Contents

[LANGGRAPH 2](#_Toc205914205)

[WHY LANGGRAPH IS NEEDED 2](#_Toc205914206)

[CORE COMPONENTS OF LANGGRAPH 2](#_Toc205914207)

[KEY FEATURES OF LANGGRAPH 2](#_Toc205914208)

[SIMPLE WORKFLOW 3](#_Toc205914209)

[KEY CONCEPTS IN THE PROGRAM 4](#_Toc205914210)

[DEFINING THE ENTRYPOINT 4](#_Toc205914211)

[VALIDATION ON STATE VARIABLE 5](#_Toc205914212)

[ASYNC INVOCATION 7](#_Toc205914213)

[EXAMPLE 7](#_Toc205914214)

[STREAMING 9](#_Toc205914215)

[CONDITIONAL ROUTING 9](#_Toc205914216)

[REDUCER 9](#_Toc205914217)

[TOOL CALLING 9](#_Toc205914218)

# LANGGRAPH

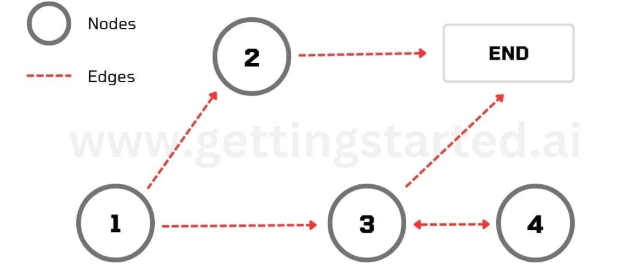
* **LangGraph** is a framework built on top of **LangChain**.
* It enables the creation of **graph-based workflows** for orchestrating complex AI agent behaviors.
* Designed to overcome limitations of **linear agentic workflows** in LangChain.

## WHY LANGGRAPH IS NEEDED

* Traditional LangChain agents operate in a **loop**:
  + Use LLMs to reason and decide which tool to use.
  + Perform tasks step-by-step in a linear fashion.

As workflows grow in complexity, **linear execution becomes inefficient**.LangGraph introduces **non-linear, flexible execution paths** using graph structures.

## CORE COMPONENTS OF LANGGRAPH

****

Nodes

* Represent **individual tasks** or **agents**.
* Can range from simple logic to complex, self-contained agents.
* Examples:
  + **LLM Node**: Uses an LLM to make decisions.
  + **RAG Node**: Handles Retrieval-Augmented Generation.
  + **Tools Node**: Executes tool calls based on LLM decisions.
  + **API Calls Node**: Makes external API requests.
  + **DB Update Node**: Performs database operations.

Edges

* Define **transitions** between nodes.
* Control the **flow of execution** in the graph.

State

* Maintains the **current context** of the workflow.
* Gets **updated dynamically** as nodes execute.

## KEY FEATURES OF LANGGRAPH

State Sharing & Memory

* Nodes can share and update a common state.
* Enables persistent memory across workflow steps.

Conditional Routing

* Allows **dynamic decision-making** on which node to execute next.
* Based on current state or LLM outputs.

Tool Calling

* LangGraph handles **tool invocation** automatically when LLMs decide to use them.

Human-in-the-Loop

* Easily insert human review or input at any point in the workflow.
* Useful for **critical decisions** or **manual overrides**.

Persistent Memory

* Simple to configure.

Supports **long-term context retention** across sessions.

Asynchronous Processing & Streaming

* Supports **streaming outputs** as they become available.
* No need to wait for the entire workflow to complete.

