Graph API concepts

Graphs

At its core, LangGraph models agent workflows as graphs. You define the behavior of your agents using three key components:

1. `State`: A shared data structure that represents the current snapshot of your application. It can be any data type, but is typically defined using a shared state schema.
2. `Nodes`: Functions that encode the logic of your agents. They receive the current state as input, perform some computation or side-effect, and return an updated state.
3. `Edges`: Functions that determine which `Node` to execute next based on the current state. They can be conditional branches or fixed transitions.

By composing `Nodes` and `Edges`, you can create complex, looping workflows that evolve the state over time. The real power, though, comes from how LangGraph manages that state. To emphasize: `Nodes` and `Edges` are nothing more than functions - they can contain an LLM or just good ol' code.

In short: *nodes do the work, edges tell what to do next*.

LangGraph's underlying graph algorithm uses message passing to define a general program. When a Node completes its operation, it sends messages along one or more edges to other node(s). These recipient nodes then execute their functions, pass the resulting messages to the next set of nodes, and the process continues. Inspired by Google's Pregel system, the program proceeds in discrete "super-steps."

A super-step can be considered a single iteration over the graph nodes. Nodes that run in parallel are part of the same super-step, while nodes that run sequentially belong to separate super-steps. At the start of graph execution, all nodes begin in an `inactive` state. A node becomes `active` when it receives a new message (state) on any of its incoming edges (or "channels"). The active node then runs its function and responds with updates. At the end of each super-step, nodes with no incoming messages vote to `halt` by marking themselves as `inactive`. The graph execution terminates when all nodes are `inactive` and no messages are in transit.

StateGraph

The `StateGraph` class is the main graph class to use. This is parameterized by a user defined `State` object.

Compiling your graph

To build your graph, you first define the state, you then add nodes and edges, and then you compile it. What exactly is compiling your graph and why is it needed?

Compiling is a pretty simple step. It provides a few basic checks on the structure of your graph (no orphaned nodes, etc). It is also where you can specify runtime args like checkpointers and breakpoints. You compile your graph by just calling the `.compile` method:

```
graph = graph_builder.compile(...)
```

You **MUST** compile your graph before you can use it.

State

The first thing you do when you define a graph is define the `State` of the graph. The `State` consists of the schema of the graph as well as `reducer` functions which specify how to apply updates to the state. The schema of the `State` will be the input schema to all `Nodes` and `Edges`

in the graph, and can be either a `TypedDict` or a `Pydantic` model. All `Nodes` will emit updates to the `State` which are then applied using the specified `reducer` function.

Schema

The main documented way to specify the schema of a graph is by using a `TypedDict`. If you want to provide default values in your state, use a `dataclass`. We also support using a Pydantic BaseModel as your graph state if you want recursive data validation (though note that pydantic is less performant than a `TypedDict` or `dataclass`).

By default, the graph will have the same input and output schemas. If you want to change this, you can also specify explicit input and output schemas directly. This is useful when you have a lot of keys, and some are explicitly for input and others for output. See the guide here for how to use.

Multiple schemas

Typically, all graph nodes communicate with a single schema. This means that they will read and write to the same state channels. But, there are cases where we want more control over this:

- Internal nodes can pass information that is not required in the graph's input / output.
- We may also want to use different input / output schemas for the graph. The output might, for example, only contain a single relevant output key.

It is possible to have nodes write to private state channels inside the graph for internal node communication. We can simply define a private schema, `PrivateState`.

It is also possible to define explicit input and output schemas for a graph. In these cases, we define an "internal" schema that contains *all* keys relevant to graph operations. But, we also define `input` and `output` schemas that are sub-sets of the "internal" schema to constrain the input and output of the graph. See this guide for more detail.

