Analytics Vidhya

Essential LangGraph Components and Functions

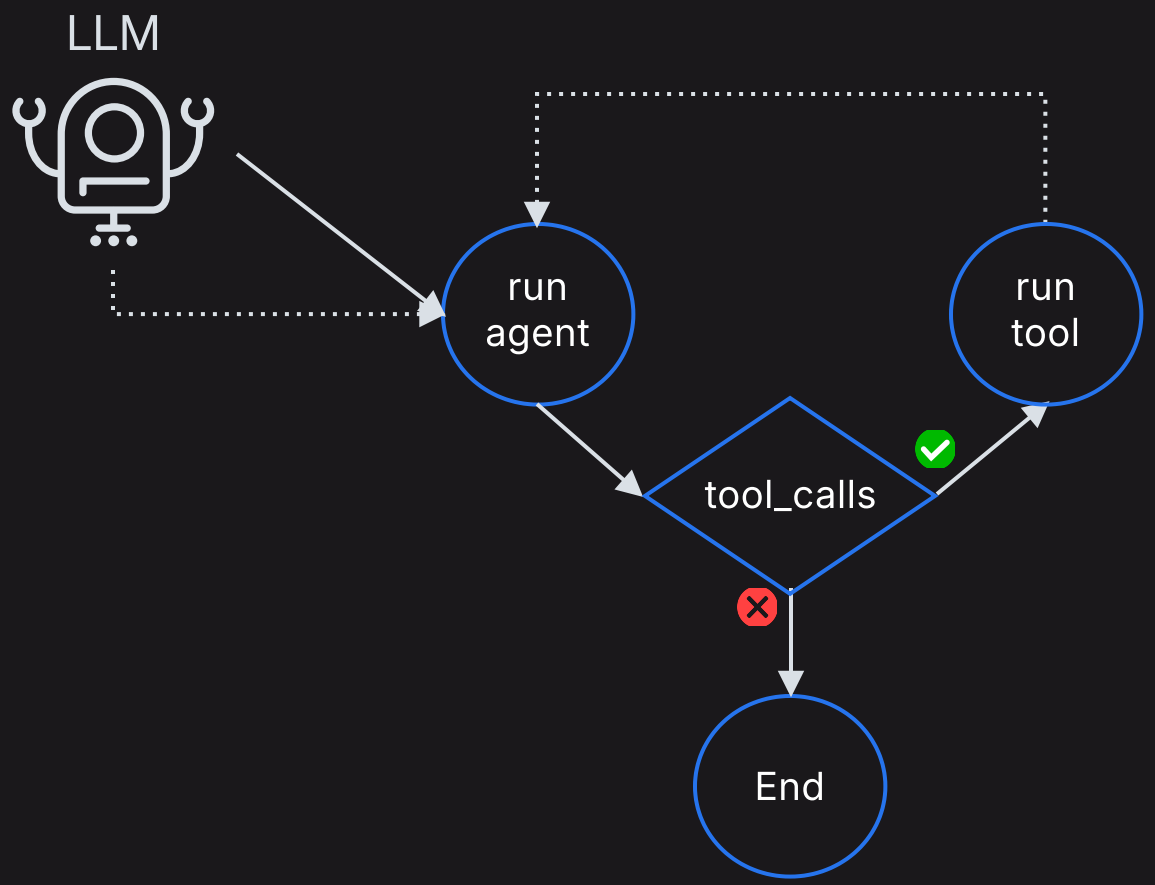
Dipanjan Sarkar

Head of Community & Principal AI Scientist at Analytics Vidhya

Google Developer Expert - ML & Cloud Champion Innovator

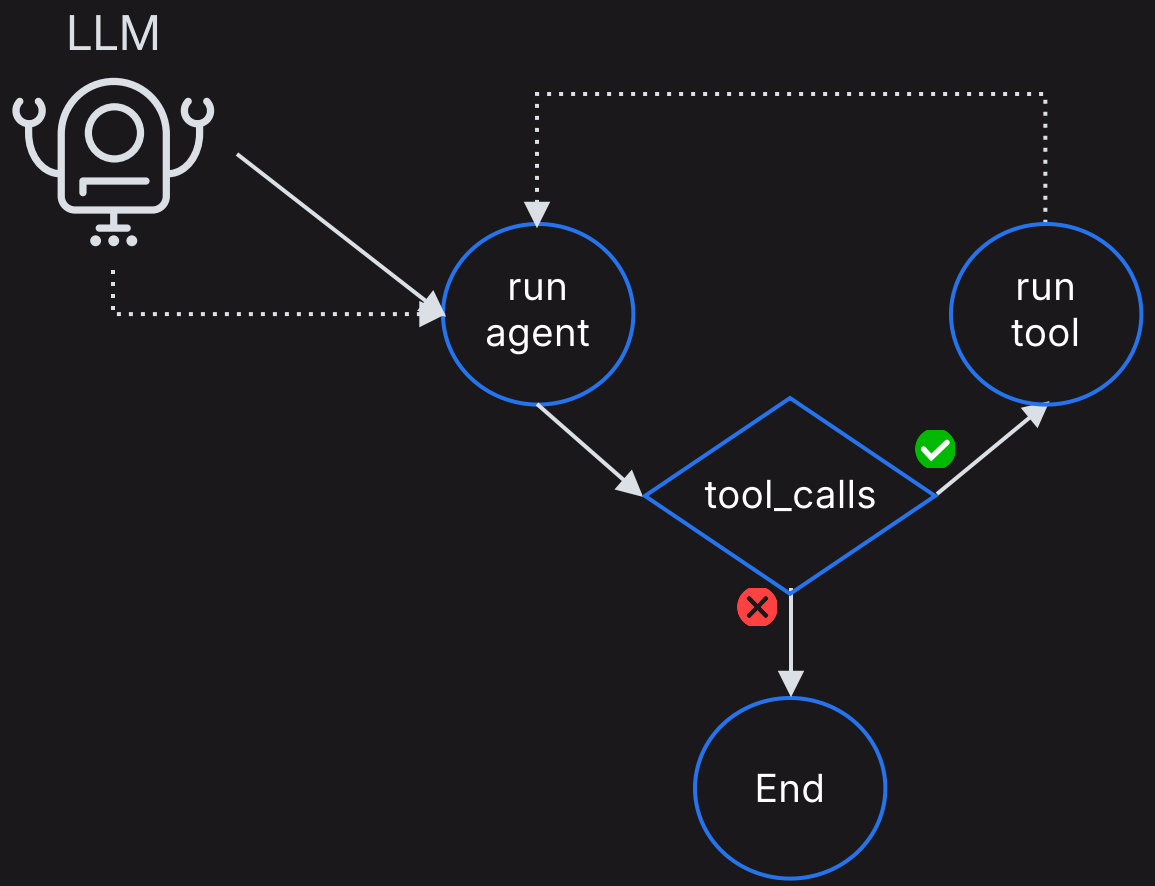
Published Author

LangGraph models Agentic Systems as Graphs



- LangGraph, built on top of LangChain, facilitates the creation of cyclical graphs essential for developing AI agents powered by LLMs.
- Its interface is inspired by the widely-used NetworkX library.
- It enables the coordination and checkpointing of multiple chains (or actors) through cyclic computational steps.

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

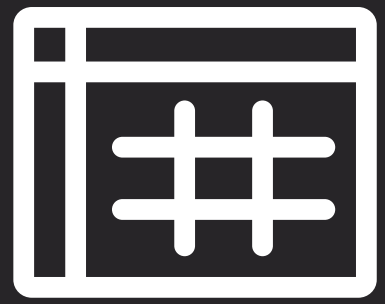
LangGraph models agent workflows as stateful graphs

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 (e.g. as key : value pairs in a python dictionary).

Key Components in LangGraph Agentic Systems

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
- Can be any Python type, but typically a **TypedDict** or **Pydantic BaseModel**.



