Understanding Agents and Agentic Systems

- Building effective agents
- LangChain vs. LangGraph

This document provides an overview of **agents**, **workflows**, and **agentic systems** — including when to use them, how to design them, and common implementation patterns.

What Are Agents?

The term **"agent"** can refer to different levels of system autonomy:

- **Fully autonomous systems** that independently use tools over extended periods to accomplish complex tasks.
- **Prescriptive implementations** that follow predefined workflows.

Both are categorized as **agentic systems**, but with a key architectural distinction:

System Type	Description
Workflows	LLMs and tools are orchestrated through **predefined code paths**.
Agents	LLMs **dynamically direct their own processes** and **decide** how to accomplish tasks.

When (and When Not) to Use Agents

Start with the **simplest solution possible** — add complexity only when needed.

Use Agents When:

- Tasks are **open-ended** or **hard to predict**.
- You need **flexible, model-driven decision-making**.
- The system must operate **autonomously at scale**.

Avoid Agents When:

- Simpler workflows or single LLM calls (with retrieval/in-context examples) suffice.
- Cost, latency, or reliability are bigger priorities than flexibility.

Tradeoff:

Agentic systems often **trade latency and cost for performance** — use them when that tradeoff makes sense.

Frameworks for Building Agents

There are many frameworks that simplify agentic development:

- LangGraph (LangChain)

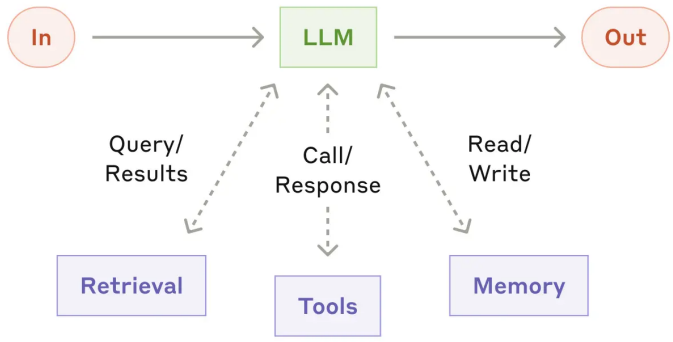
- Amazon Bedrock AI Agent Framework
- Rivet – Drag-and-drop LLM workflow builder
- Vellum – GUI for complex workflow testing

⚠ **Caution:**
Frameworks add abstraction layers that can obscure the underlying prompts and responses, making debugging harder.
Start by using **LLM APIs directly** — many patterns can be implemented in just a few lines of code.

🧠 Building Blocks, Workflows, and Agents

Agentic systems can be viewed as a **spectrum of complexity**, from simple augmented models to fully autonomous agents.

🧩 1. Building Block: The Augmented LLM



An **augmented LLM** is an LLM enhanced with:

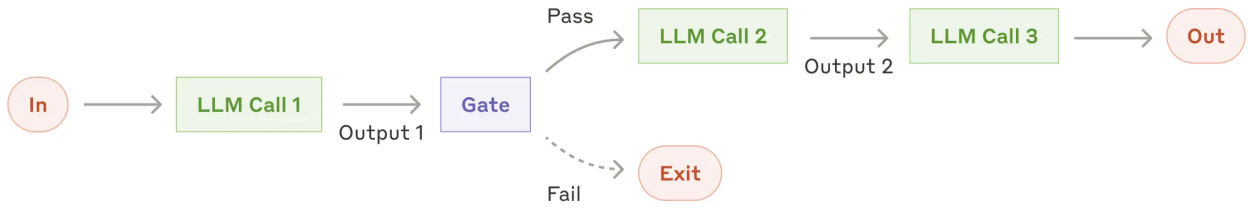
- **Retrieval** (search/query access)
- **Tools** (API calls, code execution, etc.)
- **Memory** (context retention across sessions)

These capabilities allow LLMs to:

- Generate their own queries
- Choose and use tools dynamically
- Retain and reference prior information

> **Tip:** Tailor augmentations to your use case and expose them through a clear, documented interface.
> The Model Context Protocol (MCP) is a good integration approach.

🔗 2. Workflow: Prompt Chaining



Prompt chaining decomposes a task into a sequence of steps, where each LLM call processes the output of the previous one. You can add programmatic checks (see "gate" in the diagram below) on any intermediate steps to ensure that the process is still on track.

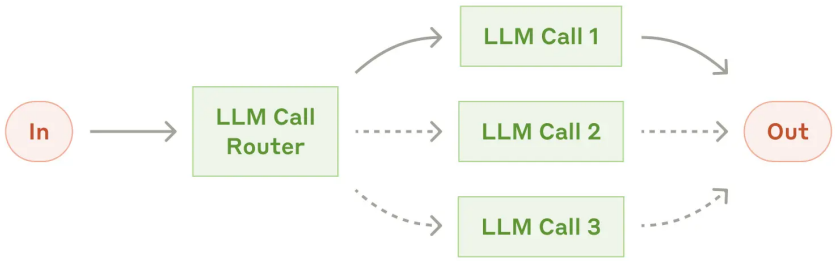
Use When:
This workflow is ideal for situations where the task can be easily and cleanly decomposed into fixed subtasks. The main goal is to trade off latency for higher accuracy, by making each LLM call an easier task.

Examples:

- Generate marketing copy → Translate it → Refine tone
- Write a document outline → Validate it → Draft full text

---

3. Workflow: Routing



Classify an input and **route it** to a specialized process or model.

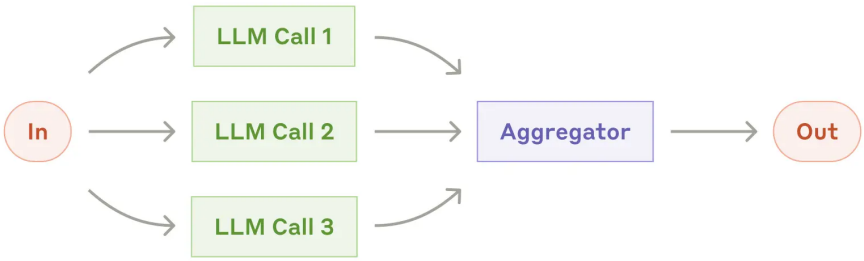
Use When:
You have **distinct input categories** requiring **different handling**.

Examples:

- Route customer queries (refunds, tech support, general)
- Send easy tasks to a small model, hard ones to a larger model

---

⚙ 4. Workflow: Parallelization



Run multiple LLMs **simultaneously** and aggregate results.

Variants:

- **Sectioning:** Split task into parallel subtasks.
- **Voting:** Running the same task multiple times to get diverse outputs.

⚙ Workflow: Parallelization

🧠 When to Use

Parallelization is effective when:

- Subtasks can be **divided and run in parallel** for faster execution.
- Multiple **perspectives or attempts** are needed to increase confidence in results.
- Complex tasks involve **multiple considerations**, each best handled by a **dedicated LLM call** for focused attention.

