Cheat Sheet: Introduction to LangGraph

Estimated time needed: 10 minutes

Getting started with LangGraph

Overview	LangGraph is an open-source (MIT-licensed) framework for building stateful, graph-based AI agents.
Extension of LangChain	It builds on LangChain by enabling workflows as graphs of nodes, with explicit control flow and state management.
State management	A central state object (typically a TypedDict or Pydantic model) is passed between nodes, each of which updates and processes that state.
Workflow capabilities	Supports branching, looping, memory retention, and conditional logic—beyond what a simple, linear LangChain chain can offer.
Advanced behaviors	Enables complex agent behaviors such as iterative reasoning, conditional paths, and human-in-the-loop interactions.
Execution features	Workflows can run over time (durable execution), support human inspection of state, and leverage both short- and long-term memory for decisions.
Ecosystem integration	Interoperable with the full LangChain ecosystem, including tools, chains, memory components, and LangSmith for observability and debugging.
Installation	`pip install langgraph`

Why graph-based agents?

Traditional LangChain chains are Directed Acyclic Graphs (DAGs). They define a fixed, linear sequence of LLM calls and tool invocations. These chains are suitable for simple, one-pass tasks but lack support for branching or looping.

LangGraph agents operate as state machines. They allow the system to revisit steps, make decisions conditionally, and model complex flows like loops, retries, and branching paths.

In a traditional chain, retrieval runs once—if the result is poor, the system is stuck. With LangGraph, the LLM can loop: it can revise the query, retrieve it again, and continue, enabling adaptive behavior.

When to use LangGraph

LangGraph is ideal for complex agent workflows that need explicit state and flexible control flow. Use it when your task involves:

Concept	Explanation
Loops or iteration	Tasks where the agent might try an action, check results, and repeat until a goal is achieved. (for example, iterative refinement of a query or planning steps.)
Conditional branching	Workflows with if/else logic. For instance, a support bot that asks follow-up questions based on user replies.
Long-running processes	Scenarios where the agent must persist state and resume after delays or failures (LangGraph supports durable execution and checkpointing).
Complex state management	When many variables or data points must be carried through the workflow; LangGraph's shared state object is more explicit than passing context through nested chains.
Multi-agent or multi-step coordination	You can design graphs where different nodes represent different agents or tools working together, with the central state tracking their interactions.

Core concepts of LangGraph

State

State is the shared, central piece of data that flows through your LangGraph workflow. Think of it as a dictionary (or, more formally, a `TypedDict` or `Pydantic` model) that carries all relevant information from one node to the next. Each node in the graph reads from and updates this state object. Below is an example of a state:

```
from typing import TypedDict
class WorkflowState(TypedDict):
    user_query: str
    summary: str
    step_count: int
```

Initialize a state field with an initial value (e.g., `{"user_query": "Hello", "summary": "", "step_count": 0}`) when invoking the graph.

StateGraph

StateGraph is the controller or blueprint of the workflow. It is a class provided by LangGraph that lets you define:
- What nodes exist
- How they connect (edges)
- Where the workflow starts and ends
- When to loop or branch conditionally

In other words, `State` is the data that flows through the system (changes during execution) but a `StateGraph` is the structure that defines how that data moves and gets transformed (fixed once compiled). You create a `StateGraph` by passing the state schema type:

```
from langgraph.graph import StateGraph
graph = StateGraph(WorkflowState)
```

Nodes

Each node is a Python function (or LangChain Runnable) that takes the state dict and returns an updated state. Nodes perform actions such as calling an LLM, running a tool, computing something, etc. For example:

```
def summarize(state: WorkflowState) -> WorkflowState:
    text = state["user_query"]
    state["summary"] = llm_summarize(text)  # some LLM call
    return state
```

You can add nodes to the graph using `graph.add_node()`. Each node should update the state and return it. LangGraph can also use LangChain chains or agents as nodes (they must conform to the same state signature).

Edges

Edges define how the workflow moves from one node to the next.

