Contents

[LANGGRAPH 3](#_Toc207544078)

[WHY LANGGRAPH IS NEEDED 3](#_Toc207544079)

[CORE COMPONENTS OF LANGGRAPH 3](#_Toc207544080)

[KEY FEATURES OF LANGGRAPH 4](#_Toc207544081)

[SIMPLE WORKFLOW 5](#_Toc207544082)

[KEY CONCEPTS IN THE PROGRAM 5](#_Toc207544083)

[DEFINING THE ENTRYPOINT 7](#_Toc207544084)

[CREATING STATE SCHEMA 7](#_Toc207544085)

[CREATING STATE SCHEMA USING DATACLASS 7](#_Toc207544086)

[CODE 7](#_Toc207544087)

[VALIDATION ON STATE VARIABLE 8](#_Toc207544088)

[VALIDATION USING PYDANTIC SCHEMA 9](#_Toc207544089)

[TYPE VALIDATION 9](#_Toc207544090)

[FIELD VALIDATION EXAMPLE 10](#_Toc207544091)

[CHAINS IN LANGGRAPH 11](#_Toc207544092)

[CHAT MESSAGES 11](#_Toc207544093)

[CLASS HIERARCHY 12](#_Toc207544094)

[MESSAGE TYPES 12](#_Toc207544095)

[FUNCTION MESSAGE EXAMPLE 13](#_Toc207544096)

[BINDING TOOLS 14](#_Toc207544097)

[ROUTERS 15](#_Toc207544098)

[WHAT IS A TOOL? 15](#_Toc207544099)

[EXAMPLE: BINDING A CALCULATOR TOOL 15](#_Toc207544100)

[ASYNC INVOCATION 16](#_Toc207544101)

[EXAMPLE 16](#_Toc207544102)

[CONDITIONAL ROUTING 17](#_Toc207544103)

[EXAMPLES 18](#_Toc207544104)

[REDUCER 21](#_Toc207544105)

[WHY REDUCERS? 22](#_Toc207544106)

[IN-BUILT REDUCERS 23](#_Toc207544107)

[add\_message REDUCER 23](#_Toc207544108)

[CUSTOM REDUCER 23](#_Toc207544109)

[MESSAGESTATE 24](#_Toc207544110)

[TOOL CALLING 25](#_Toc207544111)

[FLOW OVERVIEW 25](#_Toc207544112)

[WHAT IS A TOOL? 25](#_Toc207544113)

[TOOL SCHEMA 26](#_Toc207544114)

[TOOL CALLING IN LANGGRAPH 26](#_Toc207544115)

[HOW THE LLM DECIDES WHICH TOOL TO CALL 28](#_Toc207544116)

[TOOL NODE 29](#_Toc207544117)

[CODE EXAMPLE - TOOL CALLING IN LANGGRAPH 32](#_Toc207544118)

[MULTIPLE TOOL INTERGRATION 38](#_Toc207544119)

[ReACT AGENT ARCHITECTURE 42](#_Toc207544120)

[ReACT AGENT ARCHITECTURE IMPLEMENTATION 43](#_Toc207544121)

[MEMORY 46](#_Toc207544122)

[HOW MEMORY WORKS 49](#_Toc207544123)

[SHORT TERM MEMORY 50](#_Toc207544124)

[LONG-TERM MEMORY 55](#_Toc207544125)

[STREAMING 55](#_Toc207544126)

[STREAMING MODES IN LANGGRAPH 56](#_Toc207544127)

[EXAMPLE- STREAMING MODES 56](#_Toc207544128)

[STREAMING APIS 58](#_Toc207544129)

[EXAMPLE 58](#_Toc207544130)

[HUMAN IN THE LOOP(HITL) 58](#_Toc207544131)

[USECASE – HITL 58](#_Toc207544132)

[EXAMPLE 59](#_Toc207544133)

[RAG WITH LANGGRAPH 59](#_Toc207544134)

# 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.
* Its like a **flowchart for your AI app**, where each step (or node) can:
  + Run a language model (like GPT)
  + Call a tool or function
  + Pause for human feedback
  + Remember past interactions

What Can You Build with LangGraph?

LangGraph is perfect for building:

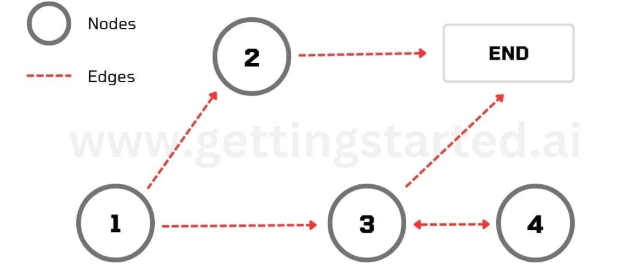
* 🤖 **Agents**: AI that can think, act, and remember
* 👥 **Multi-agent systems**: Multiple bots or tools working together
* 🧠 **Memory-aware apps**: Apps that remember past conversations or decisions
* 🛑 **Human-in-the-loop workflows**: AI that pauses and lets a human review or guide the process

## 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

* Every node in function which returns a dictionary(state).
* The returned key can be the update state values
* The Parameter to the function is the State
* 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.

A screenshot of a computer

AI-generated content may be incorrect.

State

* It serves as the input for the nodes and edges
* State is a dictionary – which helps in tracking the graph like node execution result temporary result to chat history
* It is local to the graph and accessible to all the nodes and edge in the graph.
* This can also be stored in Persistent storage , in scenarios – where we want stop and then later resume the flow execution from the same state where we left
* Maintains the **current context** of the workflow.
* Gets **updated dynamically** as nodes execute.

StateGraph

* This defines the graph’s overall state including the schema -variables and datatypes, shared across nodes during agent execution

State Schema

* The state schema the overall schema of data variables shared across the agent ‘s lifecycle.
* It can be any python data type typically TypeDict or Pydantic base Model

## 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 SCHEMA

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

TYPEDICT

|  |  |
| --- | --- |
| class HelloWorldState(TypedDict):  message: str | * TypedDict is a feature from Python’s typing module that lets us define a **dictionary with a fixed structure** — meaning we know exactly **what keys and value types** it should have. * It’s like saying -“This dictionary must have a key called message, and its value must be a string.” |

Why Use TypedDict?

Using TypedDict helps with:

1. **Type safety**: we get warnings or errors if you use the wrong key or value type. But it is not enforced during runtine
2. **Code clarity**: It’s easier to understand what kind of data your app expects.
3. **Better tooling**: IDEs and linters can give smarter suggestions and catch bugs early.