🔍 Key Variations

1. Sectioning

Split a task into **independent subtasks** that run simultaneously.

Use Cases:

- **Guardrails:**
  - One LLM processes user queries.
  - Another LLM screens for inappropriate or unsafe content.
  - Improves reliability versus combining both in one model call.

- **Automated Evals:**
  - Each LLM instance evaluates a **different aspect** of model performance on a given prompt.

---

2. Voting

Run the **same task multiple times** to gather diverse outputs and reach consensus.

Use Cases:

- **Code Review:**
  - Multiple prompts independently check code for vulnerabilities; issues flagged if any detect a problem.
- **Content Evaluation:**
  - Several LLMs assess whether content is inappropriate.
  - Uses different **criteria or thresholds** to balance false positives and negatives.

☑ Summary

Technique	Purpose	Benefit
Sectioning	Split independent subtasks	Speed & specialization
Voting	Repeat same task multiple times	Higher confidence & robustness

---

5. Workflow: Orchestrator-Workers

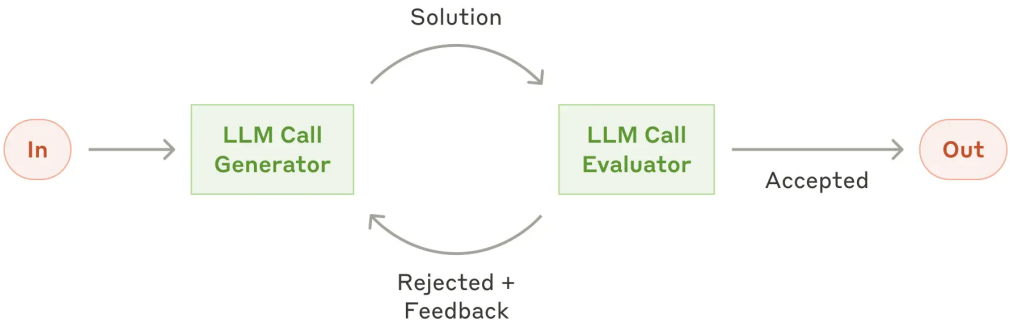
A **central LLM** (the orchestrator) dynamically breaks tasks down and delegates them to **worker LLMs**.

Use When:
This workflow is well-suited for complex tasks where you can't predict the subtasks needed (in coding, for example, the number of files that need to be changed and the nature of the change in each file likely depend on the task). Whereas it's topographically similar, the key difference from parallelization is its flexibility—subtasks aren't pre-defined, but determined by the orchestrator based on the specific input

Examples:

- Multi-file coding agents
- Search and synthesis from multiple data sources

🔁 6. Workflow: Evaluator-Optimizer



Two LLMs interact in a **feedback loop** — one produces, the other evaluates. The **Evaluator-Optimizer** workflow involves two LLMs (or two roles of the same LLM) working together in a **feedback loop**:

- One acts as the **Optimizer**, generating initial responses or outputs.
- The other serves as the **Evaluator**, reviewing those outputs against predefined criteria and providing constructive feedback.
  This process continues iteratively until the output meets the desired standard or quality threshold.

Use When:
You have clear evaluation criteria and iterative improvement helps. This workflow is particularly effective when:

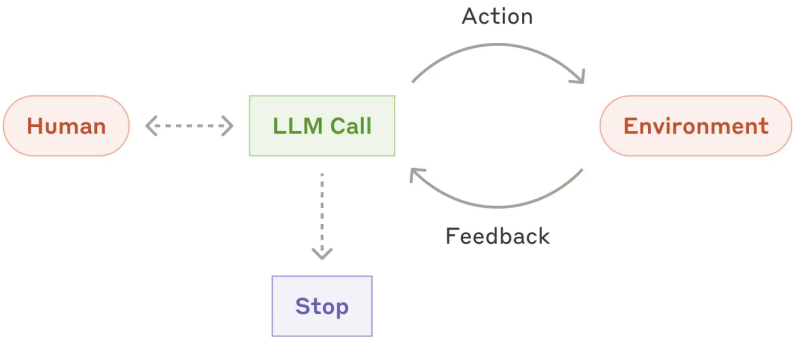
- You have **clear evaluation criteria** or measurable quality standards.
- **Iterative refinement** leads to **tangible improvements** in results.

It mirrors how a human writer drafts, reviews, and revises content — each iteration improving upon the last.

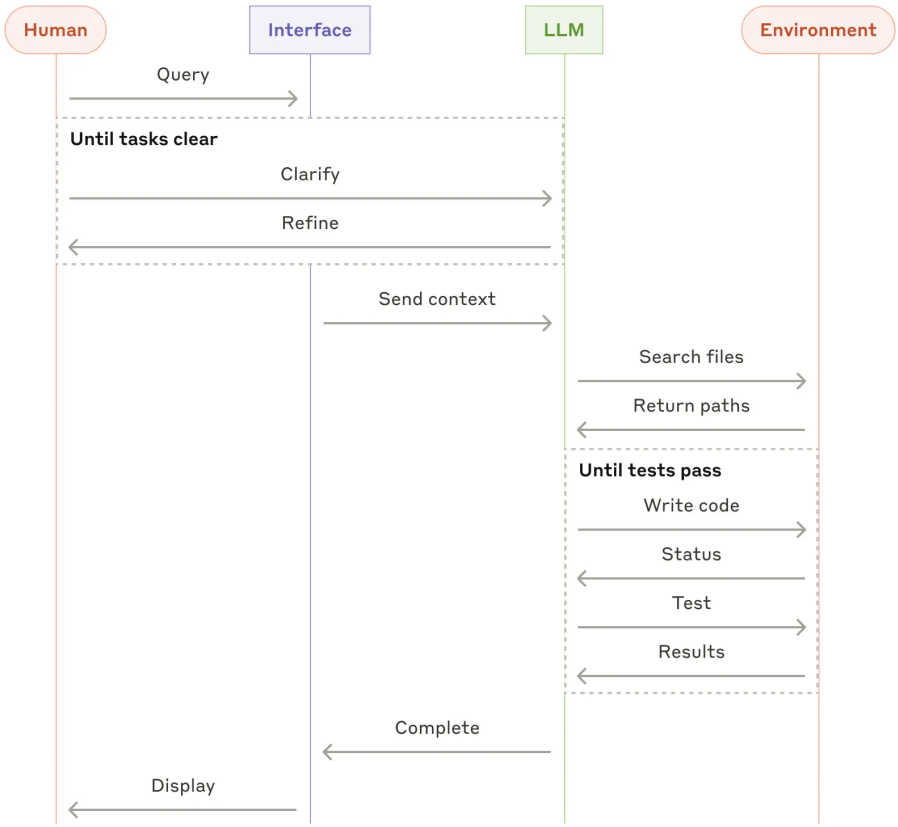
Examples:

- Literary translation refinement
- Iterative search or summarization tasks

🤖 7. Agents (Autonomous Systems)



As LLMs improve in reasoning, planning, and tool use, **agents** are emerging in production.



