Analytics Vidhya

Simple Agentic Graph and State Management in LangGraph

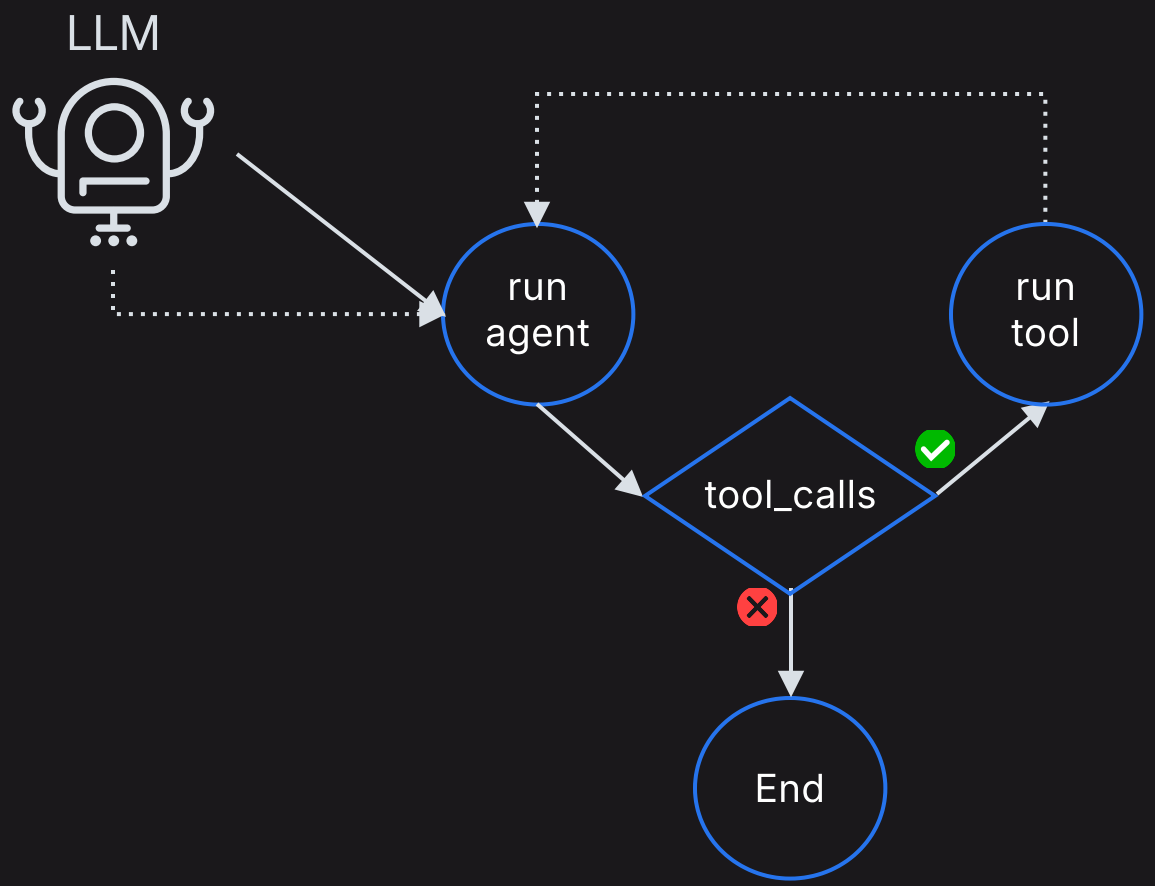
Dipanjan Sarkar

Head of Community & Principal AI Scientist at Analytics Vidhya

Google Developer Expert - ML & Cloud Champion Innovator

Published Author

LangGraph Agent Graph Components



- LangGraph treats Agent workflows as a cyclical Graph structure
- Main features
  - Nodes: Functions or LangChain Runnable objects such as tools.
  - Edges: Specify directional paths between nodes
  - Stateful Graphs: Manage and update state objects while processing data through nodes.
- LangGraph leverages this to facilitate cyclical LLM call executions with state persistence which is often required for AI Agents

LangGraph models agent workflows as stateful graphs

Stateful means all the steps executed are stored in their state to keep track of the overall execution flow and all variables, data, tool calls, and responses (like key:value pairs in a python dictionary).