Example Usage in LangGraph

We often pass around a **state object** between nodes in a graph. TypedDict helps define what that state looks like below

|  |  |
| --- | --- |
| class HelloWorldState(TypedDict):      message: str  Now, any function or node that uses state knows:   * It must contain a "message" key * That key must hold a string() | If we accidentally do this:  output = runnable.invoke({"message": 123})  # ❌ Wrong type!  The IDE or type checker will give warning, but does not give error at runtime |

Hence In LangGraph Context

|  |  |
| --- | --- |
| When building a StateGraph, you define the state type like this:  from langgraph.graph import StateGraph  graph = StateGraph(HelloWorldState) | This tells LangGraph:  “Every node in this graph will receive and return a dictionary that matches the HelloWorldState structure.” |

#### 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. * The first positional argument of the Node(function) is the state * To access the state values: **state[‘state\_variable’]** |

#### 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.

# CREATING STATE SCHEMA

There are multiple ways to create Stateschema

1. Using TypeDict(as in above example)
2. Using DataClasses

## CREATING STATE SCHEMA USING DATACLASS

Using a dataclass makes the state:

* **Structured**: We define exactly what fields exist.
* **Readable**: We can access fields with dot notation (state.graph\_info).
* **Safe**: Type hints help catch bugs early.
* **Extendable**: You can add methods, defaults, and validations.

|  |  |
| --- | --- |
| @dataclass  **class State:**  **graph\_info: str**  **is\_raining: bool** | This defines the **schema of the state** that flows through the graph. Each node receives a State object, modifies it, and returns a new version |
| INVOKING THE GRAPH | The graph will be invoked differently   * **TypeDict**: runnable.invoke({"graph\_info": "", "is\_raining": False}) * **DataClass** : runnable.invoke(State(graph\_info="", is\_raining=False)) |

### CODE

|  |
| --- |
| from typing\_extensions import TypedDict  from dataclasses import dataclass  # Import StateGraph from the appropriate module  from langgraph.graph import END, START, StateGraph  ## State definition  @dataclass  class State:      graph\_info: str      is\_raining: bool  def start\_play(state: State) -> dict:      # Initialize the state      return {"graph\_info": state.graph\_info + " I am planning to play."}  def cricket(state: State) -> dict:      # Update the state for cricket      return {"graph\_info": state.graph\_info + " cricket."}  def badminton(state: State) -> dict:      # Update the state for badminton      return {"graph\_info": state.graph\_info + " badminton."}  def play\_condition(state: State) -> str:      # Condition to decide which sport to play      if state.is\_raining:          return "badminton"      return "cricket"  graph = StateGraph(State)  graph.add\_node("start\_play", start\_play)  graph.add\_node("cricket", cricket)  graph.add\_node("badminton", badminton)  graph.add\_edge(START, "start\_play")  graph.add\_conditional\_edges("start\_play", play\_condition)  graph.add\_edge("cricket", END)  graph.add\_edge("badminton", END)  runnable = graph.compile()  print(runnable.invoke({"graph\_info": "", "is\_raining": False})) |

# VALIDATION ON STATE VARIABLE

* **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.
* When we define a class that inherits from pydantic.BaseModel, Pydantic automatically enables **automatic type validation**. Hence it will
  + Validates the types of fields.
  + Converts compatible types when possible.
  + Raises errors when data is invalid.

### TYPE VALIDATION

|  |
| --- |
| from pydantic import BaseModel  from langgraph.graph import END, START, StateGraph  class State(BaseModel):      name: str  def greet(state: State) -> dict:      greeting = f"Hello, {state.name}!"      return {"name": greeting}  graph = StateGraph(State)  graph.add\_node("greet", greet)  graph.add\_edge(START, "greet")  graph.add\_edge("greet", END)  runnable = graph.compile()  print(runnable.invoke(State(name=”Alice”)))  print(runnable.invoke(State(name=123))) 🡨 ERROR PYDANTIC VALIDATION FAILED |
| **name**  **Input should be a valid string [type=string\_type, input\_value=123, input\_type=int]** |

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).’

### FIELD VALIDATION EXAMPLE

|  |
| --- |
| 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.

# CHAINS IN LANGGRAPH

**CONCEPT TO LEARN**

* How to use chat messages as our graph state?
* How to use chat models in graph nodes?
* How to bind tools to our LLM in chat models?
* How to execute the tools call in our graph nodes ?

## CHAT MESSAGES

* In LangChain, when interacting with a Language Model (LLM), we don’t just send raw text. Instead, we send a **structured list of messages** that helps the model understand:
  + **Who is speaking** (user, assistant, system, tool, etc.)
  + **What role each message plays**
  + **How to use the context** to generate accurate and consistent responses

|  |  |  |
| --- | --- | --- |
| **Type** | **Role** | **Used By** |
| **SystemMessage** | Sets the assistant's behavior/persona | Developer |
| **HumanMessage** | Represents user input | End user |
| **AIMessage** | Represents model output | Language model |
| **ToolMessage** | Returns results from external tools | Tools (e.g., search) |
| **FunctionMessage** | Returns results from function calls | Functions |

### CLASS HIERARCHY

|  |  |
| --- | --- |
| A diagram of text and words  AI-generated content may be incorrect. | BaseMessage  ├── HumanMessage (user input)  ├── AIMessage (LLM output)  ├── SystemMessage (instructions/context)  ├── ToolMessage (results returned from tools)  └── FunctionMessage (specialized outputs) |

### MESSAGE TYPES

#### HUMAN MESSAGE

* It represents a **message authored by the user (human)** in a conversation with a language model.
* It’s a lightweight **data structure**, not the text itself.

Why is it needed?

When we call an LLM, we don’t just send raw text. Instead, we send a **list of structured messages** so the model knows:

* who is speaking,
* what role each message has,
* how to use that context.

|  |
| --- |
| Example:  from langchain\_core.messages import HumanMessage, SystemMessage  messages = [  SystemMessage(content="You are a helpful assistant."),  HumanMessage(content="What are the symptoms of diabetes?")  ] |

How it works?

|  |  |
| --- | --- |
| Internally, HumanMessage inherits from BaseMessage.  It has attributes:   * content: the actual text or structured payload the user typed. * additional\_kwargs: optional dict for things like {"name": "Alice"}. * type: always "human" for this class. | msg = HumanMessage(content="Hello doctor!")  print(msg.type) # "human"  print(msg.content) # "Hello doctor!" |

#### AI MESSAGE

* Represents a message authored by the AI (LLM) in a conversation.
* It’s the counterpart of HumanMessage.

Why we need it?

* Let’s us distinguish “assistant’s reply” from user input and system instructions.
* Crucial in multi-turn conversations, where the model needs to “see its own past outputs” to stay consistent.

|  |
| --- |
| from langchain\_core.messages import AIMessage  msg = AIMessage(content="The main symptoms of diabetes include increased thirst, frequent urination, and fatigue.")  print(msg.type) # "ai"  print(msg.content) # The assistant's reply text  **Usage in a conversation**  conversation = [  HumanMessage(content="What are symptoms of diabetes?"),  AIMessage(content="Some common symptoms include thirst, urination, fatigue...")  ] |

#### SYSTEM MESSAGE

* Represents a system-level instruction that sets context for the AI.
* Think of it as the rules of the game or the persona of the assistant.

