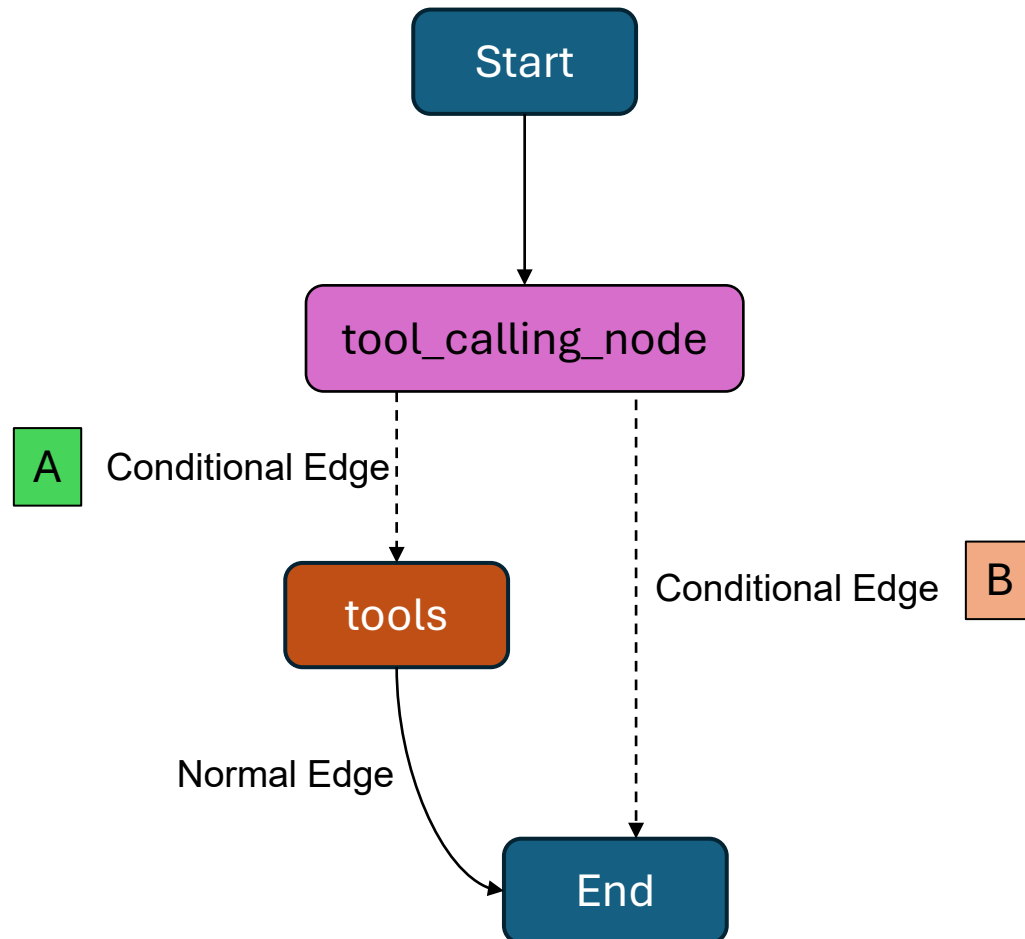
Router

Router

- Router is to route logic.
- For example, to create a helpful assistant, the assistant knows when to use tools and when to simply respond.
- It is like a worker at desk – sometimes they need to use tool to help complete the tasks (like a calculator or reference guide) and other times, he/she can just answer directly.
- Main components are:
 - The model decides whether a tool is required.
 - A special node called ToolNode that knows how to use different tools when needed.
 - A routing system that looks at each response and decide:
 - Let's use the tool and then continue the task execution.
 - No tool is needed. Continue other tasks till end.