# SIMPLE WORKFLOW

|  |  |
| --- | --- |
| **from typing import TypedDict**  **from langgraph.graph import END, START, StateGraph**  **#STEP 1**  **class HelloWorldState(TypedDict):**  **message: str**  **#STEP 2**  **def hello(state: HelloWorldState):**  **print(f"Hello Node: {state['message']}")**  **return {"message": "Hello " + state["message"]}**  **def bye(state: HelloWorldState):**  **print(f"Bye Node: {state['message']}")**  **return {"message": "Bye " + state["message"]}**  **#STEP 3**  **graph = StateGraph(HelloWorldState)**  **graph.add\_node("hello", hello)**  **graph.add\_node("bye", bye)**  **#STEP 4**  **graph.add\_edge(START, "hello")**  **graph.add\_edge("hello", "bye")**  **graph.add\_edge("bye", END)**  **#STEP 5**  **runnable = graph.compile()**  **output = runnable.invoke({"message": "World"})**  **print(f"Final Output: {output['message']}")** | STEPS   * Step 1: Define State(Shared Between Nodes) * Step 2: Add node to the graph * Step3: Connect Edges to the nodes * Step 4: Execute the workflow |

* This program demonstrates a simple LangGraph workflow using two nodes: hello and bye.
* It shows how LangGraph can be used to define a stateful, directed graph of operations.

## KEY CONCEPTS IN THE PROGRAM

State Definition

|  |  |
| --- | --- |
| class HelloWorldState(TypedDict):      message: str | * Defines the shared state across nodes. * In this case, the state contains a single field: message. |

Node Functions

|  |  |
| --- | --- |
| def hello(state: HelloWorldState):      ...  def bye(state: HelloWorldState):      ... | * Each function represents a node in the graph. * They receive the current state and return an updated state. |

Graph Construction

|  |  |
| --- | --- |
| graph = StateGraph(HelloWorldState)  graph.add\_node("hello", hello)  graph.add\_node("bye", bye) | * Initializes the graph with the defined state type. * Adds two nodes: hello and bye. |

Edge Definitions

|  |  |
| --- | --- |
| graph.add\_edge(START, "hello")  graph.add\_edge("hello", "bye")  graph.add\_edge("bye", END) | * Defines the flow of execution:   **Start → Hello → Bye → End** |

Execution

|  |  |
| --- | --- |
| runnable = graph.compile()  output = runnable.invoke({"message": "World"}) | * Compiles the graph into a runnable workflow. * Invokes the workflow with initial state {"message": "World"}. |

Final Output

The message is transformed twice:

* **First by hello: "Hello World"**
* **Then by bye: "Bye Hello World"**

### DEFINING THE ENTRYPOINT

In **LangGraph**, the **entry and exit points** are special markers that define where the workflow begins and ends. These are essential for structuring the flow of execution in a graph-based agentic system.

#### ENTRY POINT: START

* It marks the **beginning** of the workflow.
* We connect START to the first node in thegraph using **add\_edge**.
* **Example**:
  + **graph.add\_edge(START, "hello") 🡪**This means the graph execution will begin at the hello node.

|  |
| --- |
| * Alternatively, we can set the entry point explicitly by specifying the node   **graph.set\_entry\_point("hello")** |

#### EXIT POINT: END

* **It** marks the **termination** of the workflow.
* **We** connect the final node to END using add\_edge.

**Example**:

* **graph.add\_edge("bye", END) 🡪** This means the graph execution will stop after the bye node.

### VALIDATION ON STATE VARIABLE

* In **LangGraph**, **state validation** refers to the process of ensuring that the **state object** passed between nodes adheres to a defined structure and contains valid data.

|  |  |
| --- | --- |
| **TypedDict for State Definition**  LangGraph uses Python’s TypedDict to define the expected structure of the state.  **from typing import TypedDict**  **class HelloWorldState(TypedDict):**  **message: str**   * This acts as a **contract** for the state. * Every node must return a dictionary that matches this structure. | **2. Validation During Execution**   * When the graph is compiled and invoked: * LangGraph checks that the state returned by each node **matches the defined type**. * If a node returns a state that violates the schema (e.g., missing message or wrong type), it will raise an error. |
| **Example**  If a node returns:  **return {"msg": "Hello World"} # Incorrect key**   * LangGraph will raise an error because msg is not a valid key in HelloWorldState. | Expected:message is a state variable    If invoked without atleast one state variable returns:    Cause  **TypeError: Pregel.invoke() missing 1 required positional argument: 'input'** |

#### VALIDATION USING PYDANTIC SCHEMA

* Using **Pydantic** for state management in **LangGraph** provides a more robust and expressive way to define and validate the state.
* Pydantic models offer built-in type checking, validation, and serialization, which are especially useful in

Why Use Pydantic Instead of TypedDict?