Why we need it

* Guides the model’s behavior without being “user text.”
* System messages often set tone, scope, or constraints (e.g., “You are a doctor assistant who always gives polite answers”).

|  |
| --- |
| from langchain\_core.messages import SystemMessage  msg = SystemMessage(content="You are a helpful healthcare assistant. Answer truthfully and cite sources.")  print(msg.type) # "system"  print(msg.content) # "You are a helpful healthcare assistant..."  **#Usage in a conversation**  conversation = [  SystemMessage(content="You are a medical research assistant."),  HumanMessage(content="What are the latest treatments for asthma?"),  AIMessage(content="Latest treatments include...") |

#### EXAMPLE: SIMPLE CONVERSATION

|  |
| --- |
| from langchain\_core.messages import HumanMessage, AIMessage, SystemMessage  # Define the conversation  messages = [  SystemMessage(content="You are a helpful assistant."),  HumanMessage(content="What are the symptoms of diabetes?"),  AIMessage(content="Common symptoms include increased thirst, frequent urination, and fatigue.")  ] |
| * **SystemMessage**: Sets the assistant’s tone and behavior. * **HumanMessage**: User asks a health-related question. * **AIMessage**: Assistant replies with relevant information. |

#### TOOL MESSAGE EXAMPLE

Suppose the assistant uses a **search tool** to answer a question.

|  |
| --- |
| from langchain\_core.messages import ToolMessage  tool\_result = ToolMessage(content="Search results: Diabetes symptoms include thirst, fatigue, and blurred vision.") |

This message is injected into the conversation so the model can use it to generate a final response.

#### FUNCTION MESSAGE EXAMPLE

* Used when the assistant calls a **function** (e.g., to calculate BMI).

|  |
| --- |
| from langchain\_core.messages import FunctionMessage  function\_output = FunctionMessage(content="BMI is 24.5, which is considered normal.") |

* This helps the model incorporate structured outputs from backend logic.

#### SUMMARY

Here’s a full conversation flow:

conversation = [

SystemMessage(content="You are a fitness assistant."),

HumanMessage(content="Calculate my BMI. I weigh 70kg and I'm 1.75m tall."),

FunctionMessage(content="BMI is 22.86, which is considered healthy."),

AIMessage(content="Your BMI is 22.86. That's within the healthy range!")

]

LangChain message types allow for **structured, multi-turn conversations** with clear roles:

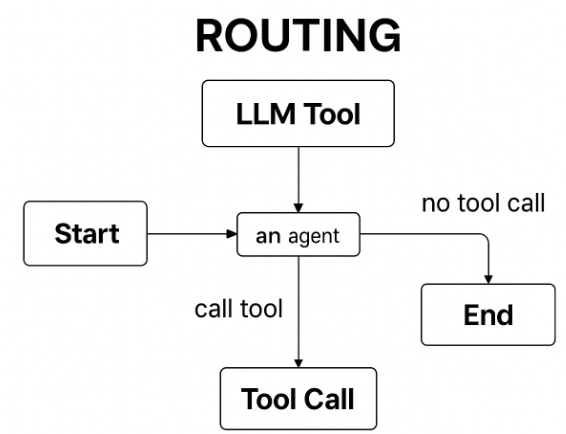
* 🧑 **HumanMessage**: What the user says
* 🤖 **AIMessage**: What the assistant replies
* ⚙️ **SystemMessage**: How the assistant should behave
* 🔍 **ToolMessage**: What external tools return
* 🧮 **FunctionMessage**: What functions compute

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Message Type** | **Description** | **Purpose** | **Attributes** | **Example Usage** |
| **BaseMessage** | Abstract base class for all message types. | Provides a common structure for all messages. | Inherited by all other message types. | Used internally as a superclass. |
| **HumanMessage** | Message authored by the user. | Identifies user input in a structured format. | content, additional\_kwargs, type="human" | HumanMessage(content="Hello doctor!") |
| **AIMessage** | Message authored by the AI (LLM). | Distinguishes assistant replies from other messages. | content, type="ai" | AIMessage(content="Symptoms include thirst...") |
| **SystemMessage** | System-level instruction or context. | Sets tone, rules, or persona for the assistant. | content, type="system" | SystemMessage(content="You are a helpful assistant.") |
| **ToolMessage** | Message containing results returned from tools. | Communicates tool outputs back to the model. | Typically includes tool name and result payload. | Used when tools like search or code execution return data. |
| **FunctionMessage** | Message containing specialized outputs from function calls. | Enables structured function call responses. | Includes function name and output. | Used in advanced workflows involving function calling. |

## BINDING TOOLS

* **Binding a tool** means connecting an external function, agent, or capability (like a LangChain tool or a custom function) to a node in the graph so it can be invoked as part of the workflow.
* This allows us to integrate **tool-augmented reasoning** into the stateful graph logic.

## ROUTERS



A **Router** is a decision-making node in a graph.It allows the **LLM to route** between:

* A **direct response** (no tool call needed). i.e*. Based on user input If no tool is needed → respond directly*.
* A **tool call** (when a function needs to be executed). i.e. *Based on user input: If a tool is needed → call the tool, execute it, and return the result.*
* This is done **automatically** by LLM, making it act like an **agent**. It is called agent because the LLM 🡪 Understands the input 🡪 Decides what tool to use 🡪 Executes the tool 🡪Returns the result 🡪
* All steps are done without human intervention -hence, it behaves like an **agent**

## WHAT IS A TOOL?

* A **tool** is typically a callable function that performs a specific task—like
  + **Searching the web**
  + **Calling an API**
  + **Doing Calculations**
  + **Querying a database.**
* These tools are often defined using LangChain's Tool class or similar wrappers.
* LangGraph allows us to **bind** these tools to nodes in the graph, enabling our workflow to use them dynamically based on the state.