Tool Calling Node

- Tool calling node decides whether to go with route A or route B.



```

# Tool node
def tool_calling_node(state: MessagesState):
    return {"messages": [model_with_tools.invoke(state["messages"])]}

builder = StateGraph(MessagesState)

builder.add_node("tool_calling_node", tool_calling_node)
builder.add_node("tools", ToolNode([addition]))

# Set the entry point - start with the tool calling node
builder.add_edge(START, "tool_calling_node")

builder.add_conditional_edges(
    "tool_calling_node",
    tools_condition
)

builder.add_edge("tools", END)

graph = builder.compile()
  
```


Custom Node Type

- A user defined component in the workflow chain.
- Unlike pre-build nodes, a custom node implements your own logic, tailored to your application needs which implement complex agent logic.
- There are two primary ways to include custom logic:
 - Define a Python function and add it as a node to the graph.
 - Use a “custom” node type.

Scenario	Python Function Node	Custom Node
Simple logic that doesn't require additional features	<input checked="" type="checkbox"/> Preferred	<input type="checkbox"/> Not Recommend
Workflow with potential for debugging or error handling	<input type="checkbox"/> Limited support	<input checked="" type="checkbox"/> Preferred
Need to reuse the function independently	<input checked="" type="checkbox"/> Easy to reuse	<input type="checkbox"/> Tie to the graph node

Custom Node Type

- LangGraph allows user to create custom node types to implement complex agent logic.

```
class MyCustomNode:
    def __init__(self, llm):
        self.llm = llm

    def __call__(self, state):
        # Implement your custom logic here
        # Access the state and perform actions
        messages = state["messages"]
        response = self.llm.invoke(messages)
        return {"messages": [response]}

graph_builder = StateGraph(State)

llm = ChatOpenAI(model="...")

custom_node = MyCustomNode(llm)

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



Tip

Tips




- The output messages from LangChain/LangGraph can be difficult to read when debugging.
- Use a Python/JSON formatter beautifier to help to “beautify” the text.