Agent State and State Schema in LangGraph

States

- **States** represent the shared context or data passed and updated as the agent (graph) progresses through steps (nodes).
- Encapsulates all necessary information for the graph to function effectively, including:
  - Input data
  - Intermediate results
  - Execution metadata

```python
from langgraph.graph import StateGraph

# Define the schema of the graph state
class State(TypedDict):
    messages: str

# state schema can also have multiple variables
class State(TypedDict):
    data: str
    counter: int


# Initialize the StateGraph with the defined
state schema
graph_builder = StateGraph(State)
```

Agent State and State Schema in LangGraph

State Schema

- **State Schema** defines the overall schema of the data variables which will be passed throughout the agent (graph) lifecycle.
- It can be any Python type but is typically a TypedDict or Pydantic BaseModel.
- You can have multiple schemas for specific nodes in a graph also.

```python
from langgraph.graph import StateGraph

# Define the schema of the graph state
class State(TypedDict):
    messages: str

# state schema can also have multiple variables
class State(TypedDict):
    data: str
    counter: int


# Initialize the StateGraph with the defined
state schema
graph_builder = StateGraph(State)
```

StateGraph in LangGraph

- In LangGraph, the StateGraph class serves as the foundational structure for defining and managing the flow of data and operations within a graph-based workflow.
- It allows us to model simple and complex agentic systems as interconnected nodes, each performing specific functions and updating a shared state.


```python
from langgraph.graph import StateGraph
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END

# define state schema
class State(TypedDict):
    messages: str

# define node functionality
def node_1(state: State) -> State:
    print("---Node 1---")
    state = state['messages']
    return {"messages": "Hello this is node 1"}

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

# connect nodes
builder.add_edge(START, "node_1")
builder.add_edge("node_1", END)

# compile graph
graph = builder.compile()

# run graph
graph.invoke({"messages": "Hi, how are you?."})

## OUTPUT
---Node 1---
{'messages': 'Hello this is node 1'}
```

StateGraph in LangGraph

Key Features of StateGraph

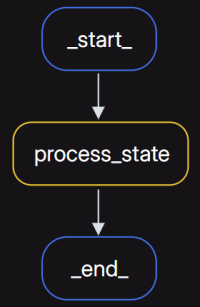
- **State Management**
  - Centralizes the state that nodes can access and modify, ensuring consistent data flow throughout the graph.
- **Node Definition**
  - Enables the addition of various nodes, each representing a discrete operation or function within the workflow.
- **Edge Specification**
  - Allows for the definition of directed edges between nodes, establishing the sequence of operations.
- **Conditional Logic**
  - Supports conditional edges, facilitating dynamic decision-making paths based on the current state.


```python
from langgraph.graph import StateGraph
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END

# define state schema
class State(TypedDict):
    messages: str

# define node functionality
def node_1(state: State) -> State:
    print("---Node 1---")
    state = state['messages']
    return {"messages": "Hello this is node 1"}

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

# connect nodes
builder.add_edge(START, "node_1")
builder.add_edge("node_1", END)

# compile graph
graph = builder.compile()

# run graph
graph.invoke({"messages": "Hi, how are you?."})

## OUTPUT
---Node 1---
{'messages': 'Hello this is node 1'}
```

Example of State Schema and StateGraph in LangGraph

State Definition

- The State class defines the schema for the shared state used throughout the workflow.
- It includes two fields:
  - data: A string field for storing data.
  - counter: An integer field to track operations or interactions.