### EXAMPLE: BINDING A CALCULATOR TOOL

|  |  |
| --- | --- |
| 1. Define the Tool  from langchain.tools import tool  **@tool**  **def add\_numbers(a: int, b: int) -> int:**  **return a + b**  This creates a LangChain-compatible tool that adds two numbers. | 2. Define the State  from pydantic import BaseModel  class State(BaseModel):  a: int  b: int  result: int = 0 |
| 3. Bind the Tool to a Node  from langgraph.graph import StateGraph, START, END  def use\_tool(state: State):  result = add\_numbers.invoke({"a": state.a, "b": state.b})  return {"result": result} | 4. Build the Graph  graph = StateGraph(State)  graph.add\_node("calculate", use\_tool)  graph.add\_edge(START, "calculate")  graph.add\_edge("calculate", END)  runnable = graph.compile()  output = runnable.invoke(State(a=5, b=7))  print(output) |

# 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. |  |

# 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.

## WHY REDUCERS?

Let’s understand the issue using below code example

|  |
| --- |
| from typing import Annotated, TypedDict  from langchain\_core.messages import AnyMessage  from operator import add  from langgraph.graph import END, START, StateGraph  from langchain\_core.messages import AIMessage, HumanMessage  # Define chatbot state with accumulated messages  class ChatBotState(TypedDict):      messages: list[AnyMessage]      discount: int  # Responses based on intent level  def connect\_to\_sales(state: ChatBotState):      return {"messages": [AIMessage(content="Great! Let me connect you with our sales team right away. 🚀")],              "discount": 10}  def sales\_response(state: ChatBotState):      return {"messages": [AIMessage(content="We have the best offer for you 🚀")],              "discount": 20}  # Build chatbot conversation flow  graph\_builder = StateGraph(ChatBotState)  # Add nodes  graph\_builder.add\_node("connect\_to\_sales", connect\_to\_sales)  graph\_builder.add\_node("sales\_response", sales\_response)  # Define conversation flow  graph\_builder.add\_edge(START, "connect\_to\_sales")  graph\_builder.add\_edge("connect\_to\_sales", "sales\_response")  graph\_builder.add\_edge("sales\_response", END)  # Compile chatbot  chatbot = graph\_builder.compile()  # Simulate different conversations  test\_inputs = "I want to buy your product."  messages = chatbot.invoke({"messages": [HumanMessage(content=test\_inputs)]})  for message in messages["messages"]:      print(f"🤖 \*\*Bot:\*\* {message.content}")  print("Final Discount: ",messages['discount'],'%') |

The example code has the chatbot state includes:

* messages: a list of chat messages.
* discount: an integer value.

Two nodes:

* connect\_to\_sales: responds with a message and offers a 10% discount.
* sales\_response: responds with another message and offers a 20% discount.

Without Reducers

When the graph is run:

* Only the last message ("We have the best offer for you") is retained.
* Only the last discount (20%) is shown.
* Earlier messages and discounts are overwritten.
* This happens because no reducer functions are defined — LangGraph uses a default reducer that overwrites values.

With Reducers

By using Annotated and the add reducer from Python’s operator module:

|  |  |
| --- | --- |
| from typing import Annotated  from operator import add  **class ChatBotState(TypedDict):**  **messages: Annotated[list[AnyMessage], add]**  **discount: Annotated[int, add]** | LangGraph now:  Appends new messages to the list.  Adds discount values together.  Result After Applying Reducers  All messages are preserved: OUTPUT   * 🤖 \*\*Bot:\*\* I want to buy your product. * 🤖 \*\*Bot:\*\* Great! Let me connect you with our sales team right away. 🚀 * 🤖 \*\*Bot:\*\* We have the best offer for you 🚀 * Final Discount: 30 % |

Key Takeaways

* Reducers control how state is updated across nodes.
* Without reducers: state is overwritten.
* With reducers: state is merged or accumulated.
* You can use built-in reducers like add, or define custom reducers for advanced control.

## IN-BUILT REDUCERS

### add\_message REDUCER

* This is a **built-in reducer** provided by LangGraph that appends new messages to the existing list in the state.
* It’s commonly used to **accumulate chat history**.

## CUSTOM REDUCER

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

1. With MessageState

When you use MessageState, LangGraph manages the chat history for you.

How it works

MessageState comes with a built-in messages: list field.

Every input (human, system, AI, tool messages) is appended automatically.

The checkpointer stores these messages per thread\_id, so when you continue the same thread, history is replayed.

from langgraph.graph import StateGraph, MessagesState

class ConversationState(MessagesState):

pass

graph = StateGraph(ConversationState)

# Each run automatically appends to state.messages

👉 Result: you don’t manually track past messages — LangGraph handles memory accumulation.

🔹 2. Without MessageState

If you don’t use MessageState, you have to define your own state schema.

Example:

from langgraph.graph import StateGraph

class MyState:

# Instead of messages, maybe you only want the last turn

user\_input: str

ai\_response: str

How you’d manage memory:

You must decide what to persist across checkpoints.

If you want a running chat history, you’d need to store messages manually in your custom state.

If you only store the *latest input/response*, you lose long-term conversational context.

👉 Example:

class MyState:

history: list[str] # manually manage memory

Then, whenever the user says something, you’d append it yourself:

def add\_message(state: MyState, msg: str):

state.history.append(msg)

return state

🔹 Comparison

| Feature | With MessageState | Without MessageState |
| --- | --- | --- |
| Chat history | Auto-managed (state.messages) | You must implement manually |
| Integration | Works seamlessly with LangChain-style messages (HumanMessage, AIMessage, etc.) | You decide structure (strings, dicts, etc.) |
| Checkpoint storage | Messages are stored and replayed automatically | Whatever you put in state is checkpointed |
| Use case | Conversational agents, multi-turn dialogue | Custom workflows (tracking slots, variables, metadata) |

✅ In short:

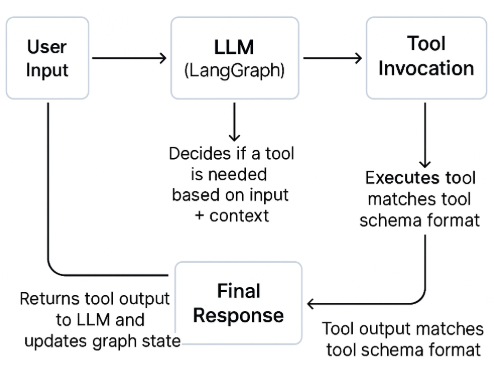
Use MessageState if your memory is message-based (like a chatbot).

Use a custom state if your memory is structured differently (like storing only slots, variables, or domain-specific context).

# TOOL CALLING

* Tool calling is a mechanism that allows a Language Model (LLM) to **interact with external systems—such as APIs, databases, or custom functions**—by invoking predefined tools based on the user's natural language input.