- **Linear (normal) edges**: Use `graph.add_edge(from_node, to_node)` to always flow from one node to the next. You must specify an entry point and exit using the special START and END tokens from langgraph.graph. For example:

```
from langgraph.graph import START, END
graph.add_edge(START, "summarize")
graph.add_edge("summarize", "finalize")
graph.add_edge("finalize", END)
```

Here, we add the edges from `START` to `summarize`, indicating that the graph workflow will begin from `summarize`. After that, we have two other edges, one from `summarize` to `finalize` and another from `finalize` to `END` indicating the end of the workflow

- **Conditional edges**: Use `graph.add_conditional_edges(from_node, decision_func, mapping)` to branch. The decision_func(state) should return a string key; then, the workflow moves to whichever node name that key maps to. For example:

```
def decide(state: WorkflowState) -> str:
    return "repeat" if state["step_count"] < 2 else "done"
graph.add_conditional_edges(
    "summarize",
    decide,
    {"repeat": "summarize", "done": END}
)
```

In this case, after "summarize" is executed, `decide()` function checks `step_count`. If it returns "repeat", the graph loops back to the "summarize" node again; if "done", it goes to the special `END` and stops. Conditional edges let LLM-driven or logic-driven functions choose the next step dynamically.

Compile and run

Once all nodes and edges are added, call `runnable = graph.compile()`. This produces a Runnable object (just like a LangChain Runnable) that you can run with `.invoke(initial_state)` or `.stream(initial_state)`. For example:

```
runnable = graph.compile()
final_state = runnable.invoke({"user_query": "Hello", "summary": "", "step_count": 0})
```

This executes the graph: it starts at `START`, follows edges (running each node's function on the state), and stops at `END`. The final state dict contains all updates. The compiled graph supports all usual LangChain methods (.stream(), async variants, batching, etc.)

Visualizing your graph with a Mermaid diagram

LangGraph supports generating Mermaid diagrams, a lightweight syntax for rendering flowcharts and state diagrams. This helps you visually understand how your agent moves from one node to another, especially when there are loops or conditional branches. Once your graph is built, you can render a Mermaid diagram using:

```
print(app.get_graph().draw_mermaid())
```

A LangGraph Example

In this example, let's build an increment counter using LangGraph.

Step	Description

Define the state schema	We start with defining the State Schema with a `TypedDict` (or `Pydantic` model) listing all fields your workflow needs. Example: from typing import TypedDict
	`class GraphState(TypedDict):`
	`  count: int`
	`  message: str`
	This says our state has an integer `count` and a string `message`.

Initialize the StateGraph	`from langgraph.graph import StateGraph` `graph = StateGraph(GraphState)`

Add nodes	For each step, write a function that takes and returns the state. Then register it with `add_node()`. Example:
	`def increment(state: GraphState) -> GraphState:`
	`  state["count"] += 1`
	`  state["message"] = f"Count is now {state['count']}"`
	`  return state`
	`graph.add_node("increment", increment)`
	You can add as many nodes as needed, possibly using the same function multiple times under different names.

Connect edges	Define the flow of execution. At minimum, set a start edge from `START`, and usually end at `END`. For linear flow:
	`from langgraph.graph import START, END`
	`graph.add_edge(START, "increment")`
	`graph.add_edge("increment", END)`

Conditional branching (optional)	To loop or branch, use `add_conditional_edge()`. For example, to repeat the "increment" node until the count reaches 3:
	`def decide_next(state: GraphState) -> str:`
	`  return "again" if state["count"] < 3 else "finish"`
	`graph.add_conditional_edges("increment", decide_next, {"again": "increment", "finish": END})`
	Now, after each "increment", the graph checks the returned key: if "again", it loops back to "increment" (making a cycle); if "finish", it goes to `END`. This simple loop will run the increment node three times.

Compile and invoke	Finally, compile the graph and run it:
	`app = graph.compile()`
	`result = app.invoke({"count": 0, "message": ""})`
	Here, the initial state has count=0. After invoking, the result contains the updated state (e.g., count = 3 if we looped three times).

Author

Karan Goswami

Other Contributor(s)

Faranak Heidari