Key Components in LangGraph Agentic Systems

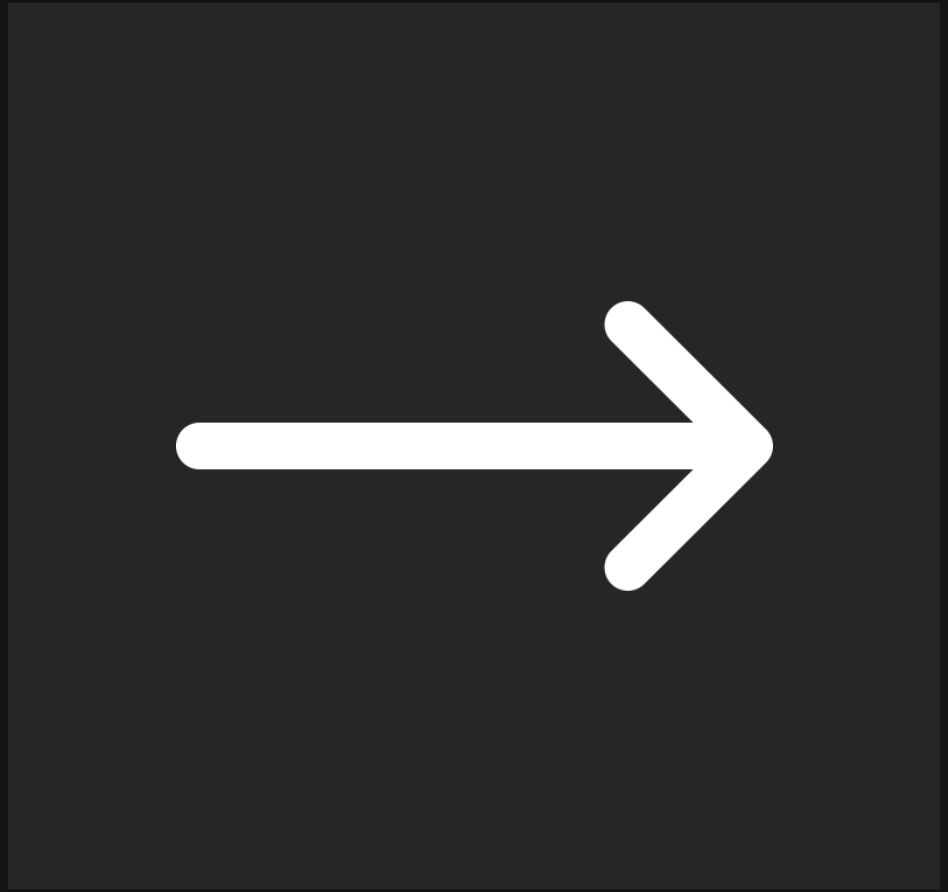
Nodes

- **Nodes** are Python functions that define the logic of your agent at different steps.
- They take the current state as input, process it, and return an updated state.
- Nodes can represent:
  - Individual functions or computations.
  - Tools the AI agent can use (e.g., calling an LLM).
  - API interactions or tool executions.

Key Components in LangGraph Agentic Systems

Edges

- **Edges** define the connections and routing logic between nodes.
- They determine the next node to execute based on the current state.
- Can represent:
  - **Conditional branches**: Dynamic paths based on state conditions.
  - **Fixed transitions**: Predefined execution paths.
- Enable flexible and dynamic execution flows.



Key Components in LangGraph Agentic Systems

State Graph & Schema:

This is typically used to define the overall state of the graph which includes the schema (variables and data types) which would be passed throughout the different nodes in the graph as the agent executes

Reducers:

Often the same state variables (e.g messages) will be updated multiple times in different nodes.

Instead of overwriting the state variable, we use reducers to store all the messages (e.g appending all new updates into the same messages variable)

Key Components in LangGraph Agentic Systems

ToolNodes:

ToolNode is a LangChain Runnable that takes graph state (with a list of messages) as input and outputs state update with the result of tool calls.

Normal Edges:

Goes directly from one node to the next, where each node is a function

Conditional Edges:

Call a function to determine which node(s) to go to next.

Key Components in LangGraph Agentic Systems

Memory & Checkpointing

- Checkpointing is the process of saving snapshots of the graph's state at specific points during its execution into memory.
- This feature is crucial for enabling persistence, fault tolerance, and enable conversational agents.

```python
from langgraph.graph import StateGraph, START, END
from langgraph.checkpoint.memory import MemorySaver

# Define your state structure
class State(TypedDict):
    data: str

# Initialize the StateGraph
graph_builder = StateGraph(State)

# Add nodes and edges to your graph
# ...

# Initialize the in-memory checkpointer
checkpointer = MemorySaver()

# Compile the graph with the in-memory checkpointer
graph = graph_builder.compile(checkpointer=checkpointer)

# Agent becomes conversational per user-session (thread_id)
initial_state = {"data": "initial value"}
result = graph.invoke(initial_state,
                      configurable={"thread_id": "userid1"})
```