## FLOW OVERVIEW

****

## WHAT IS A TOOL?

|  |  |
| --- | --- |
| **TOOL** | **TOOL SCHEMA** |
| **A tool is a callable function or API that performs a specific task. It could be:**   * A weather API * A calculator * A database query * A custom Python function | **Each tool is defined with a schema that tells the LLM**   * What the tool does * What inputs it expects * What output it returns |
| from langchain\_core.tools import tool  @tool  def multiply(a: int, b: int) -> int:      """Multiply two numbers."""      return a \* b | **This automatically generates a schema like:**  {    "name": "multiply",    "description": "Multiply two numbers.",    "args": {      "type": "object",      "properties": {        "a": {"type": "integer"},        "b": {"type": "integer"}      },      "required": ["a", "b"]    }  } |

## TOOL SCHEMA

A tool schema is a structured definition that helps the LLM understand how to use the tool. It includes:

|  |  |
| --- | --- |
| **NAME (name)** | A unique identifier for the tool. |
| **DESCRIPTION(description)** | A natural language explanation of what the tool does. |
| **ARGUMENTS(args)** | A JSON schema that defines the expected input parameters. This includes:   * type: Usually "object" * properties: A dictionary of parameter names and their types * required: A list of required parameters |

|  |  |
| --- | --- |
| Tool | Tool Schema |
| from langchain\_core.tools import tool  @tool  def multiply(a: int, b: int) -> int:      """Multiply two numbers."""      return a \* b | **This automatically generates a schema like:**  {    "name": "multiply",    "description": "Multiply two numbers.",    "args": {      "type": "object",      "properties": {        "a": {"type": "integer"},        "b": {"type": "integer"}      },      "required": ["a", "b"]    }  } |

## TOOL CALLING IN LANGGRAPH

LangGraph uses this schema to:

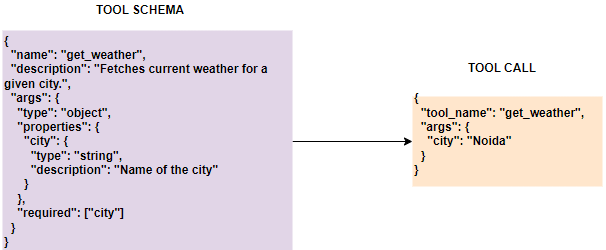
* Let the model decide **when** to call a tool.
* Structure the **tool call** with correct parameters.
* Integrate the **tool output** back into the graph state.

#### Step 1: Let the Model Decide *When* to Call a Tool

* The LLM receives a user query and determines—based on context and intent—whether a tool should be invoked.
* Example**: User Input:“What’s the weather like in Noida today?”**
* LLM Decision: The model recognizes that this query requires real-time data, which it cannot generate internally. It decides to call a tool named **get\_weather.**

#### Step 2: Structure the Tool Call with Correct Parameters

* The schema defines what arguments are needed and their types.
* Then, the LLM uses the tool schema to format the input correctly.



#### Step 3: Integrate the Tool Output Back into the Graph State

|  |  |
| --- | --- |
|  | * Once the tool returns its output, LangGraph updates the graph state with the result. * This allows the LLM to use the output in the next step of reasoning or response generation. |

## HOW THE LLM DECIDES WHICH TOOL TO CALL

1. Tool Binding with Descriptions

* Each tool is **bound to the LLM** with a **name**, **function**, and a **description** (**docstring**).
* The description tells the LLM what the tool does, what inputs it expects, and when it should be used.

2. Natural Language Understanding

* When a user sends a query (e.g., “What is 2 + 2?”), the LLM:
  + Parses the intent.
  + Matches it against the tool descriptions(**docstring**).
  + Determines if any tool is relevant.

3. Internal Routing Logic

* LangChain uses a **router node** or **agent executor** that allows the LLM to:
  + Choose between responding directly or calling a tool.
  + Automatically route the query to the correct tool if needed.

4. Tool Schema Matching

* Tools often have a **schema** (like expected input types).
* The LLM uses this schema to format the tool call correctly (e.g., add(a=2, b=2)).

EXAMPLE

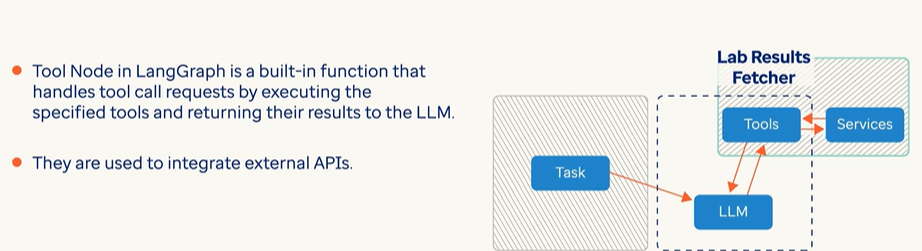
|  |  |
| --- | --- |
| Let’s say you bind a tool like this:  from langchain\_core.tools import tool  **@tool**  **def add(a: int, b: int) -> int:**  **"""Adds two numbers a and b."""**  **return a + b** | Now, if the user says:  “Can you add 5 and 7?”  The LLM will:   * Recognize the intent to perform addition. * Match it to the add tool based on the docstring. * Format the tool call: add(a=5, b=7) * Execute it and return the result. |

## TOOL NODE

A **Tool Node** is a **logical unit** in a chain or graph that:

* **Wraps a tool or function** (e.g., a calculator, web search, database query).
* **Executes a specific task** when triggered by the language model.
* **Returns structured output** that can be used in the conversation or passed to other nodes.

|  |
| --- |
| The **LLM** decides what action to take. 🡺It may call a **tool** (like a search engine or code executor🡺 The **Tool Node** handles that call and returns a result 🡺The result is wrapped in a ToolMessage and fed back into the conversation. |



Example Use Case

Imagine a user asks:

"What's the weather in Noida and convert it to Fahrenheit?"

The chain might look like this:

1. **HumanMessage**: User input.
2. **LLM**: Decides to call a weather API → triggers a **Tool Node**.
3. **Tool Node**: Calls the weather API and returns Celsius.
4. **LLM**: Uses the result to compute Fahrenheit.
5. **AIMessage**: Responds with the final answer.

A diagram of a computer program

AI-generated content may be incorrect.

