LangGraph Intro

Refer → https://langchain-ai.github.io/langgraph/agents/overview/

What is LangGraph?

Think of **LangGraph** as a *special tool* that helps you design and run **AI workflows** where **steps depend on each other** — kind of like drawing a *flowchart* for your AI app and then letting it run automatically.

It is an **extension of LangChain** (just like LangChain helps you connect LLMs, tools, and data sources), but LangGraph adds something extra:

- The ability to create **graphs** where each step (node) is a function, an LLM call, or a data transformation.
- The flow can **loop**, **branch**, or **wait for input** instead of always going in a straight line.

In simple words:

LangChain is like a *box of Lego pieces for LLM apps*.

LangGraph is like a *blueprint* that tells those Lego pieces **exactly how to connect, when to run, and in what order** — even if you want loops, conditions, or re-tries.

Core Concepts in LangGraph

You only need to remember **three main ideas**:

1. **Graph** – The full workflow.
   - Like the map of your AI process.
   - Created using `StateGraph` or `MessageGraph`.
2. **Nodes** – The steps inside the graph.
   - Can be LLM calls, functions, API calls, database queries, etc.

3. **Edges** – The connections between nodes.
    - Define *what happens next* after each node runs.

Key Concepts

At its core, LangGraph represents your application as a **graph**, where each node is a component (like an LLM call or a function) and each edge defines the flow of control between them. The application's **state** is the shared memory between these nodes.

State

The **state** is a Python dictionary that stores the data that's being passed between the nodes. Each node can read from and write to this state. This allows the components to have a shared context and work together. For example, the state could hold the user's initial query, the LLM's response, or the results of a tool call.

> All the important components

```
state = {
    "goal": "Hire a backend software engineer",
    "jd": "",  # Job description text
    "jd_approved": False,
    "jd_posted": False,
    "min_applicants": 5,
    "num_applications": 0,
    "shortlisted_candidates": {
        # "candidate_id": {"name": ..., "score": ..
    },
    "interview_questions": [],
    "offer_status": {
        "sent": False,
        "accepted": False,
        "renegotiated": False
    },
    "onboarding_status": {
        "completed": False,
        "start_date": None,
        "employee_id": None
    }
}
```

Nodes

Nodes are the building blocks of your graph. They are essentially functions that take the current state as input and return an updated state. A node can be:

- **A tool:** A function that performs a specific action, like searching the web or calling an API.
- **An LLM:** A function that makes a call to a large language model.

- **A regular Python function:** A function that handles some data manipulation or logic.

Edges

Edges define how the application flows from one node to another. There are two main types:

- **Conditional Edges:** These edges determine the next node based on the output of the current node. For example, if a node's output is "continue," the graph might move to a different node than if the output was "finish." These are essential for creating dynamic, non-linear workflows.
- **Normal Edges:** These edges simply connect one node to the next, forming a direct sequence.

How it Works

1. **Define the State:** You first define a `State` object, which is usually a subclass of `TypedDict` (or `Pydantic`), to specify what information your application will track.
   - This is accessible at every node
   - The dictionary is **mutable**.
2. **Create the Graph:** You instantiate a `StateGraph` and add your nodes and edges.
3. **Compile the Graph:** You `compile()` the graph to create a runnable object.
4. **Invoke:** You can then invoke the runnable graph with a starting state (e.g., the user's initial prompt) and watch it run!

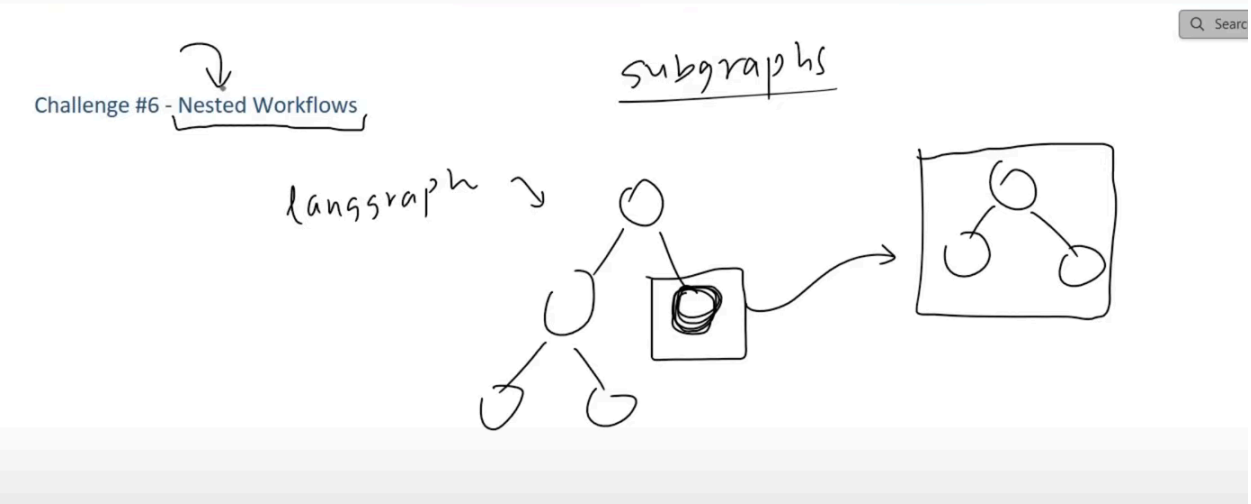
Analogy

Think of LangGraph as a **flowchart with memory.** Each box in the flowchart is a node, and the arrows are the edges. The key difference is that LangGraph remembers everything that happened in the previous boxes, allowing it to make smarter decisions and take different paths based on the accumulated information. This is what makes it so powerful for building complex agents and interactive applications.

Key Features

- **Agent support** (integrates with LangChain tools & retrievers)
- **Loops and branches** (more advanced than a simple chain)
- **Stateful execution** (remembers data between steps)
- **Streaming output** (send partial responses as they're generated)
- **Error handling** (define what to do if something fails)

Nested Workflow



- Any node can be treated as a subgraph

Reliability and Control

- **Human-in-the-loop**: Built-in capabilities for human oversight, approval workflows, and course correction
- **Moderation and quality controls**: Easy-to-add safeguards to prevent agents from going off track
- **Time travel**: Ability to rewind and explore alternative conversation paths

Statefulness and Memory

- **Persistent context**: Maintains conversation histories and context across long-running sessions
- **Built-in memory**: Stores information for personalized interactions over time

Streaming and Real-time Interaction

- **First-class streaming support**: Token-by-token streaming and intermediate step visibility
- **Real-time feedback**: Users can see agent reasoning and actions as they happen

Flexibility and Customization

- **Low-level primitives**: Build fully customizable agents without rigid abstractions
- **Multiple control flows**: Support for single agent, multi-agent, hierarchical, and sequential workflows
- **Scalable architecture**: Design complex multi-agent systems with specialized roles

Simple Example

```
pip install -U langgraph
```

```
from langgraph.graph import StateGraph, END
from typing import TypedDict

# 1. Define the data/state that passes through the graph
class State(TypedDict):
    message: str

# 2. Create the graph
workflow = StateGraph(State)
```

```
# 3. Define a node (function) for the graph
def greet(state: State):
    return {"message": f"Hello, {state['message']}!"}

# 4. Add the node to the graph
workflow.add_node("greeter", greet)

# 5. Set the entry point
workflow.set_entry_point("greeter")

# 6. Define where it ends
workflow.add_edge("greeter", END)

# 7. Compile the graph
app = workflow.compile()

# 8. Run it
result = app.invoke({"message": "Sir"})
print(result)  # {'message': 'Hello, Sir!'}
```