* **TypedDict** is static and only checks types at runtime.
* **Pydantic** adds:
  + Field validation (e.g., length, format, ranges).
  + Default values.
  + Custom validators.
  + Better error messages.
  + Serialization and parsing from various formats (e.g., JSON).

|  |
| --- |
| from pydantic import BaseModel, Field  from langgraph.graph import END, START, StateGraph  class HelloWorldState(BaseModel):      message: str = Field(min\_length=3, max\_length=50)  def hello(state: HelloWorldState):      print(f"Hello Node: {state.message}")      return {"message": "Hello " + state.message}  def bye(state: HelloWorldState):      print(f"Bye Node: {state.message}")      return {"message": "Bye " + state.message}  graph = StateGraph(HelloWorldState)  graph.add\_node("hello", hello)  graph.add\_node("bye", bye)  graph.add\_edge(START, "hello")  graph.add\_edge("hello", "bye")  graph.add\_edge("bye", END)  runnable = graph.compile()  output = runnable.invoke({"message": "World"})  print(f"Final Output: {output['message']}") |

|  |  |
| --- | --- |
| Step1: Define the State Model with Pydantic   * HelloWorldState is a **Pydantic model** * It has one field: message, which must be:   + A string (str)   + At least **3 characters** long   + At most **50 characters** long * Pydantic will **automatically validate** this whenever an instance is created or updated. | from pydantic import BaseModel, Field  class HelloWorldState(BaseModel):  message: str = Field(min\_length=3, max\_length=50) |
| Step 2: Define Node Functions   * Each function receives a HelloWorldState object. * They access state.message, modify it, and return a **new dictionary** with the updated message. * LangGraph will **re-validate** this returned dictionary against the HelloWorldState model. | def hello(state: HelloWorldState):  print(f"Hello Node: {state.message}")  return {"message": "Hello " + state.message}  def bye(state: HelloWorldState):  print(f"Bye Node: {state.message}")  return {"message": "Bye " + state.message} |
| Step 3: Build the Graph   * Create a StateGraph that uses HelloWorldState as its schema. * Nodes hello and bye are added. * Edges define the flow: START → hello → bye → END. | from langgraph.graph import END, START, StateGraph  graph = StateGraph(HelloWorldState)  graph.add\_node("hello", hello)  graph.add\_node("bye", bye)  graph.add\_edge(START, "hello")  graph.add\_edge("hello", "bye")  graph.add\_edge("bye", END) |
| Step 4: Compile and Run the Graph   * runnable.invoke(...) starts the graph with initial state {"message": "World"}. * Pydantic validates this input:   + Is message a string?   + Is it between 3 and 50 characters?   + If validation passes: * hello node runs → returns "Hello World" * bye node runs → returns "Bye Hello World" * Final output is printed. | runnable = graph.compile()  output = runnable.invoke({"message": "World"})  print(f"Final Output: {output['message']}") |
| What If Validation Fails?  If we run:  runnable.invoke({"message": "Hi"})  Pydantic will raise a ValidationError because "Hi" is only 2 characters long, violating min\_length=3. | |

Benefits

* **Validation**: Ensures message is a non-empty string.
* **Error Handling**: Raises clear exceptions if input is invalid.
* **Extensibility**: Easily add more fields and validation rules.
* **Cleaner Code**: Encapsulates state logic in a structured model.

# ASYNC INVOCATION

* Async invocation refers to executing a LangGraph workflow asynchronously using Python's async and await syntax. This is especially useful when your nodes perform I/O-bound operations like:
  + Calling APIs
  + Accessing databases
  + Reading/writing files
  + Waiting for user input or external events

Key Concept

* Instead of blocking the program while waiting for a task to complete, async invocation allows other tasks to run concurrently.
* This improves efficiency and responsiveness, especially in workflows with multiple slow or parallel operations.