```
event






{'messages': [HumanMessage(content='Add 3 and 4. Multiply the output by 2. Divide the output by 0.', additional_kwargs={}, response_metadata={}, id='2363594f-4486-4069-bd31-5d7fcb4ceebf'),
  HumanMessage(content='Ignore the last instruction. Do this: Add 3 and 4. Multiply the output by 2. Divide the output by 10.', additional_kwargs={}, response_metadata={}, id='d45f5303-b5e5-4396-944d-d14e9e5173ca'),
  AIMessage(content='', additional_kwargs={'tool_calls': [{'id': 'call_veXzFk61lMiYXxzgKArK03F0', 'function': {'arguments': '{"a": 3, "b": 4}', 'name': 'add'}, 'type': 'function'}, {'id': 'call_lwebKLVZUr44hPV6EhbQZJcV', 'function': {'arguments': '{"a": 7, "b": 2}', 'name': 'multiply'}, 'type': 'function'}, {'id': 'call_IztbVy1xfEYp6aLITRkGiCle', 'function': {'arguments': '{"a": 14, "b": 10}', 'name': 'divide'}, 'type': 'function'}], 'refusal': None}, response_metadata={'token_usage': {'completion_tokens': 67, 'prompt_tokens': 197, 'total_tokens': 264, 'completion_tokens_details': {'accepted_prediction_tokens': 0, 'audio_tokens': 0, 'reasoning_tokens': 0, 'rejected_prediction_tokens': 0}, 'prompt_tokens_details': {'audio_tokens': 0, 'cached_tokens': 0}}, 'model_name': 'gpt-4o-mini-2024-07-18', 'system_fingerprint': 'fp_6fc10e10eb', 'finish_reason': 'tool_calls', 'logprobs': None}, id='run-ee57ff7d-37f6-4412-acc4-18a6b45396c4-0', tool_calls=[{'name': 'add', 'args': {'a': 3, 'b': 4}, 'id': 'call_veXzFk61lMiYXxzgKArK03F0', 'type': 'tool_call'}, {'name': 'multiply', 'args': {'a': 7, 'b': 2}, 'id': 'call_lwebKLVZUr44hPV6EhbQZJcV', 'type': 'tool_call'}, {'name': 'divide', 'args': {'a': 14, 'b': 10}, 'id': 'call_IztbVy1xfEYp6aLITRkGiCle', 'type': 'tool_call'}], usage_metadata={'input_tokens': 197, 'output_tokens': 67, 'total_tokens': 264, 'input_token_details': {'audio': 0, 'cache_read': 0}, 'output_token_details': {'audio': 0, 'reasoning': 0}}),
  ToolMessage(content='7', name='add', id='695c8b55-8b78-431b-9295-a6a1d2d5cc05', tool_call_id='call_veXzFk61lMiYXxzgKArK03F0'),
  ToolMessage(content='14', name='multiply', id='d8aea04a-a36c-4369-b78a-170d5489fd15', tool_call_id='call_lwebKLVZUr44hPV6EhbQZJcV'),
  ToolMessage(content='1.4', name='divide', id='113a5655-0f67-4937-b2f1-bb3bd9a69c1a', tool_call_id='call_IztbVy1xfEYp6aLITRkGiCle'),
  AIMessage(content='The results are as follows:\n\n1. The sum of 3 and 4 is 7.\n2. Multiplying the sum (7) by 2 gives 14.\n3. Dividing this result (14) by 10 gives 1.4. \n\nSo, the final output is **1.4**.', additional_kwargs={'refusal': None}, response_metadata={'token_usage': {'completion_tokens': 68, 'prompt_tokens': 285, 'total_tokens': 353, 'completion_tokens_details': {'accepted_prediction_tokens': 0, 'audio_tokens': 0, 'reasoning_tokens': 0, 'rejected_prediction_tokens': 0}, 'prompt_tokens_details': {'audio_tokens': 0, 'cached_tokens': 0}}, 'model_name': 'gpt-4o-mini-2024-07-18', 'system_fingerprint': 'fp_6fc10e10eb', 'finish_reason': 'stop', 'logprobs': None}, id='run-49c13c60-73ad-475e-a578-fa4d482b882f-0', usage_metadata={'input_tokens': 285, 'output_tokens': 68, 'total_tokens': 353, 'input_token_details': {'audio': 0, 'cache_read': 0}, 'output_token_details': {'audio': 0, 'reasoning': 0}})]]
```

Python/JSON Formatter Beautifier

<https://codebeautify.org/python-formatter-beautifier>


JSON Formatter XML Formatter Calculators JSON Beautifier Recent Links Sitemap Favs 
Login 

Python Formatter
 Add to Fav
 New
Save & Share


Sample     