How Agents Work

1. Start with a **user command or discussion**.
2. **Plan and operate independently**, using tools and environment feedback.
3. **Pause for human input** when needed (e.g., blockers, checkpoints).
4. **Stop** upon task completion or after predefined conditions.

Agents are essentially:

> LLMs that loop between reasoning, taking actions (via tools), and observing results.

🧱 Implementation Notes

- Define **clear toolsets** and documentation.
- Include **stopping conditions** (e.g., max iterations).
- Test in **sandboxed environments** with guardrails.

☑ Use Agents For:

- **Open-ended, multi-step** problems
- **Dynamic tool use** and **planning**
- **Scaling complex automation** in trusted environments

⚠ Be Aware:

- Agents are **expensive** (multiple tool calls and iterations)
- **Compounding errors** can occur
- Require **trust and safety checks**

Examples:

- Coding agent that edits multiple files for SWE-bench tasks
- "Computer use" agents that operate desktop environments

🧩 Combining and Customizing Patterns

These workflows are **building blocks**, not rigid templates.
You can **mix and match** to suit your needs:

- Chain + Routing for adaptive pipelines
- Orchestrator + Evaluator for supervised autonomous systems

> Measure performance, iterate often, and **add complexity only when it improves results**.

🧭 Summary & Best Practices

Building the **right system** matters more than building the **most complex system**.

Core Principles for Agents

1. 🧠 **Keep it simple** – Add autonomy only when beneficial.
2. 🔍 **Be transparent** – Make planning steps visible.
3. 🧰 **Design your agent-computer interface (ACI)** carefully – Document and test all tools.

LangChain vs LangGraph — The Next Generation of LLM Workflow Orchestration

---

📘 Introduction

Modern LLM applications are moving from simple, sequential prompt chains to **complex, stateful, and multi-agent systems**.
While **LangChain** laid the foundation for LLM-driven pipelines, it struggles when workflows require **loops, persistence, human involvement, or fault recovery**.

To overcome these challenges, **LangGraph** was created — a **graph-based orchestration framework** built on top of LangChain.
It brings **state management**, **event-driven execution**, **fault tolerance**, and **observability** to complex LLM workflows.

This document explains:

- What LangChain is and its challenges
- How LangGraph solves them
- Key differences and when to use each

What is LangChain?

LangChain is a framework for building applications powered by Large Language Models (LLMs).
It provides tools and abstractions for connecting:

- LLMs (OpenAI, Anthropic, etc.)
- Prompts and Chains
- Tools and Agents
- Memory and Retrieval
- Data Loaders and Vector Stores

LangChain workflows are **sequential**, flowing step-by-step: Input → Preprocess → Retrieve → Prompt → LLM → Output

It's perfect for:

- Chatbots
- RAG (Retrieval-Augmented Generation)
- Summarization
- Information extraction
- Early prototypes and research

⚠ Challenges in LangChain

While LangChain is powerful, it was designed for **sequential workflows**, not **dynamic, persistent, or fault-tolerant systems**.
Below are its key limitations:

#	Challenge	Description
1	**Control Flow Complexity**	Hard to implement loops, branching, or conditional paths — designed mainly for straight chains.

#	Challenge	Description
2	**Handling State**	State (context, memory, variables) is limited and not durable across sessions or restarts.
3	**Event-Driven Execution**	Workflows execute in sequence; no event-based triggers or reactivity.
4	**Fault Tolerance**	No built-in retry, rollback, or recovery mechanisms — if a step fails, the whole chain must restart.
5	**HITL (Human-in-the-Loop)**	Integrating human feedback or approval steps is manual and complex.
6	**Nested Workflows**	Difficult to manage or visualize nested or recursive workflows.
7	**Observability**	Limited runtime insights — debugging or understanding workflow behavior is challenging.

🔁 What is LangGraph?

LangGraph is a next-generation orchestration library built on top of LangChain.
It turns LLM pipelines into **graphs** instead of linear chains.

Key Concepts:

- **Nodes** → Actions or agents
- **Edges** → Data or state flow between nodes
- **State** → Shared object representing workflow progress
- **Checkpoints** → Savepoints for resuming execution

LangGraph supports:

- **Branching**, **loops**, and **conditional flows**
- **Persistent state** and **resumable workflows**
- **Event-driven execution**
- **Automatic retries** and **failure recovery**
- **Human-in-the-loop (HITL)** support
- **Deep observability** and debugging tools

🧠 How LangGraph Solves LangChain's Limitations

Challenge	LangChain Limitation	LangGraph Solution
Control Flow Complexity	Sequential, limited branching	Graph-based flow enables conditional routing, loops, and dynamic branching
Handling State	Memory is transient	Persistent state objects with checkpointing and recovery

Challenge	LangChain Limitation	LangGraph Solution
Event-Driven Execution	Sequential, not reactive	Event-driven nodes trigger on changes or external inputs
Fault Tolerance	No retry or rollback	Automatic retry, rollback, and checkpoint recovery built-in
HITL (Human-in-the-Loop)	Manual integration required	Native pause, review, and resume workflows
Nested Workflows	Hard to nest chains	Supports subgraphs for modular, reusable workflows
Observability	Limited logs	Real-time state tracking, tracing, and visualization

🚀 Why We Need LangGraph

LLM systems are increasingly **interactive**, **stateful**, and **dynamic**.
LangChain's linear model struggles to manage:

- Multiple agents interacting in loops
- Conditional branching based on model output
- Long-running sessions with user intervention
- Error handling, retries, and resumption
- Live monitoring and debugging

LangGraph introduces the control and resilience needed for **production-grade AI applications**, not just prototypes.

🔑 Key Differences

Feature	LangChain	LangGraph
Architecture	Linear Chains	Graph-based Workflow
Control Flow	Sequential	Conditional / Looping
State Handling	Transient Memory	Persistent, Serializable State
Fault Tolerance	Manual	Built-in Retry & Recovery
Execution Model	Step-by-step	Event-driven
Human-in-the-Loop	Manual integration	Native pause/resume support
Nested Workflows	Difficult	Supported (subgraphs)
Observability	Limited logs	Full runtime introspection
Use Case	RAG, Chatbots, Prototypes	Multi-agent, Stateful, Complex Systems

🧰 When to Use Each

☑ Use **LangChain** when:

- You're building a **prototype** or **simple linear workflow**
- You need **fast iteration and simplicity**
- Your task fits a **retrieve** → **generate** → **output** pattern

🚀 Use **LangGraph** when:

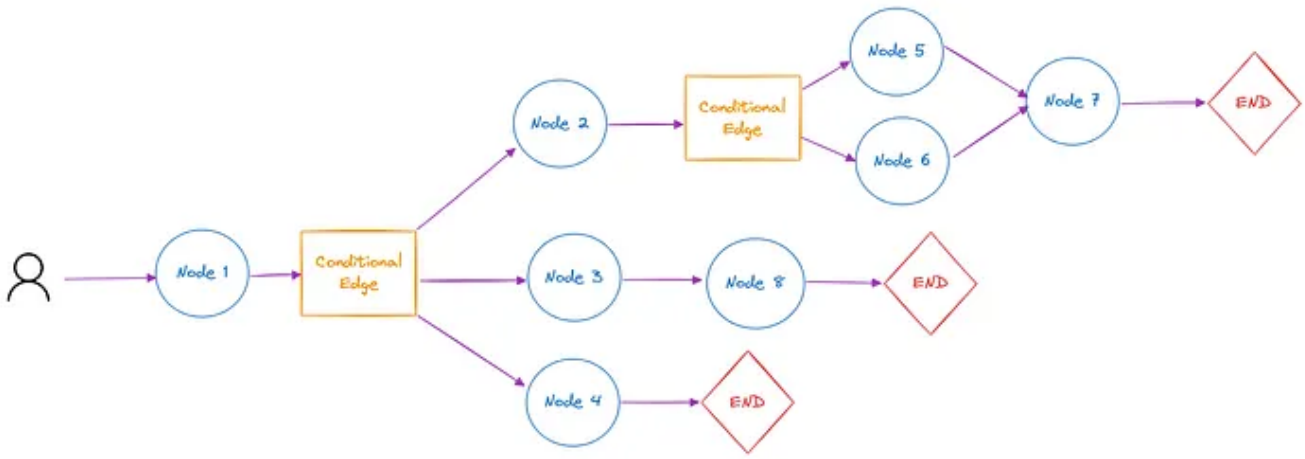
- You need **branching**, **loops**, or **dynamic routing**
- You're building **multi-agent** or **long-running** workflows
- You want **fault tolerance** and **state persistence**
- You need **HITL (Human-in-the-Loop)** interaction
- You want **monitoring**, **debugging**, and **workflow visualization**

🧪 Conceptual Example

LangChain (Linear)

LangChain: User → Retriever → Prompt → LLM → Output

LangGraph:



🏁 Summary

LangChain Strengths	LangGraph Advantages
Quick prototyping	Production-ready orchestration
Simple linear flows	Dynamic, event-driven graphs
Great for demos	Great for real-world systems
Minimal setup	Advanced state, retry, and HITL support

In short:

> **LangChain** helps you build LLM pipelines.
> **LangGraph** helps you orchestrate resilient, intelligent LLM systems.

References

- LangChain Official Docs
- LangGraph Official Docs
- LangChain Blog: Why LangGraph
- Medium: LangChain vs LangGraph Analysis
- DuploCloud Blog: LangChain vs LangGraph

LangGraph API Reference

LangGraph is a **message-passing framework** for building modular, graph-based AI workflows and agent systems. It orchestrates complex workflows by connecting logical components (**nodes**) and controlling their interactions via **edges** and **state**.

Table of Contents

Core Concepts

1. State

It is the shared memory that flows through your workflow.It holds all the data being passed between nodes as your graph runs. A **State** represents the **current data snapshot** in your graph at any point in execution. It's passed between nodes as input and output, making it the central communication mechanism.

> Think of it as a shared message that evolves as nodes process and update data.

Example:

```
from typing_extensions import TypedDict

class State(TypedDict):
    user_input: str
    result: str
```

2. 🧱 Schema

A **Schema** defines the structure of the state — ensuring consistent communication between nodes.

Supported schema types:

- ☑ **TypedDict** (fast and simple, recommended)
- ☑ **dataclass** (adds default values)
- ⚙ **Pydantic BaseModel** (adds validation, slower)

Example:

```
from typing_extensions import TypedDict
from dataclasses import dataclass

class InputState(TypedDict):
    question: str

class OutputState(TypedDict):
    answer: str

@dataclass
class OverallState:
    question: str
    answer: str = ""
```

3. ⚙ Nodes and Edges

LangGraph workflows are graphs composed of:

- **Nodes** → perform computation or logic (LLM calls, APIs, etc.)
- **Edges** → define the control flow between nodes

> 🧩 **"Nodes do the work. Edges tell what to do next."**

Example:

```
from langgraph.graph import StateGraph, START, END
from typing_extensions import TypedDict

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

graph = StateGraph(State)

def greet_node(state: State):
    return {"output": f"Hello, {state['input']}!"}

graph.add_node("greet_node", greet_node)
```

```
graph.add_edge(START, "greet_node")
graph.add_edge("greet_node", END)

app = graph.compile()
print(app.invoke({"input": "LangGraph"}))
# Output: {'input': 'LangGraph', 'output': 'Hello, LangGraph!'}
```

🏗️ Building a Graph

Step-by-Step Process

1. **Define your State**
2. **Add Nodes** (functions)
3. **Connect them using Edges**
4. **Compile the graph** ⚠️ (required before running)

```
from langgraph.graph import StateGraph, START, END
from typing_extensions import TypedDict

# 1. Define state
class State(TypedDict):
    input: str
    output: str

# 2. Create builder
builder = StateGraph(State)

# 3. Add nodes
builder.add_node("process", process_node)
builder.add_node("finalize", finalize_node)

# 4. Add edges
builder.add_edge(START, "process")
builder.add_edge("process", "finalize")
builder.add_edge("finalize", END)

# 5. Compile (REQUIRED!)
graph = builder.compile()

# 6. Run
result = graph.invoke({"input": "hello"})
```

⚙️ How LangGraph Executes

- Execution happens in **super-steps** (inspired by Google's Pregel)
- Each node runs when it receives a new message (state)
- **Parallel nodes** run in the same super-step
- **Sequential nodes** run in separate super-steps

- Execution ends when all nodes are inactive and no messages remain

---

Reducers

Reducers define **how node updates modify 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.

Default Reducer (Overwrite)

If not specified, updates simply **overwrite** existing values.

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

# Update behavior:
# Input: {"foo": 1, "bar": ["hi"]}
# Node returns {"foo": 2}
# Result: {"foo": 2, "bar": ["hi"]}
```

Custom Reducer (Append/Combine)

Use `Annotated` to specify a reducer function.

```
from typing import Annotated
from operator import add

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

# Update behavior:
# Input: {"foo": 1, "bar": ["hi"]}
# Node returns {"bar": ["bye"]}
# Result: {"foo": 1, "bar": ["hi", "bye"]}
```

Overwrite Reducer

Use `Overwrite` to bypass reducers and replace data directly.