How It Works

* LangGraph provides an asynchronous method called: **await runnable.ainvoke(input\_state)**
* Note: This is the async counterpart to: **runnable.invoke(input\_state)**

When to Use Async Invocation

Use ainvoke() when:

* The node functions are defined with **async def**
* You use await inside those functions (e.g., await asyncio.sleep(), await httpx.get())
* You want to run the graph inside an async context (like a FastAPI route or an event loop)

## EXAMPLE

|  |
| --- |
| from typing import TypedDict  from langgraph.graph import END, START, StateGraph  import asyncio  class HelloWorldState(TypedDict):      message: str  async def hello(state: HelloWorldState):      print(f"Hello Node: {state['message']}")      # TODO: Simulate Async Processing      await asyncio.sleep(1)  # Simulating async processing      return {"message": "Hello " + state["message"]}  async def bye(state: HelloWorldState):      print(f"Bye Node: {state['message']}")      # TODO: Simulate Async Processing      await asyncio.sleep(3)  # Simulating async processing      return {"message": "Bye " + state["message"]}  graph = StateGraph(HelloWorldState)  graph.add\_node("hello", hello)  graph.add\_node("bye", bye)  graph.add\_edge(START, "hello")  graph.add\_edge("hello", "bye")  graph.add\_edge("bye", END)  runnable = graph.compile()  # TODO: Async Invocation  async def main():      output = await runnable.ainvoke({"message": "World"})      print(f"Final Output: {output['message']}")  asyncio.run(main()) |

Goal

* Building a **LangGraph workflow** where each node performs **asynchronous operations**
* Simulated using asyncio.sleep() to mimic I/O-bound tasks like API calls, database queries, or long computations.

Step-by-Step Breakdown

|  |  |
| --- | --- |
| Define Asynchronous Node Functions   * These are **async functions**, meaning they return coroutines. * await asyncio.sleep(n) simulates a delay, mimicking real-world async tasks. * Each function receives the current state, processes it, and returns an updated state. | **async def hello(state: HelloWorldState):**  **print(f"Hello Node: {state['message']}")**  **await asyncio.sleep(1)**  **return {"message": "Hello " + state["message"]}**  **async def bye(state: HelloWorldState):**  **print(f"Bye Node: {state['message']}")**  **await asyncio.sleep(3)**  **return {"message": "Bye " + state["message"]}** |
| **.**Asynchronous Invocation   * ainvoke() is the **async version** of invoke(). * It awaits the completion of the entire graph execution. * Each node is awaited in sequence, respecting the graph flow. | async def main():  output = await runnable.ainvoke({"message": "World"})  print(f"Final Output: {output['message']}") |
| Run the Async Main Function   * This starts the event loop and runs the main() coroutine. * The graph executes:   + hello waits 1 second.   + bye waits 3 seconds.   Final output: "Bye Hello World | **asyncio.run(main())** |
| Execution Timeline  Time 0s: START → hello  Time 1s: hello completes → bye  Time 4s: bye completes → END  Total time: ~4 seconds due to sequential async waits. |  |

# STREAMING

* Streaming in LangGraph refers to the ability to receive partial outputs from nodes as they are processed, rather than waiting for the entire graph to finish execution. This is especially useful for:
  + Real-time user interfaces
  + Long-running tasks
  + Incremental updates (e.g., token-by-token LLM generation)

Benefits

* Improves responsiveness in applications.
* Enables human-in-the-loop interactions.
* Supports progressive rendering of results.
* Useful in chatbots, dashboards, and agentic workflows.

How Streaming Works in LangGraph

|  |
| --- |
| LangGraph provides a method called:  **async for event in runnable.astream(input\_state):**  **...**  This allows us to listen to events emitted by each node as the graph executes.  **The data we get in each Streamed Event**   1. The node name that just executed 2. The updated state 3. Optionally, metadata or intermediate results |

## STREAMING MODES IN LANGGRAPH

LangGraph supports multiple **streaming modes** that allow developers to monitor and interact with graph execution in real time.