Key Components in LangGraph Agentic Systems

Key Components of Memory & Checkpointing

- Checkpoints
  - Snapshots of the graph's state saved at each super-step, represented by StateSnapshot objects containing the configuration, metadata, state values, and information about the next tasks to execute.
- Threads
  - Unique identifiers assigned to each series of checkpoints, allowing the management of multiple conversation sessions or workflows. This enables you to build multi-user conversational agents
- Memory
  - This can be transient memory (temporary in-memory) or persistent memory (permanent on the disk)

```python
from langgraph.graph import StateGraph, START, END
from langgraph.checkpoint.memory import MemorySaver

# Define your state structure
class State(TypedDict):
    data: str

# Initialize the StateGraph
graph_builder = StateGraph(State)

# Add nodes and edges to your graph
# ...

# Initialize the in-memory checkpointer
checkpointer = MemorySaver()

# Compile the graph with the in-memory checkpointer
graph = graph_builder.compile(checkpointer=checkpointer)

# Agent becomes conversational per user-session (thread_id)
initial_state = {"data": "initial value"}
result = graph.invoke(initial_state,
                      configurable={"thread_id": "userid1"})
```

Key Built-In Functions in LangGraph Agentic Systems

Overview of Built-In Functions

- In LangGraph, nodes and edges are usually predefined and operate in a shared state.
- Sometimes, you don't know the edges in advance or need multiple versions of the state for different tasks.
- This is common in **map-reduce patterns**, where:
  - One node generates multiple items.
  - Each item needs to be processed individually by another node.

Key Built-In Functions in LangGraph Agentic Systems

The Send Function

- The **Send** function handles this flexibility.
- It allows you to dynamically create edges and send specific versions of the state to downstream nodes.
- Each **Send** object specifies:
  - **The target node**: Where the state should go.
  - **The specific state**: The data to process for that node.

Key Built-In Functions in LangGraph Agentic Systems

Example:

- A simple example would be based on a topic like ‘Artificial Intelligence’
- One node generates 3–5 questions on the topic using an LLM.
- The **Send** function is used to:
  - Parallelly send each question to another node.
  - The second node answers the questions using search or an LLM.
- The same answer generation node is called multiple times in parallel, once for each question.

Key Built-In Functions in LangGraph Agentic Systems

- Here we use LangGraph to process a topic using the following steps:
  - Generate sub-questions from the main topic.
  - Generate answer for each question (in-parallel)
  - Combine them later (not shown here)

```python
questions_prompt = """Generate a list of concise sub-questions related
                      to this overall topic: {topic} which would help
                      build a good report.
                      Number of questions should be 3 for simple topics
                      and 5 for more complex topics
                   """
answer_prompt = """Generate the answer about {question}."""

# assume state variable is our graph state
# this variable stores all the questions and answers
def generate_questions(state: OverallState):
    prompt = questions_prompt.format(topic=state["topic"])
    response = llm.with_structured_output(Questions).invoke(prompt)
    return {"questions": response.questions}

# Node to generate answer to one question
def generate_answer(state: Answer):
    prompt = answer_prompt.format(question=state["question"])
    response = llm.with_structured_output(Answer).invoke(prompt)
    return {"answers": [{"question": state["question"],
                         "answer": response.answer}]}

# Node to call generate_answer multiple times (once for each question)
def continue_to_answers(state: OverallState):
    return [Send("generate_answer", {"question": q})
                for q in state["questions"]]
```

Key Built-In Functions in LangGraph Agentic Systems

- The generate_questions function creates a list of sub-questions based on the given topic.
- The generate_answer function generates an answer for a single question using an LLM.
- The continue_to_answers function dynamically connects the "generate_questions" node to the "generate_answer" node, once for each question (in parallel) using Send
- Instead of predefining edges, Send creates connections dynamically at runtime which is useful especially when the number of inputs to a node could vary (e.g multiple questions to answer which are not fixed in count)

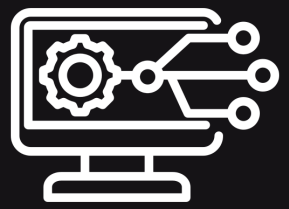
```python
questions_prompt = """Generate a list of concise sub-questions related
                      to this overall topic: {topic} which would help
                      build a good report.
                      Number of questions should be 3 for simple topics
                      and 5 for more complex topics
                   """
answer_prompt = """Generate the answer about {question}."""

# assume state variable is our graph state
# this variable stores all the questions and answers
def generate_questions(state: OverallState):
    prompt = questions_prompt.format(topic=state["topic"])
    response = llm.with_structured_output(Questions).invoke(prompt)
    return {"questions": response.questions}

# Node to generate answer to one question
def generate_answer(state: Answer):
    prompt = answer_prompt.format(question=state["question"])
    response = llm.with_structured_output(Answer).invoke(prompt)
    return {"answers": [{"question": state["question"],
                         "answer": response.answer}]}

# Node to call generate_answer multiple times (once for each question)
def continue_to_answers(state: OverallState):
    return [Send("generate_answer", {"question": q})
                for q in state["questions"]]
```