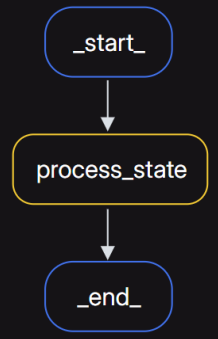



```python
from langgraph.graph import StateGraph, START, END

# Define the state structure
class State(TypedDict):
    data: str
    counter: int

# Initialize the StateGraph with the state schema
graph_builder = StateGraph(State)

# Add nodes and edges to your graph
# Define a node that processes the state
def process_state(state: State) -> State:
    print(f"""Current data: {state['data']},
          Counter: {state['counter']}""")
    return {"data": state["data"].upper(),
            "counter": state["counter"] + 1}

# Add the node to the graph
graph_builder.add_node("process_state", process_state)
graph_builder.add_edge(START, "process_state")
graph_builder.add_edge("process_state", END)

# Compile and run the graph
graph = graph_builder.compile()
initial_state = {"data": "my data", "counter": 0}
result = graph.invoke(initial_state)
# Print the result
print("Final State:", result)

## OUTPUT
Current data: my data, Counter: 0
Final State: {'data': 'MY DATA', 'counter': 1}
```

Example of State Schema and StateGraph in LangGraph

Node Function

- The process_state node function processes the current state, printing its values, converting data to uppercase, and incrementing the counter.

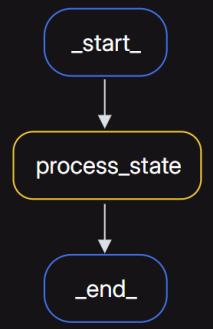



```python
from langgraph.graph import StateGraph, START, END

# Define the state structure
class State(TypedDict):
    data: str
    counter: int

# Initialize the StateGraph with the state schema
graph_builder = StateGraph(State)

# Add nodes and edges to your graph
# Define a node that processes the state
def process_state(state: State) -> State:
    print(f"""Current data: {state['data']},
            Counter: {state['counter']}""")
    return {"data": state["data"].upper(),
            "counter": state["counter"] + 1}

# Add the node to the graph
graph_builder.add_node("process_state", process_state)
graph_builder.add_edge(START, "process_state")
graph_builder.add_edge("process_state", END)

# Compile and run the graph
graph = graph_builder.compile()
initial_state = {"data": "my data", "counter": 0}
result = graph.invoke(initial_state)
# Print the result
print("Final State:", result)

## OUTPUT
Current data: my data, Counter: 0
Final State: {'data': 'MY DATA', 'counter': 1}
```

Example of State Schema and StateGraph in LangGraph

StateGraph Integration

- The StateGraph is initialized with the State schema.
- Nodes and edges are added to define the workflow, starting with START, passing through process_state, and ending with END.

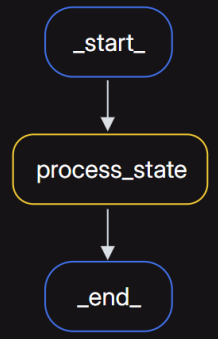



```python
from langgraph.graph import StateGraph, START, END

# Define the state structure
class State(TypedDict):
    data: str
    counter: int

# Initialize the StateGraph with the state schema
graph_builder = StateGraph(State)

# Add nodes and edges to your graph
# Define a node that processes the state
def process_state(state: State) -> State:
    print(f"""Current data: {state['data']},
          Counter: {state['counter']}""")
    return {"data": state["data"].upper(),
            "counter": state["counter"] + 1}

# Add the node to the graph
graph_builder.add_node("process_state", process_state)
graph_builder.add_edge(START, "process_state")
graph_builder.add_edge("process_state", END)

# Compile and run the graph
graph = graph_builder.compile()
initial_state = {"data": "my data", "counter": 0}
result = graph.invoke(initial_state)
# Print the result
print("Final State:", result)

## OUTPUT
Current data: my data, Counter: 0
Final State: {'data': 'MY DATA', 'counter': 1}
```

Example of State Schema and StateGraph in LangGraph

Execution

- The graph is invoked with an initial state. The graph processes the state through the defined nodes, updating and returning the final state.