Let's look at an example:

```python
class InputState(TypedDict):
    user_input: str

class OutputState(TypedDict):
    graph_output: str

class OverallState(TypedDict):
    foo: str
    user_input: str
    graph_output: str

class PrivateState(TypedDict):
    bar: str

def node_1(state: InputState) -> OverallState:
    # Write to OverallState
    return {"foo": state["user_input"] + " name"}

def node_2(state: OverallState) -> PrivateState:
    # Read from OverallState, write to PrivateState
    return {"bar": state["foo"] + " is"}

def node_3(state: PrivateState) -> OutputState:
    # Read from PrivateState, write to OutputState
    return {"graph_output": state["bar"] + " Lance"}

builder = StateGraph(OverallState,input_schema=InputState,output_schema=OutputState)
builder.add_node("node_1", node_1)
builder.add_node("node_2", node_2)
builder.add_node("node_3", node_3)
builder.add_edge(START, "node_1")
builder.add_edge("node_1", "node_2")
builder.add_edge("node_2", "node_3")
builder.add_edge("node_3", END)

graph = builder.compile()
```

```
graph.invoke({"user_input":"My"})
# {'graph_output': 'My name is Lance'}
```

There are two subtle and important points to note here:

1. We pass `state: InputState` as the input schema to `node_1`. But, we write out to `foo`, a channel in `OverallState`. How can we write out to a state channel that is not included in the input schema? This is because a node *can write to any state channel in the graph state.* The graph state is the union of the state channels defined at initialization, which includes `OverallState` and the filters `InputState` and `OutputState`.
2. We initialize the graph with `StateGraph(OverallState,input_schema=InputState,output_schema=OutputState)`. So, how can we write to `PrivateState` in `node_2`? How does the graph gain access to this schema if it was not passed in the `StateGraph` initialization? We can do this because *nodes can also declare additional state channels* as long as the state schema definition exists. In this case, the `PrivateState` schema is defined, so we can add `bar` as a new state channel in the graph and write to it.

Reducers

Reducers are key to understanding how updates from nodes are applied to the `State`. Each key in the `State` has its own independent reducer function. If no reducer function is explicitly specified then it is assumed that all updates to that key should override it. There are a few different types of reducers, starting with the default type of reducer:

Default Reducer

These two examples show how to use the default reducer:

Example A:

```
from typing_extensions import TypedDict

class State(TypedDict):
    foo: int
    bar: list[str]
```

In this example, no reducer functions are specified for any key. Let's assume the input to the graph is:

`{"foo": 1, "bar": ["hi"]}`. Let's then assume the first `Node` returns `{"foo": 2}`. This is treated as an update to the state. Notice that the `Node` does not need to return the whole `State` schema - just an update. After applying this update, the `State` would then be `{"foo": 2, "bar": ["hi"]}`. If the second node returns `{"bar": ["bye"]}` then the `State` would then be `{"foo": 2, "bar": ["bye"]}`

Example B:

```
from typing import Annotated
from typing_extensions import TypedDict
from operator import add

class State(TypedDict):
    foo: int
    bar: Annotated[list[str], add]
```

In this example, we've used the `Annotated` type to specify a reducer function (`operator.add`) for the second key (`bar`). Note that the first key remains unchanged. Let's assume the input to the graph is `{"foo": 1, "bar": ["hi"]}`. Let's then assume the first `Node` returns `{"foo": 2}`. This is treated as an update to the state. Notice that the `Node` does not need to return the whole `State` schema - just an update. After applying this update, the `State` would then be `{"foo": 2, "bar": ["hi"]}`. If the second node returns `{"bar": ["bye"]}` then the `State` would then be `{"foo": 2, "bar": ["hi", "bye"]}`. Notice here that the `bar` key is updated by adding the two lists together.

Working with Messages in Graph State

Why use messages?

Most modern LLM providers have a chat model interface that accepts a list of messages as input. LangChain's `ChatModel` in particular accepts a list of `Message` objects as inputs. These messages come in a variety of forms such as `HumanMessage` (user input) or `AIMessage` (LLM response). To read more about what message objects are, please refer to this conceptual guide.

Using Messages in your Graph

In many cases, it is helpful to store prior conversation history as a list of messages in your graph state. To do so, we can add a key (channel) to the graph state that stores a list of `Message` objects and annotate it with a reducer function (see `messages` key in the example below). The reducer function is vital to telling the graph how to update the list of `Message` objects in the state with each state update (for example, when a node sends an update). If you don't specify a reducer, every state update will overwrite the list of messages with the most recently provided value. If you wanted to simply append messages to the existing list, you could use `operator.add` as a reducer.

However, you might also want to manually update messages in your graph state (e.g. human-in-the-loop). If you were to use `operator.add`, the manual state updates you send to the graph would be appended to the existing list of messages, instead of updating existing messages. To avoid that, you need a reducer that can keep track of message IDs and overwrite existing messages, if updated. To achieve this, you can use the prebuilt `add_messages` function. For brand new messages, it will simply append to existing list, but it will also handle the updates for existing messages correctly.