```
from langgraph.types import Overwrite

# Use when you need to force overwrite
return {"key": Overwrite(new_value)}
```

💬 Working with Messages

LangGraph supports **chat-style message passing**, ideal for LLM conversations.

Using add_messages

```
from langchain.messages import AnyMessage, HumanMessage
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]

# Supported input formats:
{"messages": [HumanMessage(content="Hello!")]}
# OR
{"messages": [{"type": "human", "content": "Hello!"}]}
```

Benefits of `add_messages`:

- ☑ Appends new messages
- ☑ Updates existing messages by ID
- ☑ Auto-deserializes messages into LangChain Message objects

MessagesState (Built-in)

Pre-built state for chat applications:

```
from langgraph.graph import MessagesState

class State(MessagesState):
    documents: list[str]  # Add custom fields
    user_id: str
```

🧠 Nodes

Nodes are Python functions (sync or async) that process state.

Basic Node

```
def my_node(state: State):
    return {"result": f"Processed: {state['input']}"}
```

Node with Config

```
from langchain_core.runnables import RunnableConfig

def node_with_config(state: State, config: RunnableConfig):
    thread_id = config["configurable"]["thread_id"]
    current_step = config["metadata"]["langgraph_step"]
    return {"result": f"Processing at step {current_step}"}
```

Node with Runtime Context

```
from langgraph.runtime import Runtime
from dataclasses import dataclass

@dataclass
class Context:
    user_id: str
    llm_provider: str = "openai"

def node_with_runtime(state: State, runtime: Runtime[Context]):
    print(f"User: {runtime.context.user_id}")
    print(f"Provider: {runtime.context.llm_provider}")
    return {"result": "Done"}
```

Special Nodes

Node	Purpose
START	Entry point of the graph (receives user input)
END	Terminal node (marks workflow completion)

```
from langgraph.graph import START, END

graph.add_edge(START, "first_node")
graph.add_edge("last_node", END)
```

Node Caching

Cache expensive computations to improve performance:

```
import time
from langgraph.cache.memory import InMemoryCache
from langgraph.types import CachePolicy

def expensive_node(state: State):
    time.sleep(2)  # Expensive operation
```

```
    return {"result": state["x"] * 2}

builder.add_node(
    "expensive_node",
    expensive_node,
    cache_policy=CachePolicy(ttl=60)  # Cache for 60 seconds
)

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

# First call: slow (2 seconds)
graph.invoke({"x": 5})

# Second call: instant (cached)
graph.invoke({"x": 5})
```

Edges and Control Flow

1. Normal Edges

Direct transition from one node to another:

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

2. Conditional Edges

Route dynamically based on state:

```
def router(state: State):
    if state["score"] > 5:
        return "node_b"
    return "node_c"

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

With mapping dictionary:

```
def router(state: State):
    return state["score"] > 5

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

3. Entry Point

Define where the graph starts:

```
from langgraph.graph import START

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

4. Conditional Entry Point

Start at different nodes based on logic:

```
def entry_router(state: State):
    if state["is_premium"]:
        return "premium_flow"
    return "standard_flow"

graph.add_conditional_edges(START, entry_router)
```

🚀 Advanced Features

Command (State Update + Routing)

Combine state updates and control flow in one step:

```
from langgraph.types import Command
from typing import Literal

def my_node(state: State) -> Command[Literal["next_node"]]:
    return Command(
        update={"foo": "bar"},  # Update state
        goto="next_node"         # Route to next node
    )
```

When to use Command:

- ☑ Need to update state AND route in the same node
- ☑ Implementing multi-agent handoffs
- ☑ Dynamic workflow control

When to use Conditional Edges:

- ☑ Route between nodes without updating state

Send (Dynamic Parallelism)

Create parallel branches dynamically (map-reduce pattern):

```
from langgraph.types import Send

def fan_out(state: State):
    # Process each item in parallel
    return [
        Send("process_item", {"item": item})
        for item in state['items']
    ]

graph.add_conditional_edges("fan_out_node", fan_out)
```

Multiple Schemas

Use different schemas for input/output/internal nodes:

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

class OutputState(TypedDict):
    final_result: str

class InternalState(TypedDict):
    user_input: str
    final_result: str
    intermediate_data: str  # Only used internally

builder = StateGraph(
    InternalState,
    input_schema=InputState,
    output_schema=OutputState
)
```

Context Schema

Pass runtime information (like model provider) to nodes:

```
from dataclasses import dataclass
from langgraph.runtime import Runtime

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

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

```
# Invoke with context
graph.invoke(
    {"input": "Test"},
    context={"llm_provider": "anthropic", "user_id": "user123"}
)

# Access in node
def my_node(state: State, runtime: Runtime[ContextSchema]):
    provider = runtime.context.llm_provider
    user_id = runtime.context.user_id
    return state
```

Recursion Limit

Prevent infinite loops with recursion limits (default = 25 steps):

```
# Set custom limit
graph.invoke(inputs, config={"recursion_limit": 50})
```

Monitor and handle proactively:

```
from langchain_core.runnables import RunnableConfig

def reasoning_node(state: dict, config: RunnableConfig):
    current_step = config["metadata"]["langgraph_step"]
    limit = config["recursion_limit"]

    # Check if approaching limit (80% threshold)
    if current_step >= limit * 0.8:
        return {**state, "route_to": "fallback"}

    # Normal processing
    return {"messages": state["messages"] + ["thinking..."]}
```

Available metadata:

```
def inspect_metadata(state: dict, config: RunnableConfig):
    metadata = config["metadata"]

    print(f"Step: {metadata['langgraph_step']}")
    print(f"Node: {metadata['langgraph_node']}")
    print(f"Triggers: {metadata['langgraph_triggers']}")
    print(f"Path: {metadata['langgraph_path']}")

    return state
```

Complete Examples

Example 1: Simple Greeting Bot

```
from langgraph.graph import StateGraph, START, END
from typing_extensions import TypedDict

class State(TypedDict):
    name: str
    greeting: str

def greet(state: State):
    return {"greeting": f"Hello, {state['name']}!"}

graph = StateGraph(State)
graph.add_node("greet", greet)
graph.add_edge(START, "greet")
graph.add_edge("greet", END)

app = graph.compile()
print(app.invoke({"name": "Alice"}))
# Output: {'name': 'Alice', 'greeting': 'Hello, Alice!'}
```

Example 2: Conditional Routing

```
from langgraph.graph import StateGraph, START, END

class State(TypedDict):
    score: int
    result: str

def evaluate(state: State):
    return {"result": "Evaluating..."}

def pass_node(state: State):
    return {"result": "Passed!"}

def fail_node(state: State):
    return {"result": "Failed!"}

def router(state: State):
    return "pass" if state["score"] >= 50 else "fail"

builder = StateGraph(State)
builder.add_node("evaluate", evaluate)
builder.add_node("pass", pass_node)
builder.add_node("fail", fail_node)

builder.add_edge(START, "evaluate")
builder.add_conditional_edges("evaluate", router)
```

```
builder.add_edge("pass", END)
builder.add_edge("fail", END)

graph = builder.compile()

print(graph.invoke({"score": 75}))
# Output: {'score': 75, 'result': 'Passed!'}

print(graph.invoke({"score": 30}))
# Output: {'score': 30, 'result': 'Failed!'}
```