Key Built-In Functions in LangGraph Agentic Systems

- In LangGraph, the **Command** object enhances workflow flexibility by allowing nodes to dynamically determine the next node (or sub-graph) to execute, eliminating the need for predefined edges.
- This is particularly useful in multi-agent systems where control flow decisions are made at runtime (between multiple agents - which are represented as sub-graphs).

Key Built-In Functions in LangGraph Agentic Systems

Key Features of Command



Dynamic Control Flow

Nodes can specify subsequent nodes to execute based on runtime conditions, enabling adaptable workflows.

State Updates

Along with directing control flow, nodes can update the shared state, ensuring subsequent nodes have the necessary context.

Precision

By returning Command objects, nodes define exactly which node to go to next reducing hallucinations and drop in precision of the agentic system

Key Built-In Functions in LangGraph Agentic Systems

The Command object in LangGraph allows nodes to:

- Update State
  - Modify the shared state as needed
- Control Flow
  - Dynamically decide the next node to execute

```python
from langgraph.types import Command
from typing_extensions import Literal

# EXAMPLE 1: Node that performs state update and chooses the next
node - fixed flow
def my_node(state: State) -> Command[Literal["my_other_node"]]:
    return Command(
        # State update
        update={"foo": "bar"},
        # Control flow
        goto="my_other_node"
    )


# EXAMPLE 2: Node that performs state update and chooses the next
node based on a condition - dynamic conditional flow
def my_node(state: State) -> Command[Literal["my_other_node"]]:
    if state["foo"] == "bar":
        return Command(update={"foo": "baz"},
                       goto="my_other_node")
```

Key Built-In Functions in LangGraph Agentic Systems

Example 1: Fixed Flow

- update: Modifies the state by setting foo to "bar"
- goto: Specifies that the next node to execute is "my_other_node" after completing the execution of “my_node”

```python
from langgraph.types import Command
from typing_extensions import Literal

# EXAMPLE 1: Node that performs state update and chooses the next
node - fixed flow
def my_node(state: State) -> Command[Literal["my_other_node"]]:
    return Command(
        # State update
        update={"foo": "bar"},
        # Control flow
        goto="my_other_node"
    )


# EXAMPLE 2: Node that performs state update and chooses the next
node based on a condition - dynamic conditional flow
def my_node(state: State) -> Command[Literal["my_other_node"]]:
    if state["foo"] == "bar":
        return Command(update={"foo": "baz"},
                       goto="my_other_node")
```

Key Built-In Functions in LangGraph Agentic Systems

Example 2: Dynamic Conditional Flow

- Checks the state (state["foo"]) and makes decisions dynamically
- Updates the state and routes to "my_other_node" if the condition is met.

```python
from langgraph.types import Command
from typing_extensions import Literal

# EXAMPLE 1: Node that performs state update and chooses the next
node - fixed flow
def my_node(state: State) -> Command[Literal["my_other_node"]]:
    return Command(
        # State update
        update={"foo": "bar"},
        # Control flow
        goto="my_other_node"
    )


# EXAMPLE 2: Node that performs state update and chooses the next
node based on a condition - dynamic conditional flow
def my_node(state: State) -> Command[Literal["my_other_node"]]:
    if state["foo"] == "bar":
        return Command(update={"foo": "baz"},
                       goto="my_other_node")
```

Key Built-In Functions in LangGraph Agentic Systems

- In LangGraph, the interrupt function facilitates human-in-the-loop workflows by pausing graph execution at a specific point, allowing human intervention, and resuming once input is provided.
- This is particularly useful for tasks requiring approvals, edits, or additional information from a human operator.

```python
from langgraph.types import interrupt

def human_approval_node(state: State):
    ...
    answer = interrupt(
        # This value will be sent to the client.
        # It can be any JSON serializable value.
        {"question": "is it ok to continue?"},
    )
    ...
```

Key Built-In Functions in LangGraph Agentic Systems

Key Features of interrupt:

Pausing Execution

- Halts the graph at a designated node to await human input.