## EXAMPLE- STREAMING MODES

|  |  |
| --- | --- |
| from typing import TypedDict  from langgraph.graph import END, START, StateGraph  from langgraph.types import StreamWriter  class HelloWorldState(TypedDict):      message: str  def hello(state: HelloWorldState, writer: StreamWriter):      # TODO: Write Custom Keys      return {"message": "Hello " + state['message']}  def bye(state: HelloWorldState):      return {"message": "Bye " + state['message']}  # Define the async graph  graph = StateGraph(HelloWorldState)  graph.add\_node("hello", hello)  graph.add\_node("bye", bye)  graph.add\_edge(START, "hello")  graph.add\_edge("hello", "bye")  graph.add\_edge("bye", END)  runnable = graph.compile()  # TODO: Stream  for chunks in runnable.stream({"message": "World"}, **stream\_mode=""):**      for chunk in chunks:          print(f"Chunk: {chunk['message']}") | The value of stream\_mode can be   * **values** * **updates** * **messages** * **custom** * **debug** |

|  |  |  |
| --- | --- | --- |
| STREAMING MODE | CODE | CONCEPT |
| values | for chunks in runnable.stream({"message": "World"}, stream\_mode="**values**"):      print(chunks)  **OUTPUT**  {'message': 'World'}  {'message': 'Hello World'}  {'message': 'Bye Hello World'} | * It will emit all the values of the of state in after each step(node) * **Use case**: When we want full visibility into the graph’s state at every step. * **Example**: Useful for debugging or logging complete snapshots of the workflow |
| updates | for chunks in runnable.stream({"message": "World"}, stream\_mode="**updates**"):      print(chunks)  **OUTPUT**  **{'hello': {'message': 'Hello World'}}**  **{'bye': {'message': 'Bye Hello World'}}** | * Emits only the node name & states value after each whose values are changed / updated |
| messages |  |  |
| custom | ...  from langgraph.types import StreamWriter  ....  def hello(state: HelloWorldState, writer: StreamWriter):  # TODO: Write Custom Keys  writer({"custom\_key": "custom\_value"})  return {"message": "Hello " + state["message"]}  ......  # TODO: Stream  for chunks in runnable.stream({"message": "World"}, stream\_mode="**custom**"):  print(chunks)  **OUTPUT**  {'custom\_key': 'custom\_value'} | Details Below |
| debug | for chunks in runnable.stream({"message": "World"}, stream\_mode="**debug**"):      print(chunks) | * **What it streams**: Detailed traces and internal execution metadata. |

CUSTOM MODE – IN DETAIL

* The custom streaming mode gives full control over what data is emitted during graph execution. It’s designed for scenarios where the built-in modes (values, updates, messages, debug) don’t fit our needs.
* It uses a **StreamWriter** object that we can call inside the node functions to emit **arbitrary data**.
* This data can be anything: progress updates, intermediate results, tool outputs, or even structured logs.

Why Use It?

* **Fine-grained control**: We can decide what gets streamed and when.
* **Performance**: Emit only what’s necessary, reducing overhead.
* **UX customization**: Tailor the streaming experience for frontend or monitoring tools.

Example Use Case

Suppose we have a node that fetches data in batches. You can stream progress like this:

|  |  |
| --- | --- |
| from langgraph.graph import StateGraph  from langgraph.streaming import StreamWriter  def fetch\_data\_node(state, stream: StreamWriter):  for i in range(10):  # Simulate fetching  data = f"Fetched batch {i+1}"  stream.write({"progress": data})  return state | This will emit:  {"progress": "Fetched batch 1"}  {"progress": "Fetched batch 2"} |

How to Enable It

|  |  |
| --- | --- |
| SYNCHRONOUSLY | ASYNCHRONOUSLY |
| graph.stream(mode="custom") | async for event in graph.astream(mode="custom"):  print(event) |

### STREAMING APIS

LangGraph supports both **synchronous** and **asynchronous** streaming:

* **stream()** – sync iterator for chunked outputs.
* **astream()** – async iterator for non-blocking workflows.
* **astream\_events()** – async API for custom event streaming (mainly for LCEL, but usable with LangGraph too).