Example 3: Using Command

```
from langgraph.graph import StateGraph, START, END
from langgraph.types import Command
from typing import Literal

class State(TypedDict):
    count: int
    status: str

def increment(state: State) -> Command[Literal["check", "finalize"]]:
    new_count = state["count"] + 1

    if new_count >= 3:
        return Command(
            update={"count": new_count},
            goto="finalize"
        )

    return Command(
        update={"count": new_count},
        goto="check"
    )

def check(state: State) -> Command[Literal["increment"]]:
    return Command(
        update={"status": f"Count is {state['count']}"},
        goto="increment"
    )

def finalize(state: State):
    return {"status": f"Done! Final count: {state['count']}"}

builder = StateGraph(State)
builder.add_node("increment", increment)
builder.add_node("check", check)
builder.add_node("finalize", finalize)

builder.add_edge(START, "increment")
builder.add_edge("finalize", END)
```

```
graph = builder.compile()
print(graph.invoke({"count": 0}))
# Output: {'count': 3, 'status': 'Done! Final count: 3'}
```

Example 4: Chat with Messages

```
from langgraph.graph import StateGraph, MessagesState, START, END
from langchain.messages import HumanMessage, AIMessage

class State(MessagesState):
    pass

def chatbot(state: State):
    user_msg = state["messages"][-1].content
    response = f"You said: {user_msg}"
    return {"messages": [AIMessage(content=response)]}

builder = StateGraph(State)
builder.add_node("chatbot", chatbot)
builder.add_edge(START, "chatbot")
builder.add_edge("chatbot", END)

graph = builder.compile()

result = graph.invoke({
    "messages": [HumanMessage(content="Hello!")]
})

for msg in result["messages"]:
    print(f"{msg.type}: {msg.content}")
# Output:
# human: Hello!
# ai: You said: Hello!
```

Summary Table

Concept	Purpose	Example
State	Shared data structure	`class State(TypedDict): ...`
Schema	Defines structure of State	TypedDict, Pydantic, dataclass
Node	Function that processes data	`def node(state): return {...}`
Edge	Connects nodes	`graph.add_edge("A", "B")`
Command	Combines state + routing	`return Command(update=..., goto=...)`
Send	Dynamic parallel execution	`return [Send("node", {...})]`

Concept	Purpose	Example
Reducer	Merges state updates	`Annotated[key, add]`
Cache	Caches node outputs	`InMemoryCache()`
Context	Runtime data	`Runtime[ContextSchema]`
Messages	Conversation history	`add_messages`
Recursion Limit	Prevents loops	`config={"recursion_limit": 10}`

Key Takeaways

- ☑ **LangGraph** = Graph-based orchestration for AI agents
- ☑ **State** carries shared data between nodes
- ☑ **Schema** defines structure and ensures consistency
- ☑ **Nodes** execute logic (can be any Python function)
- ☑ **Edges** define control flow
- ☑ **Reducers** control how state updates are merged
- ☑ **Command** combines state updates and routing
- ☑ **Send** enables dynamic parallel execution
- ☑ **Context** passes runtime information to nodes
- ☑ **Caching** improves performance for expensive operations

Build transparent, modular, and scalable LLM-powered systems — one node at a time.

Additional Resources

- LangGraph Documentation
- LangGraph GitHub
- LangChain Documentation

Made with ♥ for building intelligent agent workflows

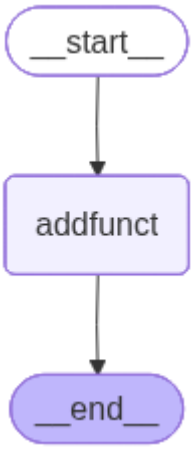
CODES

Sequential worflow

1. BASIC LangGraph additon code

```
# Import necessary modules from langgraph and typing
from langgraph.graph import StateGraph, START, END
from typing import TypedDict

# 👉 1) Define a TypedDict to describe the structure of our state data
class AddState(TypedDict):
```

```
    a: int  # First input number
    b: int  # Second input number
    c: int  # Result of the multiplication (output)

# 👉 2) Create a new state graph using the defined state structure
graph = StateGraph(AddState)

# Define a function that performs the main computation
def addfunct(state: AddState) -> AddState:
    """
    Multiplies 'a' and 'b' from the state and stores the result in 'c'.
    Returns the updated state.
    """
    a = state['a']
    b = state['b']
    c = a * b  # Perform multiplication
    state['c'] = c  # Store result in the state
    return state

# 👉 3) Add Nodes
# Add the computation function as a node in the graph
graph.add_node("addfunct", addfunct)

# 👉 4) Add Edges
# Define the workflow connections:
# - Start from START node
# - Go to 'addfunct' node
# - End at END node
graph.add_edge(START, "addfunct")
graph.add_edge("addfunct", END)

# 👉 5) Compile
# Compile the workflow graph into an executable workflow
workflow = graph.compile()

# 👉 4) Run
# Invoke the workflow with input values for 'a' and 'b'
result = workflow.invoke({"a": 40, "b": 40})

from IPython.display import Image

# Call the method to get the PNG bytes
Image(workflow.get_graph().draw_mermaid_png())
```



2. Calculate BMI

```
# Import necessary modules
from langgraph.graph import StateGraph, START, END
from typing import TypedDict

# Define a TypedDict to represent the data structure (state) used in the workflow
class BmiState(TypedDict):
    weight: float     # User's weight in kilograms
    height: float     # User's height in meters
    bmi: float        # Calculated Body Mass Index
    category: str     # BMI classification (e.g., Normal, Overweight, etc.)

# Define a function to calculate BMI based on the weight and height
# Har node function ek hi signature follow karta hai:
# 👉 Input: state (dict type)
# 👉 Output: updated state (dict type)
def calc_bmi(state: BmiState) -> BmiState:
    """
    Calculates BMI using the formula: BMI = weight / (height ** 2)
    Adds the result to the state dictionary under 'bmi'.
    """
    weight = state['weight']
    height = state['height']

    # Calculate BMI and round to 2 decimal places
    bmi = weight / (height ** 2)
    state['bmi'] = round(bmi, 2)

    return state

# Define a function to classify BMI into different categories
def level_bmi(state: BmiState) -> BmiState:
    """
    Determines the BMI category based on the calculated BMI value.
    Updates the 'category' field in the state.
    """
    bmi = state['bmi']
```

```
    if bmi < 18.5:
        state['category'] = "Underweight"
    elif 18.5 <= bmi < 25:
        state['category'] = "Normal"
    elif 25 <= bmi < 30:
        state['category'] = "Overweight"
    else:
        state['category'] = "Obese"

    return state

# Initialize a state graph using the defined BmiState structure
graph = StateGraph(BmiState)

# Add nodes (functions) to the graph
graph.add_node("calc_bmi", calc_bmi)
graph.add_node("level_bmi", level_bmi)

# Define the execution flow between nodes
graph.add_edge(START, "calc_bmi")       # Start → Calculate BMI
graph.add_edge("calc_bmi", "level_bmi") # Calculate BMI → Determine Category
graph.add_edge("level_bmi", END)        # Determine Category → End

# Compile the workflow graph into an executable workflow
# Compile karne ke baad ye ek executable workflow object ban jata hai
workflow = graph.compile()

# Run the workflow with input data
result = workflow.invoke({"weight": 40, "height": 1.4})

# Display the final result
print(result)

{'weight': 40, 'height': 1.4, 'bmi': 20.41, 'category': 'Normal'}
```