State Persistence

- Utilizes a checkpointer to save the current state, ensuring seamless resumption.

Resuming Execution

- Continues the graph's workflow upon receiving the necessary input.

Limitation is that the system will rely on manual inputs to proceed.

```python
from langgraph.types import interrupt

def human_approval_node(state: State):
    ...
    answer = interrupt(
        # This value will be sent to the client.
        # It can be any JSON serializable value.
        {"question": "is it ok to continue?"},
    )
    ...
```

Key Built-In Functions in LangGraph Agentic Systems

- In LangGraph, the create_react_agent function simplifies the creation of ReAct (Reasoning and Acting) agents typically based on the Tool-use & ReAct design patterns, enabling language models to interact dynamically with external tools.

```python
from langchain_openai import ChatOpenAI
from langgraph.prebuilt import create_react_agent
from langchain_core.tools import tool

# Initialize the language model
model = ChatOpenAI(model="gpt-4o", temperature=0)

# Define a tool
@tool
def get_weather(city: str):
    """Retrieve weather information for a
specified city."""
    if city.lower() == "nyc":
        return "It might be cloudy in NYC."
    elif city.lower() == "sf":
        return "It's always sunny in SF."
    else:
        return "Weather information for this city
is unavailable."

tools = [get_weather]

# Create the ReAct agent
graph = create_react_agent(model, tools=tools)
```

Key Built-In Functions in LangGraph Agentic Systems

Key Features:

- Tool Integration
  - Allows agents to utilize predefined or custom-defined tools for specific tasks.
- Dynamic Decision-Making
  - Enables agents to decide at runtime whether to use tools or respond directly.
- Customizable State Management
  - Supports state schemas to maintain context across interactions.
- *For simpler agents, you can use this function instead of creating agents from scratch using nodes, edges and states.*

```python
from langchain_openai import ChatOpenAI
from langgraph.prebuilt import create_react_agent
from langchain_core.tools import tool

# Initialize the language model
model = ChatOpenAI(model="gpt-4o", temperature=0)

# Define a tool
@tool
def get_weather(city: str):
    """Retrieve weather information for a
specified city."""
    if city.lower() == "nyc":
        return "It might be cloudy in NYC."
    elif city.lower() == "sf":
        return "It's always sunny in SF."
    else:
        return "Weather information for this city
is unavailable."

tools = [get_weather]

# Create the ReAct agent
graph = create_react_agent(model, tools=tools)
```

Key Built-In Functions in LangGraph Agentic Systems

Streaming vs. Invoking in AI Agents in LangGraph.

- LangGraph allows agents to process inputs and provide outputs in two distinct ways: streaming and invoking
  - Streaming outputs intermediate steps in real-time, making it ideal for debugging, monitoring, or observing agent reasoning or even showing real-time progress.
  - Invoking directly retrieves the final output and displays it without showing intermediate steps. Useful when you want to just show the final response.

Feature	Streaming	Invoking
Purpose	Observe reasoning and intermediate steps	Directly retrieve final response
Use Case	Debugging, monitoring, tool behavior	Efficient production workflows
Output	Includes intermediate reasoning, tool calls	Final answer only

Key Built-In Functions in LangGraph Agentic Systems

```python
# Define user input
inputs = {"messages": [("user", "What is the weather in
SF?")]}

# Stream the agent's responses
for response in graph.stream(inputs, stream_mode="values"):
    message = response["messages"][-1]
    if isinstance(message, tuple):  # User messages
        print(message)
    else:  # AI responses
        message.pretty_print()

## OUTPUT
('user', 'What is the weather in SF?')
====================== AI Message ======================
Tool Calls:
  get_weather (call_zVvnU9DKr6jsNnluFIl59mHb)
 Call ID: call_zVvnU9DKr6jsNnluFIl59mHb
  Args:
    city: SF
====================== Tool Message ====================
Name: get_weather
It's always sunny in SF
====================== AI Message ======================
```

Agent Streaming

```python
# Define user input
inputs = {"messages": [("user", "What is the weather
in SF?")]}

# Directly invoke the agent for a final response
result = graph.invoke(inputs)

# Print the final response
final_message = result["messages"][-1]
if isinstance(final_message, tuple):
    print(final_message)  # User message
else:
    final_message.pretty_print()  # AI response

## OUTPUT
==================== AI Message ====================
It's always sunny in SF
```

Agent Invoking

Analytics Vidhya

Thanks