STAGE 1

|  |
| --- |
| from langchain\_core.messages import HumanMessage  from langchain\_core.tools import tool  from langchain\_openai import AzureChatOpenAI  import os  from dotenv import load\_dotenv  from langchain.agents.output\_parsers import ReActSingleInputOutputParser  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"]  @tool  def get\_restaurant\_recommendations(location: str):      """Provides a list of top restaurant recommendations for a given location."""      recommendations = {          "munich": ["Hofbräuhaus", "Augustiner-Keller", "Tantris"],          "new york": ["Le Bernardin", "Eleven Madison Park", "Joe's Pizza"],          "paris": ["Le Meurice", "L'Ambroisie", "Bistrot Paul Bert"],      }      return recommendations.get(          location.lower(), ["No recommendations available for this location."]      )  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  # TODO: Bind the tool to the model  tools = [get\_restaurant\_recommendations]  llm\_with\_tools = llm.bind\_tools(tools)  messages = [HumanMessage("Recommend some restaurants in Munich.")]  # TODO: Invoke the llm  llm\_output = llm\_with\_tools.invoke(messages)  print("LLM Output:", llm\_output) |
| **OUTPUT**  LLM Output: content='' additional\_kwargs={'tool\_calls': [{'id': 'call\_mj9gU841Xr4IZRRb4Q3hLTI8', 'function': {'arguments': '{"location":"Munich"}', 'name': 'get\_restaurant\_recommendations'}, 'type': 'function'}], 'refusal': None} response\_metadata={'token\_usage': {'completion\_tokens': 20, 'prompt\_tokens': 59, 'total\_tokens': 79, 'completion\_tokens\_details': {'accepted\_prediction\_tokens': 0, 'audio\_tokens': 0, 'reasoning\_tokens': 0, 'rejected\_prediction\_tokens': 0}, 'prompt\_tokens\_details': {'audio\_tokens': 0, 'cached\_tokens': 0}}, 'model\_name': 'gpt-4o-mini-2024-07-18', 'system\_fingerprint': 'fp\_efad92c60b', 'id': 'chatcmpl-C5X7CXxUobJn5MVGWcoWSRxDR9ZDq', 'service\_tier': None, 'prompt\_filter\_results': [{'prompt\_index': 0, 'content\_filter\_results': {'hate': {'filtered': False, 'severity': 'safe'}, 'jailbreak': {'filtered': False, 'detected': False}, 'self\_harm': {'filtered': False, 'severity': 'safe'}, 'sexual': {'filtered': False, 'severity': 'safe'}, 'violence': {'filtered': False, 'severity': 'safe'}}}], 'finish\_reason': 'tool\_calls', 'logprobs': None, 'content\_filter\_results': {}} id='run--dee21e28-8bfb-48e5-a2f1-c058a2c86668-0' tool\_calls=[{'name': 'get\_restaurant\_recommendations', 'args': {'location': 'Munich'}, 'id': 'call\_mj9gU841Xr4IZRRb4Q3hLTI8', 'type': 'tool\_call'}] usage\_metadata={'input\_tokens': 59, 'output\_tokens': 20, 'total\_tokens': 79, 'input\_token\_details': {'audio': 0, 'cache\_read': 0}, 'output\_token\_details': {'audio': 0, 'reasoning': 0}} |

STAGE 2

|  |
| --- |
| from langchain\_core.tools import tool  from langgraph.prebuilt import ToolNode  from langchain\_core.messages import AIMessage  @tool  def get\_restaurant\_recommendations(location: str):      """Provides a list of top restaurant recommendations for a given location."""      recommendations = {          "munich": ["Hofbräuhaus", "Augustiner-Keller", "Tantris"],          "new york": ["Le Bernardin", "Eleven Madison Park", "Joe's Pizza"],          "paris": ["Le Meurice", "L'Ambroisie", "Bistrot Paul Bert"],      }      return recommendations.get(          location.lower(), ["No recommendations available for this location."]      )  tools = [get\_restaurant\_recommendations]  tool\_node = ToolNode(tools)  # TODO: Create an AIMessage for the tool call  message\_with\_tool\_call = AIMessage(      content="",      tool\_calls=[          {              "name": "get\_restaurant\_recommendations",              "args": {"location": "Munich"},              "id": "call\_mj9gU841Xr4IZRRb4Q3hLTI8",              "type": "tool\_call",          }      ],  )  # TODO: Invoke the ToolNode with the state and get the result  result = tool\_node.invoke(      {"messages": [message\_with\_tool\_call]}  )  # Pass as a list of messages  # TODO: Output the result  print(result) |
| **OUTPUT**  {'messages': [ToolMessage(content='["Hofbräuhaus", "Augustiner-Keller", "Tantris"]', name='get\_restaurant\_recommendations', tool\_call\_id='call\_mj9gU841Xr4IZRRb4Q3hLTI8')]} |

Both the stages are handled by LangGraph internally

## CODE EXAMPLE - TOOL CALLING IN LANGGRAPH

EXAMPLE 1 - To set up a tool to recommend a Restaurant for a given location

|  |
| --- |
| from langgraph.graph import END, START, StateGraph, MessagesState  from langchain\_openai import AzureChatOpenAI  from langchain\_core.tools import tool  from langgraph.prebuilt import ToolNode  from langchain\_core.messages import HumanMessage  # from util.langgraph\_util import display  from langgraph.checkpoint.memory import MemorySaver  import os  from dotenv import load\_dotenv  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,  )  @tool  def get\_restaurant\_recommendations(location: str):      """Provides a single top restaurant recommendation for a given location."""      recommendations = {          "munich": ["Hofbräuhaus", "Augustiner-Keller", "Tantris"],          "new york": ["Le Bernardin", "Eleven Madison Park", "Joe's Pizza"],          "paris": ["Le Meurice", "L'Ambroisie", "Bistrot Paul Bert"],      }      return recommendations.get(location.lower(), ["No recommendations available."])  @tool  def book\_table(restaurant: str, time: str):      """Books a table at a specified restaurant and time."""      return f"Table booked at {restaurant} for {time}."  # Bind the tool to the model  tools = [get\_restaurant\_recommendations, book\_table]  model = llm.bind\_tools(tools)  tool\_node = ToolNode(tools)  # TODO: Define functions for the workflow  def call\_model(state: MessagesState):      messages = state["messages"]      response = model.invoke(messages)      return {"messages": response}  # TODO: Define Conditional Routing  def should\_continue(state: MessagesState):      messages = state["messages"]      last\_mesage = messages[-1]      if last\_mesage.tool\_calls:          return "tools"      return END  # TODO: Define the workflow  workflow = StateGraph(MessagesState)  workflow.add\_node("agent", call\_model)  workflow.add\_node("tools", tool\_node)  workflow.add\_edge(START, "agent")  workflow.add\_conditional\_edges("agent", should\_continue)  workflow.add\_edge("tools", "agent")  graph = workflow.compile()  # display(graph)  # First invoke - Get one restaurant recommendation  response = graph.invoke(      {          "messages": [              HumanMessage(                  content="Can you recommend just one top restaurant in Munich? "                  "The response should contain just the restaurant name"              )          ]      }  )  # TODO: Extract the recommended restaurant  recommended\_resturant = response["messages"][-1].content  print(recommended\_resturant) |