3. Simple Chatbot

```
# ☑ Import necessary libraries
from langgraph.graph import StateGraph, START, END  # For building the workflow
graph
from typing import TypedDict  # To define the structure of the state dictionary
from openai import AzureOpenAI  # Azure OpenAI client for API interaction
from dotenv import load_dotenv  # To load environment variables from a .env file
import os  # For accessing environment variables

# ☑ Load environment variables from the .env file
# This securely loads Azure OpenAI credentials from your .env file
load_dotenv()

# ☑ Retrieve Azure OpenAI credentials from environment variables
azure_endpoint = os.getenv("AZURE_OPENAI_ENDPOINT")
azure_api_key = os.getenv("AZURE_OPENAI_API_KEY")
```

```
azure_api_version = os.getenv("AZURE_OPENAI_API_VERSION")
azure_deployment = os.getenv("AZURE_OPENAI_DEPLOYMENT_NAME")

# ☑ Initialize the Azure OpenAI client
# The client object is used to communicate with the Azure OpenAI service
client = AzureOpenAI(
    azure_endpoint=azure_endpoint,
    api_key=azure_api_key,
    api_version=azure_api_version
)

# ☑ Define the structure of the state that flows through the workflow
class ChatbotState(TypedDict):
    question: str  # The user's input question
    answer: str    # The model's generated answer

# ☑ Define the function (node) that will handle chat responses
def Chat_resp(state: ChatbotState) -> ChatbotState:
    """
    This function takes the current workflow state as input,
    uses Azure OpenAI to generate an answer based on the user's question,
    and returns the updated state containing the answer.
    """
    # Extract user question from the workflow state
    question = state['question']

    # Create a prompt to guide the model response
    prompt = f"Please provide a short and clear answer related to: {question}"

    try:
        # Send the prompt to Azure OpenAI and get the model response
        response = client.chat.completions.create(
            model=azure_deployment,  # The Azure model deployment name (from your
Azure resource)
            messages=[
                {"role": "system", "content": "You are a helpful AI assistant."},
                {"role": "user", "content": prompt}
            ],
            temperature=0.7,  # Controls creativity (higher = more creative)
            max_tokens=150    # Limits the length of the model's response
        )

        # Extract the model-generated text response
        answer = response.choices[0].message.content

        # Store the generated answer back in the state
        state['answer'] = answer

    except Exception as e:
        # Handle any errors during the API request
        state['answer'] = f"Error: {str(e)}"

    # Return the updated state to continue the workflow
    return state
```

```
# ☑ Create a new workflow graph for the Chatbot
graph = StateGraph(ChatbotState)

# Add the response-generating node to the graph
graph.add_node("Chat_resp", Chat_resp)

# Define the workflow execution order (edges)
# The workflow starts -> runs Chat_resp -> ends
graph.add_edge(START, "Chat_resp")
graph.add_edge("Chat_resp", END)

# ☑ Compile the graph into an executable workflow
workflow = graph.compile()

# ☑ Run the workflow with a user question as input
result = workflow.invoke({"question": "Tell me about Inosuke?"})

# ☑ Print the model's response
print(result)

{'question': 'Tell me about Inosuke?',
 'answer': 'Inosuke Hashibira is a prominent character from *Demon Slayer: Kimetsu no Yaiba*. He is a brash and aggressive Demon Slayer known for wearing a boar mask and dual-wielding serrated swords. Raised in the wild, he has a feral personality, exceptional physical abilities, and uses the Beast Breathing combat style. Despite his rough demeanor, he shows moments of loyalty and growth throughout the series.'}
```

4. Prompt Chaining

```
from langgraph.graph import StateGraph,START,END
from typing import TypedDict
from openai import AzureOpenAI
from dotenv import load_dotenv
import os
load_dotenv()

# ☑ Step 3: Fetch environment variables from the system
azure_endpoint = os.getenv("AZURE_OPENAI_ENDPOINT")
azure_api_key = os.getenv("AZURE_OPENAI_API_KEY")
azure_api_version = os.getenv("AZURE_OPENAI_API_VERSION")
azure_deployment = os.getenv("AZURE_OPENAI_DEPLOYMENT_NAME")

client = AzureOpenAI(
    azure_endpoint=azure_endpoint,
    api_key=azure_api_key,
    api_version=azure_api_version
)

class BlogState(TypedDict):
```

```
    title:str
    outline:str
    content:str

# ☑ Function 1: Create an outline for the blog topic
def create_outline(state: BlogState) -> BlogState:
    """
    Generates a blog outline based on the given title using Azure OpenAI.
    Updates the state with the generated outline.
    """
    title = state['title']
    prompt = f"Generate a detailed outline for a blog post on the topic:
'{title}'."

    try:
        # Azure OpenAI chat completion request
        response = client.chat.completions.create(
            model=azure_deployment,  # Azure deployment name from your environment
            messages=[
                {"role": "system", "content": "You are a skilled blog writer and
content strategist."},
                {"role": "user", "content": prompt}
            ],
            temperature=0.7,  # Controls creativity
            max_tokens=400    # Increased for more detailed outlines
        )

        # Extract model's response text
        outline = response.choices[0].message.content.strip()
        state['outline'] = outline

    except Exception as e:
        # Store error message in the state if API call fails
        state['outline'] = f"Error while generating outline: {str(e)}"

    return state


# ☑ Function 2: Create the actual blog content from the outline
def create_blog(state: BlogState) -> BlogState:
    """
    Generates a 300-word blog post based on the previously created outline.
    Updates the state with the generated content.
    """
    outline = state.get('outline', '')

    # If outline is empty or contains an error, handle gracefully
    if not outline or outline.startswith("Error"):
        state['content'] = "Cannot generate blog content because outline is
missing or invalid."
        return state

    prompt = f"Write a short, engaging blog post of about 300 words based on this
outline:\n\n{outline}"
```

```
    try:
        # Azure OpenAI chat completion request
        response = client.chat.completions.create(
            model=azure_deployment,  # Azure deployment name
            messages=[
                {"role": "system", "content": "You are a creative content writer
who writes in a clear and engaging tone."},
                {"role": "user", "content": prompt}
            ],
            temperature=0.8,  # Slightly more creative for blog writing
            max_tokens=600    # Allow enough tokens for ~300 words
        )

        # Extract and clean up the response text
        content = response.choices[0].message.content.strip()
        state['content'] = content

    except Exception as e:
        # Store any error message in the state
        state['content'] = f"Error while generating content: {str(e)}"

    return state

graph=StateGraph(BlogState)

graph.add_node("create_outline",create_outline)
graph.add_node("create_blog",create_blog)

graph.add_edge(START,"create_outline")

# prompt chaining
graph.add_edge("create_outline","create_blog")

graph.add_edge("create_blog",END)

workflow=graph.compile()

workflow.invoke({"title":"mera bharat mahan"})
{'title': 'mera bharat mahan',
 'outline': '**Outline for Blog Post: "Mera Bharat Mahan"**\n\n---\n\n###
**Introduction**\n1. **Opening Statement**: A heartfelt introduction highlighting
the pride and emotional connection Indians feel toward their motherland.\n   -
Example: "India is not just a country; it\'s a civilization, a culture, and a
symbol of unity amidst diversity."\n2. **Purpose of the Blog**:\n   - To celebrate
India\'s greatness, explore its achievements, and reflect on why it holds an
esteemed position in the world.\n3. **Hook**: A famous quote or saying that
captures the essence of India, like "Unity in Diversity" or "Sare Jahan Se Achha
Hindustan Hamara,
 'content': '**Mera Bharat Mahan: Celebrating the Soul of India**\n\nIndia is not
just a country; it’s a living, breathing civilization—a vibrant tapestry of
cultures, languages, and traditions woven together with the golden thread of
unity. Known as the land of "Unity in Diversity," India holds a unique place in
the world, inspiring awe with its unparalleled richness and resilience. Today,
```

```
let’s take a moment to reflect on why every Indian proudly proclaims, *Mera Bharat Mahan*.\n\n### **I. Cultural Richness Beyond Compare**\n\nFrom the snow-capped Himalayas to the sun-soaked beaches of Kerala, India’s diversity is its greatest strength. It is home to an astounding array of religions, festivals, and traditions, all coexisting harmoniously. Each corner of the country has its own story—be it the vibrant colors of Holi in the North or the Onam festivities in the South, every celebration reflects the nation\'s inclusive spirit. \n\nLanguages? India boasts 22 officially recognized ones, including Hindi, Bengali, Tamil, and Telugu, and thousands of dialects that bring nuance to its identity. And who can forget India’s classical music, mesmerizing dance forms like Bharatanatyam and Kathak, or intricate art like Madhubani and Warli? These treasures make India a cultural powerhouse.
```