```


1 [{"messages": [HumanMessage(content='Add 3 and 4. Multiply the output by 2.
  Divide the output by 0.', additional_kwargs={}, response_metadata={}, id
  ='2363594f-4486-4069-bd31-5d7fcb4ceeef'),
2 HumanMessage(content='Ignore the last instruction. Do this: Add 3 and 4.
  Multiply the output by 2. Divide the output by 10.', additional_kwargs
  ={}, response_metadata={}, id='d45f5303-b5e5-4396-944d-d14e9e5173ca'),
3 AIMessage(content='', additional_kwargs={'tool_calls': [{'id':
  'call_veXzFk61lMiYXxzgKArK03F0', 'function': {'arguments': '{"a": 3,
  "b": 4}', 'name': 'add'}, 'type': 'function'}, {'id':
  'call_lwebKLVZUr44hPV6EhbQZJcV', 'function': {'arguments': '{"a": 7,
  "b": 2}', 'name': 'multiply'}, 'type': 'function'}, {'id':
  'call_IztbVy1xfEYp6aLITRkGiCle', 'function': {'arguments': '{"a": 14,
  "b": 10}', 'name': 'divide'}, 'type': 'function'}], 'refusal': None},
  response_metadata={'token_usage': {'completion_tokens': 67,
  'prompt_tokens': 197, 'total_tokens': 264, 'completion_tokens_details':
  {'accepted_prediction_tokens': 0, 'audio_tokens': 0, 'reasoning_tokens':
  0, 'rejected_prediction_tokens': 0}, 'prompt_tokens_details':
  {'audio_tokens': 0, 'cached_tokens': 0}}, 'model_name': 'gpt-4o-mini-
  2024-07-18', 'system_fingerprint': 'fp_6fc10e10eb', 'finish_reason':
  'tool_calls', 'logprobs': None}, id='run-ee57ff7d-37f6-4412-acc4-
  -18a6b45396c4-0', tool_calls=[{'name': 'add', 'args': {'a': 3, 'b': 4},
  'id': 'call_veXzFk61lMiYXxzgKArK03F0', 'type': 'tool_call'}, {'name':
  'multiply', 'args': {'a': 7, 'b': 2}, 'id':
  'call_lwebKLVZUr44hPV6EhbQZJcV', 'type': 'tool_call'}, {'name':
  'divide', 'args': {'a': 14, 'b': 10}, 'id':
  'call_IztbVy1xfEYp6aLITRkGiCle', 'type': 'tool_call'}], usage_metadata
  ={'input_tokens': 197, 'output_tokens': 67, 'total_tokens': 264,
  'input_token_details': {'audio': 0, 'cache_read': 0},
  'output_token_details': {'audio': 0, 'reasoning': 0}}}],




```

Ln: 7 Col: 916 size: 3.13 KB

File  URL

Python Version 

 Download

Output   

```

1 {
2   "messages": [
3     HumanMessage(
4       content="Add 3 and 4. Multiply the output by 2. Divide the
5         output by 0.",
6       additional_kwargs={},
7       response_metadata={},
8       id="2363594f-4486-4069-bd31-5d7fcb4ceeef",
9     ),
10    HumanMessage(
11      content="Ignore the last instruction. Do this: Add 3 and 4.
12        Multiply the output by 2. Divide the output by 10.",
13      additional_kwargs={},
14      response_metadata={},
15      id="d45f5303-b5e5-4396-944d-d14e9e5173ca",
16    ),
17    AIMessage(
18      content="",
19      additional_kwargs={
20        "tool_calls": [
21          {
22            "id": "call_veXzFk61lMiYXxzgKArK03F0",
23            "function": {"arguments": '{"a": 3, "b": 4}',
24              "name": "add"},
25            "type": "function",
26          },

```

Ln: 140 Col: 0 size: 5.76 KB



Activity

Activity

- Activities to code in LangGraph. LangGraph is a framework designed to model workflow as directed graph.
- In a LangGraph-style architecture, nodes represent individual tasks or operations.
- Edges define the flow of information between these tasks. Every node is a function.
- Each node in the graph encapsulates a specific piece of functionality. It takes an input, performs a computation or action and produces an output.
- You will be creating:
 - Basic Chatbot
 - Tools whenever you want a model to interact with external systems
 - Agent
 - Human intervention

Summary

- To build a graph, here are the steps:
 1. First define the State of the graph.
 - The State schema serves as input schema for all Nodes and Edges in the graph, You can use either TypedDict class or Pydantic library to define the schema.
 2. Create the node.
 - The nodes are just Python functions or it can be a custom node.
 3. Connect the nodes using Edges.
 - Normal edges are used when it always go from, for example Node A to Node B. Conditional edges are used to route between 2 or more nodes.
 4. Construct the graph from StateGraph class.
 - Initialise the StateGraph with the State class defined earlier.
 - Build the graph by compiling it.
 5. Invoke the graph
 - When invoke is called, the graph starts execution from the START node.
 - The execution continues until it reaches the END node.

Reference

- LangChain – LangGraph Quick Start
<https://langchain-ai.github.io/langgraph/tutorials/introduction/>
- LangChain Academy – LangGraph
<https://academy.langchain.com/courses/intro-to-langgraph>
- LangGraph Glossary
https://langchain-ai.github.io/langgraph/concepts/low_level/#multiple-schemas

Thank you!