Serialization

In addition to keeping track of message IDs, the `add_messages` function will also try to deserialize messages into LangChain `Message` objects whenever a state update is received on the `messages` channel. See more information on LangChain serialization/deserialization here. This allows sending graph inputs / state updates in the following format:

```
# this is supported
{"messages": [HumanMessage(content="message")]}

# and this is also supported
{"messages": [{"type": "human", "content": "message"}]}
```

Since the state updates are always deserialized into LangChain `Messages` when using `add_messages`, you should use dot notation to access message attributes, like `state["messages"][-1].content`. Below is an example of a graph that uses `add_messages` as its reducer function.

API Reference: AnyMessage | add_messages

```
from langchain_core.messages import AnyMessage
from langgraph.graph.message import add_messages
from typing import Annotated
from typing_extensions import TypedDict

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

MessagesState

Since having a list of messages in your state is so common, there exists a prebuilt state called `MessagesState` which makes it easy to use messages. `MessagesState` is defined with a single `messages` key which is a list of `AnyMessage` objects and uses the `add_messages` reducer. Typically, there is more state to track than just messages, so we see people subclass this state and add more fields, like:

```
from langgraph.graph import MessagesState

class State(MessagesState):
    documents: list[str]
```

Nodes

In LangGraph, nodes are Python functions (either synchronous or asynchronous) that accept the following arguments:

1. `state`: The state of the graph
2. `config`: A `RunnableConfig` object that contains configuration information like `thread_id` and tracing information like `tags`

3. `runtime`: A `Runtime` object that contains runtime `context` and other information like `store` and `stream_writer`

Similar to `NetworkX`, you add these nodes to a graph using the add_node method:

API Reference: RunnableConfig | StateGraph

```python
from dataclasses import dataclass
from typing_extensions import TypedDict

from langchain_core.runnables import RunnableConfig
from langgraph.graph import StateGraph
from langgraph.runtime import Runtime

class State(TypedDict):
    input: str
    results: str

@dataclass
class Context:
    user_id: str

builder = StateGraph(State)

def plain_node(state: State):
    return state

def node_with_runtime(state: State, runtime: Runtime[Context]):
    print("In node: ", runtime.context.user_id)
    return {"results": f"Hello, {state['input']}!"}

def node_with_config(state: State, config: RunnableConfig):
    print("In node with thread_id: ", config["configurable"]["thread_id"])
    return {"results": f"Hello, {state['input']}!"}


builder.add_node("plain_node", plain_node)
```

```
builder.add_node("node_with_runtime", node_with_runtime)
builder.add_node("node_with_config", node_with_config)
...
```

Behind the scenes, functions are converted to RunnableLambdas, which add batch and async support to your function, along with native tracing and debugging.

If you add a node to a graph without specifying a name, it will be given a default name equivalent to the function name.

```
builder.add_node(my_node)
# You can then create edges to/from this node by referencing it as `"my_node"`
```

`START` Node

The `START` Node is a special node that represents the node that sends user input to the graph. The main purpose for referencing this node is to determine which nodes should be called first.

API Reference: START

```
from langgraph.graph import START

graph.add_edge(START, "node_a")
```

`END` Node

The `END` Node is a special node that represents a terminal node. This node is referenced when you want to denote which edges have no actions after they are done.

API Reference: END

```
from langgraph.graph import END

graph.add_edge("node_a", END)
```

Node Caching

LangGraph supports caching of tasks/nodes based on the input to the node. To use caching:

- Specify a cache when compiling a graph (or specifying an entrypoint)
- Specify a cache policy for nodes. Each cache policy supports:
- `key_func` used to generate a cache key based on the input to a node, which defaults to a `hash` of the input with pickle.
- `ttl`, the time to live for the cache in seconds. If not specified, the cache will never expire.

For example:

API Reference: StateGraph