EXAMPLE 2 – Check Symptoms and Recommend Doctor

|  |
| --- |
| from langchain\_openai import AzureChatOpenAI  from langchain\_core.messages import HumanMessage  from langgraph.graph import END, START, StateGraph, MessagesState  from langchain\_core.tools import tool  from langgraph.prebuilt import ToolNode  from langgraph.checkpoint.memory import MemorySaver  import os  from dotenv import load\_dotenv  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,  )  @tool  def check\_symptoms(symptom: str):      """Provides possible conditions based on the symptom described."""      conditions = {          "fever": ["Flu", "COVID-19", "Common Cold"],          "cough": ["Bronchitis", "Pneumonia", "Common Cold"],          "headache": ["Migraine", "Tension Headache", "Sinus Infection"],      }      return conditions.get(          symptom.lower(), ["No specific conditions found. Please consult a doctor."]      )  @tool  def book\_doctor\_appointment(specialty: str, date: str, time: str):      """Books an appointment with a doctor based on the required specialty."""      available\_specialties = [          "General Physician",          "Cardiologist",          "Neurologist",          "Pediatrician",      ]      if specialty in available\_specialties:          return f"Appointment booked with {specialty} on {date} at {time}."      else:          return f"Sorry, no available {specialty} at this time."  # Define tools  tools = [check\_symptoms, book\_doctor\_appointment]  # Initialize the LLM  llm\_with\_tools = llm.bind\_tools(tools)  # TODO: Create the ToolNode  tool\_node = ToolNode(tools)  # TODO: Implement the Node  def call\_model(state: MessagesState):      messages = state["messages"]      response = llm\_with\_tools.invoke(messages)      return {"messages": response}  # TODO: Define Conditional Routing  def should\_continue(state: MessagesState):      messages = state["messages"]      last\_message = messages[-1]      if last\_message.tool\_calls:          return "tools"      return END  # ✅ Define the Workflow  workflow = StateGraph(MessagesState)  workflow.add\_node("agent", call\_model)  workflow.add\_node("tools", tool\_node)  workflow.add\_edge(START, "agent")  workflow.add\_conditional\_edges("agent", should\_continue)  workflow.add\_edge("agent", "tools")  # ✅ Compile Workflow  checkpointer = MemorySaver()  graph = workflow.compile(checkpointer=checkpointer)  # ✅ Test the Workflow  config = {"configurable": {"thread\_id": "1"}}  # ✅ Step 1: Check Symptoms  response = graph.invoke(      {          "messages": [              HumanMessage(                  content="I have a fever. Can you tell me what this condition might be?"              )          ]      },      config,  )  print(response["messages"][-1])  # ✅ Extract the conditions  conditions = response["messages"][-1].content  print("\n🔍 \*\*Possible Conditions Based on Symptoms:\*\*")  print(conditions)  # ✅ Step 2: Book Doctor Appointment  response = graph.invoke(      {          "messages": [              HumanMessage(                  content="Book an appointment for these conditions"                  " with a General Physician for tomorrow at 10 AM."              )          ]      },      config,  )  # ✅ Extract the final response  final\_response = response["messages"][-1].content  # ✅ Print the final response  print("\n📅 \*\*Doctor Appointment Confirmation:\*\*")  print(final\_response) |