Parallel worflow

- Parallel-workflows-in-langgraph

5. Simple parallel workflow

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

class CricState(TypedDict):
    runs: int
    balls: int
    fours: int
    sixes: int

    sr: float                    # Strike rate
    bpb: float                   # Balls per boundary
    boundary_percentage: float   # % of runs scored from boundaries
    summary: str                 # Final summary text

# Define function node to calculate strike rate
def calc_sr(state: CricState):
    # Strike rate = (runs / balls) * 100
    sr = (state['runs'] / state['balls']) * 100
    state['sr'] = sr
    return state
# Define function node to calculate balls per boundary (BPB)
def calc_bpb(state: CricState):
    # It divides balls by (fours + sixes) — though mathematically the current code
    # uses wrong precedence. (But we are not fixing as per user request.)
    bpb = (state['balls'] / state['fours'] + state['sixes'])
    state['bpb'] = bpb
    return state
# Define function node to calculate boundary percentage
def calc_bp(state: CricState):
    # Boundary percentage = (runs from boundaries / total runs) * 100
    boundary_percentage = (((state['fours'] * 4) + (state['sixes'] * 6)) /
```

```
state['runs']) * 100
    state['boundary_percentage'] = boundary_percentage
    return state

# Define summary node to produce the final description
def summary(state: CricState) -> str:
    # Returns a formatted string summarizing the batting performance
    return (
        f"The batsman scored runs at a strike rate of {state['sr']}, "
        f"his balls per boundary (BPB) is {state['bpb']}, "
        f"and his boundary percentage is {state['boundary_percentage']}%."
    )

# Create the LangGraph workflow
graph = StateGraph(CricState)

# Add all computation nodes to the graph
graph.add_node("calc_sr", calc_sr)
graph.add_node("calc_bpb", calc_bpb)
graph.add_node("calc_bp", calc_bp)
graph.add_node("summary", summary)

# --- Define edges (workflow sequence) ---

# Each of the following edges connects START directly to the three calculation
nodes.
# This means all three nodes will execute in parallel as soon as the workflow
starts.
graph.add_edge(START, "calc_sr")
graph.add_edge(START, "calc_bpb")
graph.add_edge(START, "calc_bp")

# After calculations, all three nodes connect to the summary node
graph.add_edge("calc_sr", "summary")
graph.add_edge("calc_bpb", "summary")
graph.add_edge("calc_bp", "summary")

# The summary node then connects to END to mark workflow completion
graph.add_edge("summary", END)

# Compile the graph into an executable workflow
workflow = graph.compile()

# Invoke the workflow with initial data
workflow.invoke({"runs": 100, "balls": 60, "fours": 10, "sixes": 2})

# ------------------------------------------------------------
# ⚠ EXPLANATION OF THE ERROR:
# You will get the following runtime error:
# InvalidUpdateError: At key 'runs': Can receive only one value per step.
#
# WHY THIS HAPPENS:
# In LangGraph, multiple nodes ('calc_sr', 'calc_bpb', 'calc_bp') are all
connected
```

```
# directly to the START node, meaning they execute *in parallel*.
# Each node reads and writes to the same shared state (the CricState dict),
# causing multiple concurrent updates to the same keys ('runs', 'balls', etc.).
#
# LangGraph enforces single-writer semantics: only ONE node may update a given key
# in a single step. Hence, this conflict triggers the InvalidUpdateError.
#
# HOW TO FIX
# ---------------------------------------------------------------
# ------------------------  SOLUTIONS  ------------------------------------

** Returning only the updated key-value pair (instead of the entire state) **

def calc_sr(state:CricState):
    sr=(state['runs']/state['balls'])*100

    return {"sr": sr}

def calc_bpb(state:CricState):
    bpb=(state['balls']/state['fours']+state['sixes'])

    return {"bpb": bpb}

def calc_bp(state:CricState):
    boundary_percentage=(((state['fours']*4)+
(state['sixes']*6))/state['runs'])*100

    return {"boundary_percentage": boundary_percentage}

def summary(state: CricState) -> str:
    text = (
        f"The batsman scored runs at a strike rate of {state['sr']:.2f}, "
        f"his balls per boundary (BPB) is {state['bpb']:.2f}, "
        f"and his boundary percentage is {state['boundary_percentage']:.2f}%."
    )
    return {"summary": text}

OUTPUT:-------------------------

{'runs': 100,
 'balls': 60,
 'fours': 10,
 'sixes': 2,
 'sr': 166.66666666666669,
 'bpb': 8.0,
 'boundary_percentage': 52.0,
 'summary': 'The batsman scored runs at a strike rate of 166.67, his balls per
boundary (BPB) is 8.00, and his boundary percentage is 52.00%.'}
```

```python
from IPython.display import Image

# Call the method to get the PNG bytes
Image(workflow.get_graph().draw_mermaid_png())
```

✓ 1.2s