## EXAMPLE

# CONDITIONAL ROUTING

* Conditional routing allows dynamic control over the flow of execution based on the current state of the system.
* Conditional routing lets us define **edges** in the graph that are only followed if a certain condition is met. These are called **conditional edges**, and they are typically used to:
  + Route to different nodes based on tool usage.
  + Handle errors or retries.
  + Implement multi-agent or intent-based workflows.

## EXAMPLES

#### EXAMPLE 1

|  |
| --- |
| from typing import TypedDict  from langgraph.graph import END, START, StateGraph  # Define the structure of the input state (customer support request)  class SupportRequest(TypedDict):      message: str      priority: int  # 1 (high), 2 (medium), 3 (low)  # Function to categorize the support request  def categorize\_request(request: SupportRequest):      # TODO: Implement Conditional Routing      print(f"Received request: {request}")      if request["priority"] == 1:          return "handle\_urgent"      elif request["priority"] == 2:          return "handle\_standard"      else:          print("Low priority request, no action taken.")          return END  # No further action for low priority requests    # Function to process high-priority requests  def handle\_urgent(request: SupportRequest):      print(f"Routing to Urgent Support Team: {request}")      return request  # Function to process standard requests  def handle\_standard(request: SupportRequest):      print(f"Routing to Standard Support Queue: {request}")      return request  # Create the state graph  graph = StateGraph(SupportRequest)  # TODO: Create the graph  graph.add\_node("handle\_urgent", handle\_urgent)  graph.add\_node("handle\_standard", handle\_standard)  graph.add\_conditional\_edges(START, categorize\_request)  graph.add\_edge("handle\_urgent", END)  graph.add\_edge("handle\_standard", END)  runnable = graph.compile()  # Simulate a customer support request  print(runnable.invoke({"message": "My account was hacked! Urgent help needed.", "priority": 1}))  print(runnable.invoke({"message": "I need help with password reset.", "priority": 3})) |

Concept Overview

**Conditional routing** in LangGraph allows you to dynamically decide which node (function) to execute next based on the current state. Instead of a fixed sequence, the graph uses a **routing function** to determine the next step.

Routing Function

|  |  |
| --- | --- |
| def categorize\_request(request: SupportRequest):  if request["priority"] == 1:  return "handle\_urgent"  elif request["priority"] == 2:  return "handle\_standard"  else:  return END | This is the **conditional routing function**. It inspects the priority and returns the name of the next node:   * handle\_urgent for high priority * handle\_standard for medium * END for low priority (no further action) |

Graph Construction

|  |  |
| --- | --- |
| graph = StateGraph(SupportRequest)  graph.add\_node("handle\_urgent", handle\_urgent)  graph.add\_node("handle\_standard", handle\_standard)  graph.add\_conditional\_edges(START, categorize\_request)  graph.add\_edge("handle\_urgent", END)  graph.add\_edge("handle\_standard", END) | * add\_node: Registers the processing functions. * add\_conditional\_edges: Connects the START node to the routing function. * add\_edge: Connects each processing node to END. |