```
import time
from typing_extensions import TypedDict
from langgraph.graph import StateGraph
from langgraph.cache.memory import InMemoryCache
from langgraph.types import CachePolicy


class State(TypedDict):
    x: int
    result: int


builder = StateGraph(State)
```

```python
def expensive_node(state: State) -> dict[str, int]:
    # expensive computation
    time.sleep(2)
    return {"result": state["x"] * 2}


builder.add_node("expensive_node", expensive_node, cache_policy=CachePolicy(ttl=3))
builder.set_entry_point("expensive_node")
builder.set_finish_point("expensive_node")

graph = builder.compile(cache=InMemoryCache())

print(graph.invoke({"x": 5}, stream_mode='updates'))  # (1)
[{'expensive_node': {'result': 10}}]
print(graph.invoke({"x": 5}, stream_mode='updates'))  # (2)
[{'expensive_node': {'result': 10}, '__metadata__': {'cached': True}}]
```

1. First run takes two seconds to run (due to mocked expensive computation).
2. Second run utilizes cache and returns quickly.

Edges

Edges define how the logic is routed and how the graph decides to stop. This is a big part of how your agents work and how different nodes communicate with each other. There are a few key types of edges:

- Normal Edges: Go directly from one node to the next.
- Conditional Edges: Call a function to determine which node(s) to go to next.
- Entry Point: Which node to call first when user input arrives.
- Conditional Entry Point: Call a function to determine which node(s) to call first when user input arrives.

A node can have MULTIPLE outgoing edges. If a node has multiple out-going edges, **all** of those destination nodes will be executed in parallel as a part of the next superstep.

Normal Edges

If you **always** want to go from node A to node B, you can use the add_edge method directly.

```
graph.add_edge("node_a", "node_b")
```

Conditional Edges

If you want to **optionally** route to 1 or more edges (or optionally terminate), you can use the add_conditional_edges method. This method accepts the name of a node and a "routing function" to call after that node is executed:

```
graph.add_conditional_edges("node_a", routing_function)
```

Similar to nodes, the `routing_function` accepts the current `state` of the graph and returns a value.

By default, the return value `routing_function` is used as the name of the node (or list of nodes) to send the state to next. All those nodes will be run in parallel as a part of the next superstep.

You can optionally provide a dictionary that maps the `routing_function`'s output to the name of the next node.

```
graph.add_conditional_edges("node_a", routing_function, {True: "node_b", False: "node_c"})
```

> **Tip**
>
> Use `Command` instead of conditional edges if you want to combine state updates and routing in a single function.

Entry Point

The entry point is the first node(s) that are run when the graph starts. You can use the `add_edge` method from the virtual `START` node to the first node to execute to specify where to enter the graph.

API Reference: START

```
from langgraph.graph import START

graph.add_edge(START, "node_a")
```

Conditional Entry Point

A conditional entry point lets you start at different nodes depending on custom logic. You can use `add_conditional_edges` from the virtual `START` node to accomplish this.

API Reference: START

```
from langgraph.graph import START

graph.add_conditional_edges(START, routing_function)
```

You can optionally provide a dictionary that maps the `routing_function`'s output to the name of the next node.

```
graph.add_conditional_edges(START, routing_function, {True: "node_b", False: "node_c"})
```

Send

By default, `Nodes` and `Edges` are defined ahead of time and operate on the same shared state. However, there can be cases where the exact edges are not known ahead of time and/or you may want different versions of `State` to exist at the same time. A common example of this is with map-reduce design patterns. In this design pattern, a first node may generate a list of objects, and you may want to apply some other node to all those objects. The number of objects may be unknown ahead of time (meaning the number of edges may not be known) and the input `State` to the downstream `Node` should be different (one for each generated object).

To support this design pattern, LangGraph supports returning `Send` objects from conditional edges. `Send` takes two arguments: first is the name of the node, and second is the state to pass to that node.

```
def continue_to_jokes(state: OverallState):
    return [Send("generate_joke", {"subject": s}) for s in state['subjects']]

graph.add_conditional_edges("node_a", continue_to_jokes)
```

Command

It can be useful to combine control flow (edges) and state updates (nodes). For example, you might want to BOTH perform state updates AND decide which node to go to next in the SAME node. LangGraph provides a way to do so by returning a `Command` object from node functions:

```
def my_node(state: State) -> Command[Literal["my_other_node"]]:
    return Command(
        # state update
        update={"foo": "bar"},
        # control flow
        goto="my_other_node"
```

```
    )
```

With `Command` you can also achieve dynamic control flow behavior (identical to conditional edges):