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

# Define the state structure
class State(TypedDict):
    data: str
    counter: int

# Initialize the StateGraph with the state schema
graph_builder = StateGraph(State)

# Add nodes and edges to your graph
# Define a node that processes the state
def process_state(state: State) -> State:
    print(f"""Current data: {state['data']},
          Counter: {state['counter']}""")
    return {"data": state["data"].upper(),
            "counter": state["counter"] + 1}

# Add the node to the graph
graph_builder.add_node("process_state", process_state)
graph_builder.add_edge(START, "process_state")
graph_builder.add_edge("process_state", END)

# Compile and run the graph
graph = graph_builder.compile()
initial_state = {"data": "my data", "counter": 0}
result = graph.invoke(initial_state)
# Print the result
print("Final State:", result)

## OUTPUT
Current data: my data, Counter: 0
Final State: {'data': 'MY DATA', 'counter': 1}
```

Updating States in LangGraph with Reducers

- In LangGraph, **reducers** are functions that define how updates are applied to specific keys within the State.
- By default, updates to a state key replace its existing value.
- However, when dealing with sequences like message lists, it's often desirable to append new items rather than overwrite the entire list.
- This is where the add_messages reducer comes into play.

```python
from typing import Annotated
from langgraph.graph.message import add_messages

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

def node_1(state: State) -> State:
    print("---Node 1---")
    state = state['messages']
    return {"messages": "Hello this is node 1"}

def final_node(state):
    print("---Node 2---")
    state = state['messages']
    return {"messages": "Hello this is the final node"}

# Build graph
builder = StateGraph(State)
builder.add_node("node_1", node_1)
builder.add_node("node_2", final_node)
builder.add_edge(START, "node_1")
builder.add_edge("node_1", "node_2")
builder.add_edge("node_2", END)

# Compile and run
graph = builder.compile()
graph.invoke({"messages": "Hi, how are you?."})

## OUTPUT
---Node 1---
---Node 2---
{'messages': [HumanMessage(content='Hi, how are you?.', ...),
              HumanMessage(content='Hello this is node 1', ...),
              HumanMessage(content='Hello this is the final node', ...)]}
```

Updating States in LangGraph with Reducers

add_messages Reducer

- The add_messages reducer is designed to handle lists of messages within the state.
- It ensures that new messages are appended to the existing list, maintaining the conversation history.
- Additionally, it manages message IDs to prevent duplication and allows for the removal of specific messages when needed.

```python
from typing import Annotated
from langgraph.graph.message import add_messages

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

def node_1(state: State) -> State:
    print("---Node 1---")
    state = state['messages']
    return {"messages": "Hello this is node 1"}

def final_node(state):
    print("---Node 2---")
    state = state['messages']
    return {"messages": "Hello this is the final node"}

# Build graph
builder = StateGraph(State)
builder.add_node("node_1", node_1)
builder.add_node("node_2", final_node)
builder.add_edge(START, "node_1")
builder.add_edge("node_1", "node_2")
builder.add_edge("node_2", END)

# Compile and run
graph = builder.compile()
graph.invoke({"messages": "Hi, how are you?."})

## OUTPUT
---Node 1---
---Node 2---
{'messages': [HumanMessage(content='Hi, how are you?.', ...),
              HumanMessage(content='Hello this is node 1', ...),
              HumanMessage(content='Hello this is the final node', ...)]}
```

Updating States in LangGraph with Reducers

Implementing add_messages in State

- To utilize the add_messages reducer, you define your state schema and annotate the messages key accordingly as depicted

```python
from typing import Annotated
from langgraph.graph.message import add_messages

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

def node_1(state: State) -> State:
    print("---Node 1---")
    state = state['messages']
    return {"messages": "Hello this is node 1"}

def final_node(state):
    print("---Node 2---")
    state = state['messages']
    return {"messages": "Hello this is the final node"}

# Build graph
builder = StateGraph(State)
builder.add_node("node_1", node_1)
builder.add_node("node_2", final_node)
builder.add_edge(START, "node_1")
builder.add_edge("node_1", "node_2")
builder.add_edge("node_2", END)

# Compile and run
graph = builder.compile()
graph.invoke({"messages": "Hi, how are you?."})

## OUTPUT
---Node 1---
---Node 2---
{'messages': [HumanMessage(content='Hi, how are you?.', ...),
              HumanMessage(content='Hello this is node 1', ...),
              HumanMessage(content='Hello this is the final node', ...)]}
```

Thanks