**Langchain vs LangGraph**

**What is LangChain ?**

* LangChain is an open-source library designed to simplify the process of building LLM based applications.
* It provides modular building blocks that let you create sophisticated LLM-based workflows with ease.
* LangChain consists of multiple components:
  + **Model** components give a unified interface to interact with various LLM providers.
  + **Prompts** component helps you engineer prompts.
  + **Retrievers** component helps you fetch relevant documents from a vector store.
* But the biggest offering of LangChain is **Chains.**

**What can you do with LangChain ?**

1. Simple conversational workflows like Chatbots, Text Summarizers
2. Multistep workflows
3. RAG applications
4. Basic level agents

**Challenges in building complex workflows in LangChain:**

1. Control Flow Complexity
2. Handling State
3. Event Driven Execution
4. Fault Tolerance
5. Human in the loop
6. Nested Workflows
7. Observeability

**What is LangGraph ?**

* LangGraph is an orchestration framework that enables you to build **stateful**, **multi-step**, and **event-driven workflows** using large language models (LLMs). It's ideal for designing both single-agent and multi-agent agentic Al applications.
* Think of LangGraph as a **flowchart engine for LLMs**-you define the steps (nodes), how they're connected (edges), and the logic that governs the transitions. LangGraph takes care of **state management**, **conditional branching**, **looping**, **pausing/resuming**, and **fault recovery**-features essential for building robust, production-grade Al systems.

**When to Use What?**

* Use **LangChain** when you're building **simple, linear workflows** - like a prompt chain, summarizer, or a basic retrieval system.
* Use **LangGraph** when your use case involves **complex, non-linear workflows** that need:
  + Conditional paths
  + Loops
  + Human-in-the-loop steps
  + Multi-agent coordination
  + Asynchronous or event-driven execution

**Should We Still Use LangChain?**

* **Yes**. LangGraph is built **on top of LangChain** - it doesn't replace it.
* You'll still use **LangChain components** like:
  + ChatOpenAI (LLMs)
  + PromptTemplate
  + Retrievers
  + Document Loaders
  + Tools, etc.
* LangGraph handles **workflow orchestration**, while LangChain provides the **building blocks** for each step in that workflow.

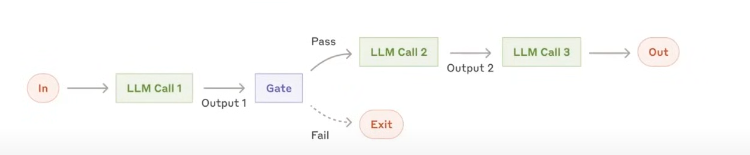
**LangGraph Core Concepts**

**LLM Workflows:**

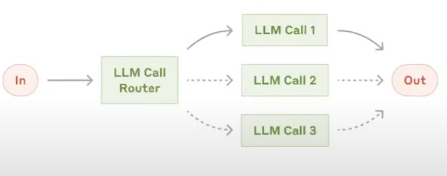
* LLM workflows are a step by step process using which we can build complex LLM applications.
* Each step in a workflow performs a distinct task - such as prompting, reasoning, tool calling, memory access, or decision-making.
* Workflows can be linear, parallel, branched, or looped, allowing for complex behaviours like retries, multi-agent communication, or tool-augmented reasoning.

**Common workflows:**

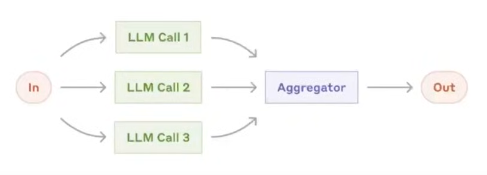
1. Prompt Chaining



1. Routing



1. Parallelization



1. Orchestrator Workers



1. Evaluator Optimizer



**Graphs:**

* Represents the **overall workflow**.
* Made up of **nodes** (actions) and **edges** (transitions).
* Allows for **loops**, **conditional routing**, and **parallelism**.
* Can model **single-agent pipelines** or **multi-agent interactions**.
* Think of the graph as the blueprint or control flow diagram of your LLM application.