Summary of Flow

START

↓

categorize\_request

├── priority == 1 → handle\_urgent → END

├── priority == 2 → handle\_standard → END

└── priority == 3 → END

#### EXAMPLE 2

|  |
| --- |
| from typing import TypedDict  from langgraph.graph import END, START, StateGraph  # Define the structure of the input state (job application)  class JobApplication(TypedDict):      applicant\_name: str      years\_experience: int  # TODO: Implement the function to categorize candidates based on experience  def categorize\_candidate(application: JobApplication):      years\_of\_exp = application["years\_experience"]      if years\_of\_exp >= 5:          return "schedule\_interview"      return "assign\_skills\_test"  # Function for interview scheduling  def schedule\_interview(application: JobApplication):      print(f"Candidate {application['applicant\_name']} is shortlisted for an interview.")      return {"status": "Interview Scheduled"}  # Function for skills test  def assign\_skills\_test(application: JobApplication):      print(f"Candidate {application['applicant\_name']} is assigned a skills test.")      return {"status": "Skills Test Assigned"}  # Create the state graph  graph = StateGraph(JobApplication)  # TODO: Add nodes to the graph  graph.add\_node("schedule\_interview", schedule\_interview)  graph.add\_node("assign\_skills\_test", assign\_skills\_test)  # TODO: Define edges (workflow steps)  graph.add\_conditional\_edges(START, categorize\_candidate)  graph.add\_edge("schedule\_interview", END)  graph.add\_edge("assign\_skills\_test", END)  # Compile the workflow  runnable = graph.compile()  # Simulate job applications  print(runnable.invoke({"applicant\_name": "Alice", "years\_experience": 6}))  print(runnable.invoke({"applicant\_name": "Bob", "years\_experience": 3})) |

#### EXAMPLE 3

|  |
| --- |
| from typing import TypedDict  import os  from dotenv import load\_dotenv  from langchain\_openai import AzureChatOpenAI  from langgraph.graph import END, START, StateGraph  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  class DecisionMessage(TypedDict):      message: str  def conditional\_action(state: DecisionMessage):      if "evening" in state["message"]:          return "play"      else:          return "study"  def study(state: DecisionMessage):      print(f"Message Recieved {state['message']}")      return {"message": "Time To Play!!"}  def play(state: DecisionMessage):      print(f"Message Recieved {state['message']}")      return {"message": "Time To Study!!"}  graph = StateGraph(DecisionMessage)  graph.add\_node("study", study)  graph.add\_node("play", play)  graph.add\_conditional\_edges(START, conditional\_action)  graph.add\_edge("play", END)  graph.add\_edge("study", END)  runnable = graph.compile()  output = runnable.invoke({"message": "Its evening time"})  print(f"Final Output: {output['message']}") |

# REDUCER

* **reducer** is a special kind of node used to **accumulate or transform state** across multiple steps.
* It takes the current state and the output of a node, and returns a **new state**
* It’s particularly useful when we want to **merge outputs**, **track history**, or **update shared context** as your graph progresses.

How to Use a Reducer in LangGraph

|  |  |
| --- | --- |
| Define State | from typing import TypedDict, List  class ChatState(TypedDict):  history: List[str]  latest\_message: str |
| Define a Reducer Function | def update\_history(state: ChatState, new\_message: str) -> ChatState:  state["history"].append(new\_message)  state["latest\_message"] = new\_message  return state   * ***This function takes the current state and the new message, appends it to history, and updates the latest message.*** |
| Add Reducer to Graph | from langgraph.graph import StateGraph  graph = StateGraph(ChatState)  graph.add\_node("chat", chat\_function)  graph.add\_reducer("chat", update\_history)  **OUTPUT**   * chat\_function returns a new message. * update\_history merges it into the state. |

## MESSAGESTATE

# TOOL CALLING

|  |
| --- |
| The **reasoning capability of a Large Language Model (LLM)** refers to its ability to simulate logical thinking, problem-solving, and inference-making based on the patterns it has learned from vast amounts of data. |

A diagram of a agent

AI-generated content may be incorrect.

* The power of these agents comes from the **reasoning capabilities of the large language models that we use and also the tools we provide to these agents**.
* For example, these tools can be
  + *Search tools to perform search on the internet or in our organization,*
  + *Retrieval tools to perform rag,*
  + *Code execution tools that can execute the code on the fly.*
  + *Querying tools to interact with different types of databases,*
  + *File manipulation tools,*
  + *Custom API tools to invoke apis, and more.*

## TOOL CALLING USING LANGGRAPH

Example

|  |
| --- |
| To set a tool to recommend a Restaurant for a given location |