```
def my_node(state: State) -> Command[Literal["my_other_node"]]:
    if state["foo"] == "bar":
        return Command(update={"foo": "baz"}, goto="my_other_node")
```

Important

When returning `Command` in your node functions, you must add return type annotations with the list of node names the node is routing to, e.g. `Command[Literal["my_other_node"]]`. This is necessary for the graph rendering and tells LangGraph that `my_node` can navigate to `my_other_node`.

Check out this how-to guide for an end-to-end example of how to use `Command`.

When should I use Command instead of conditional edges?

- Use `Command` when you need to **both** update the graph state **and** route to a different node. For example, when implementing multi-agent handoffs where it's important to route to a different agent and pass some information to that agent.
- Use conditional edges to route between nodes conditionally without updating the state.

Navigating to a node in a parent graph

If you are using subgraphs, you might want to navigate from a node within a subgraph to a different subgraph (i.e. a different node in the parent graph). To do so, you can specify `graph=Command.PARENT` in `Command`:

```
def my_node(state: State) -> Command[Literal["other_subgraph"]]:
```

```
    return Command(
        update={"foo": "bar"},
        goto="other_subgraph",  # where `other_subgraph` is a node in the parent graph
        graph=Command.PARENT
    )
```

Note

Setting `graph` to `Command.PARENT` will navigate to the closest parent graph.

State updates with `Command.PARENT`

When you send updates from a subgraph node to a parent graph node for a key that's shared by both parent and subgraph state schemas, you **must** define a reducer for the key you're updating in the parent graph state. See this example.

This is particularly useful when implementing multi-agent handoffs.

Check out this guide for detail.

Using inside tools

A common use case is updating graph state from inside a tool. For example, in a customer support application you might want to look up customer information based on their account number or ID in the beginning of the conversation.

Refer to this guide for detail.

Human-in-the-loop

`Command` is an important part of human-in-the-loop workflows: when using `interrupt()` to collect user input, `Command` is then used to supply the input and resume execution via `Command(resume="User input")`. Check out this conceptual guide for more information.

Graph Migrations

LangGraph can easily handle migrations of graph definitions (nodes, edges, and state) even when using a checkpointer to track state.

- For threads at the end of the graph (i.e. not interrupted) you can change the entire topology of the graph (i.e. all nodes and edges, remove, add, rename, etc)
- For threads currently interrupted, we support all topology changes other than renaming / removing nodes (as that thread could now be about to enter a node that no longer exists) -- if this is a blocker please reach out and we can prioritize a solution.
- For modifying state, we have full backwards and forwards compatibility for adding and removing keys
- State keys that are renamed lose their saved state in existing threads
- State keys whose types change in incompatible ways could currently cause issues in threads with state from before the change -- if this is a blocker please reach out and we can prioritize a solution.

Runtime Context

When creating a graph, you can specify a `context_schema` for runtime context passed to nodes. This is useful for passing information to nodes that is not part of the graph state. For example, you might want to pass dependencies such as model name or a database connection.

```
@dataclass
class ContextSchema:
    llm_provider: str = "openai"

graph = StateGraph(State, context_schema=ContextSchema)
```

You can then pass this context into the graph using the `context` parameter of the `invoke` method.

```
graph.invoke(inputs, context={"llm_provider": "anthropic"})
```

You can then access and use this context inside a node or conditional edge:

```
from langgraph.runtime import Runtime

def node_a(state: State, runtime: Runtime[ContextSchema]):
    llm = get_llm(runtime.context.llm_provider)
    ...
```

See this guide for a full breakdown on configuration. :::

Recursion Limit

The recursion limit sets the maximum number of super-steps the graph can execute during a single execution. Once the limit is reached, LangGraph will raise `GraphRecursionError`. By default this value is set to 25 steps. The recursion limit can be set on any graph at runtime, and is passed to `.invoke` / `.stream` via the config dictionary. Importantly, `recursion_limit` is a standalone `config` key and should not be passed inside the `configurable` key as all other user-defined configuration. See the example below:

```
graph.invoke(inputs, config={"recursion_limit": 5}, context={"llm": "anthropic"})
```

Read this how-to to learn more about how the recursion limit works.

Visualization

It's often nice to be able to visualize graphs, especially as they get more complex. LangGraph comes with several built-in ways to visualize graphs. See this how-to guide for more info.