**Nodes:**

* Individual **units of computation** (e.g., a function, tool call, or LLM).
* Each node performs an operation and updates the **shared state**.
* Can be:
  + Simple functions
  + Agents
  + LangChain chains or tools
* A node is a step or action in the workflow.

**Edges:**

* Define **how execution moves** from one node to another.
* Types:
  + **Static**: Always go to the same next node.
  + **Conditional**: Route based on state or output.
  + **Looping**: Allow repetition of nodes.
* Edges determine logic and control flow between steps.

**Reducers:**

* Reducers in LangGraph define how updates from nodes are applied to the shared state.
* Each key in the state can have its own reducer, which determines whether **new data replaces, merges, or adds to the existing value.**
* Essential in **parallel** or **multi-agent** workflows.

**LangGraph Execution Model:**

1. **Graph Definition**

* You define:
  + The **state schema**
  + **Nodes** (functions that perform tasks)
  + **Edges** (which node connects to which)

1. **Compilation**

* You call .compile() on the StateGraph.
* This checks the graph structure and prepares it for execution.

1. **Invocation**

* You run the graph with .invoke(initial\_state).
* LangGraph sends the initial state as a **message** to the entry node(s).

1. **Super-Steps Begin**

* Execution proceeds in rounds.
* In each round (super-step):
  + All **active nodes** (those that received messages) run **in parallel**
  + Each returns an **update** (message) to the state

1. **Message Passing & Node Activation**

* The messages are passed to downstream nodes via edges.
* Nodes that receive messages become active for the **next round.**

1. **Halting Condition**

* Execution stops when:
  + No nodes are active, and
  + No messages are in transit

**Persistence in LangGraph**

* **Persistence** refers to the ability to **store and resume** the execution state of a graph over time. This is essential for long-running, multi-step workflows or agentic processes that may:
  + Span multiple user interactions
  + Be interrupted and resumed later
  + Involve waiting for external input (like a user form or an API response)
  + Need recovery after failure.

**What does persistence enable ?**

1. **Checkpointing:** The state of the graph can be saved at each step.
2. **Pausing & Resuming:** You can pause execution (e.g., awaiting user input) and later resume it from the same point.
3. **State Recovery:** If the system crashes or restarts, it can pick up where it left off.
4. **Multi-turn Interaction:** Great for agent workflows where decisions depend on external events or human feedback.

**Benefits of persistence:**

1. Short term memory
2. Fault Tolerance
3. Human in the Loop (HITL)
4. Time Travel

**Short Term Memory**

* Persistence lets a graph “remember” what happened earlier in the same execution flow, even if you stop and resume it.
* It enables stateful conversations or workflows without keeping everything in RAM.
* You don’t need to re-send the entire conversation history manually — the graph state is the memory.

**Fault Tolerance**

* If the process crashes, the graph can recover from the last saved checkpoint instead of restarting from the beginning.
* Saves compute time and money.
* Essential for production workflows that can’t risk losing partial progress.

**Human in the Loop**

* The graph can pause mid-execution, wait for human input, and later resume without forgetting its progress.
* Enables multi-day decision-making workflows.
* Perfect for review/approval pipelines.

**Time Travel**

* Because each state checkpoint is saved, you can rewind to a specific point in execution and rerun from there — similar to restoring a saved game.
* Debugging and testing are much faster.
* Lets you experiment with different branches of execution without repeating all prior steps.

| **Sr.no** | **Benefit** | **What it does** | **Real-World Win** |
| --- | --- | --- | --- |
| 1 | Short-term memory | Keeps context within a single workflow | Continue chats/workflows without reloading all data |
| 2 | Fault Tolerance | Resumes from last checkpoint after crash | Saves time, avoids starting over |
| 3 | Human in the loop | Pauses until human input arrives | Multi-day or approval-based workflows |
| 4 | Time Travel | Rewinds to a past execution point | Debugging, A/B testing parts of a flow |