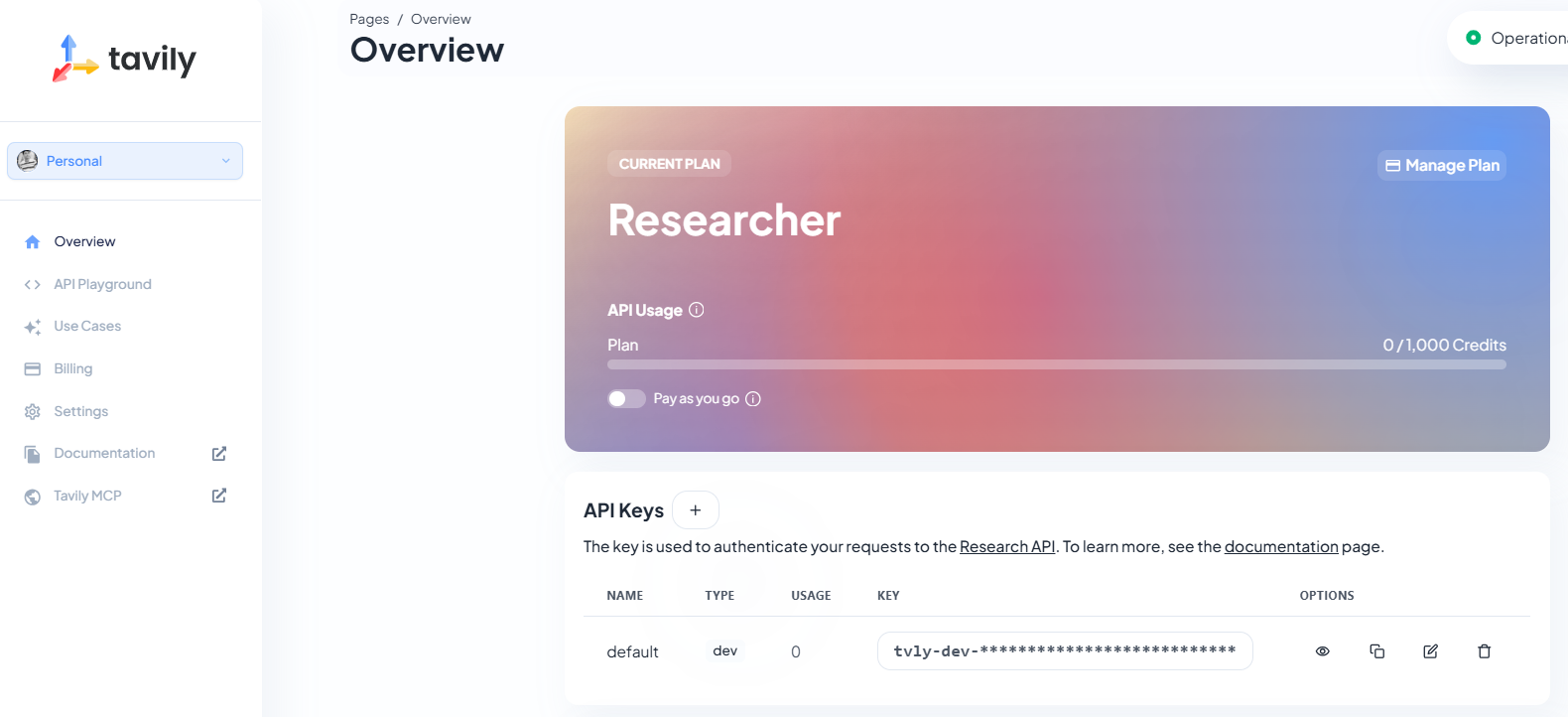
Example 3- TOOL CALL TO ADD NUMBERS

|  |
| --- |
| from langchain\_core.messages import HumanMessage, AnyMessage  from langchain\_core.tools import tool  from langchain\_openai import AzureChatOpenAI  import os  from dotenv import load\_dotenv  from langchain.agents.output\_parsers import ReActSingleInputOutputParser  from typing import Annotated  from langgraph.graph.message import add\_messages  from langgraph.graph import StateGraph, START, END  from langgraph.prebuilt import ToolNode, tools\_condition  from typing import TypedDict  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"]  class State(TypedDict):      messages: Annotated[list[AnyMessage], add\_messages]  @tool  def add(num1: int, num2: int) -> int:      """      Adds two integers and returns their sum.      Args:          num1 (int): The first integer to add.          num2 (int): The second integer to add.      Returns:          int: The sum of num1 and num2.      """      return num1 + num2  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  tools = [add]  # Binding Tools with LLM  llm\_with\_tools = llm.bind\_tools(tools)  def llm\_tool(state: State):      return {"messages": [llm\_with\_tools.invoke(state["messages"])]}  graph = StateGraph(State)  graph.add\_node("llm\_tool", llm\_tool)  graph.add\_node("tools", ToolNode(tools))  graph.add\_edge(START, "llm\_tool")  graph.add\_conditional\_edges(      "llm\_tool",      # If the latest message (result) from assistant is a tool call -> tools\_condition routes to tools      # If the latest message (result) from assistant is a not a tool call -> tools\_condition routes to END      tools\_condition,  )  graph.add\_edge("tools", END)  runnable = graph.compile()  messages = runnable.invoke({"messages": [HumanMessage("What is 5 + 3?")]})  for message in messages["messages"]:      message.pretty\_print() |

## MULTIPLE TOOL INTERGRATION

In the below example – we will integrate with multiple tools. The tools are

1. ArXiv: Seach Engine to search research papers
2. Wikipedia
3. Calculator function like add() and multiply()
4. Tavily – Web Search.
   1. Note for tavily – we need to create and use API key
   2. Add the API key in the environment variable



|  |
| --- |
| * Langchain has so many multiple in-built tools: <https://python.langchain.com/docs/integrations/tools/> |
| To INSTALL the in-built tools : **pip install arxiv wikipedia** |

A screenshot of a computer

AI-generated content may be incorrect.

|  |
| --- |
| How To Use Arxiv And WikiPedia Search  **from langchain\_community.utilities import WikipediaAPIWrapper, ArxivAPIWrapper**  **api\_wrapper = ArxivAPIWrapper(top\_k\_results=2, doc\_content\_chars\_max=500)**  **arxiv\_tool = ArxivQueryRun(api\_wrapper=api\_wrapper, verbose=True)**  **arxiv\_tool.invoke("Quantum computing")**  **wiki\_api\_wrapper = WikipediaAPIWrapper(top\_k\_results=2, doc\_content\_chars\_max=500)**  **wiki\_tool = WikipediaQueryRun(api\_wrapper=wiki\_api\_wrapper, verbose=True)**  **wiki\_tool.invoke("Quantum computing")** |
| How To Use Tavily SearcH  **from langchain\_tavily import TavilySearch**  **tavily\_tool = TavilySearch(**  **api\_key=tavily\_api\_key,**  **max\_results=5,**  **topic="general",**  **# include\_answer=False,**  **# include\_raw\_content=False,**  **# include\_images=False,**  **# include\_image\_descriptions=False,**  **# search\_depth="basic",**  **# time\_range="day",**  **# include\_domains=None,**  **# exclude\_domains=None**  **)**  **print(tavily\_tool.invoke({"query": "What happened at the last wimbledon"}))** |

|  |
| --- |
| from langchain\_core.messages import HumanMessage, AnyMessage  from langchain\_core.tools import tool  from langchain\_openai import AzureChatOpenAI  import os  from dotenv import load\_dotenv  from typing import Annotated  from langgraph.graph.message import add\_messages  from langgraph.graph import StateGraph, START, END  from langgraph.prebuilt import ToolNode, tools\_condition  from typing import TypedDict  from langchain\_community.tools import WikipediaQueryRun  from langchain\_community.utilities import WikipediaAPIWrapper  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"]  tavily\_api\_key = os.environ["TAVILY\_API\_KEY"]  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  ######################### Tool Defination  Start ##################################  @tool  def add(op1: int, op2: int) -> int:      """      Adds two integers and returns their sum.      Args:          op1 (int): The first operand.          op2 (int): The second operand.      Returns:          int: The sum of op1 and op2.      """      return op1 + op2  @tool  def multiply(op1: int, op2: int) -> int:      """      Multiplies two integers and returns the result.      Args:          op1 (int): The first operand.          op2 (int): The second operand.      Returns:          int: The product of op1 and op2.      """      return op1 \* op2  @tool  def wikipedia\_search(query: str) -> str:      """      Searches Wikipedia for the given query.      """      wiki\_api\_wrapper = WikipediaAPIWrapper(top\_k\_results=2, doc\_content\_chars\_max=500)      wiki\_tool = WikipediaQueryRun(api\_wrapper=wiki\_api\_wrapper, verbose=True)      return wiki\_tool.invoke(query)  # Return the result directly  ######################### Tool Defination  End ##################################  tools = [add, multiply, wikipedia\_search]  llm\_with\_tools = llm.bind\_tools(tools)  class State(TypedDict):      messages: Annotated[list[AnyMessage], add\_messages]  def llm\_tool(state: State):      return {"messages": [llm\_with\_tools.invoke(state["messages"])]}  graph = StateGraph(State)  graph.add\_node("llm\_tool", llm\_tool)  graph.add\_node("tools", ToolNode(tools))  graph.add\_edge(START, "llm\_tool")  graph.add\_conditional\_edges("llm\_tool", tools\_condition)  graph.add\_edge("tools", END)  runnable = graph.compile()  messages = runnable.invoke({"messages": [HumanMessage("What is 5 + 3?")]})  for message in messages["messages"]:      message.pretty\_print()  messages = runnable.invoke({"messages": [HumanMessage("What is 5 \* 3?")]})  for message in messages["messages"]:      message.pretty\_print()  messages = runnable.invoke({"messages": [HumanMessage("Who is Sachin Tendulkar?")]})  for message in messages["messages"]:      message.pretty\_print() |

# ReACT AGENT ARCHITECTURE

What is an Agent?

* An **agent** in the context of AI is a system that can **make decisions autonomously** based on inputs, tools, and logic. It doesn't just respond—it **reasons**, **acts**, and **learns from its environment**.

What is React Agent Architecture?

**React** stands for **Reason + Act**. It’s a **general agent architecture** that allows a language model (LLM) to:

* **Act**: Decide what to do based on the input.
* **Observe**: Look at the result of its action.
* **Reason**: Decide what to do next based on the result and the original input.

This loop continues until the agent decides to end the conversation.

Example

Let’s understand each component with your example:

**1. Act**

* The LLM receives a **natural language input**: *"Please add five plus five and then multiply by three."*
* It **decides to call a tool** (like an add function) because it understands that "add five plus five" is a task.
* This is the **Act** phase: the model takes an action.

**2. Observe**

* The tool (add) returns a result: 10. Instead of ending the conversation, the agent **feeds this result back to the LLM**.
* The LLM now has:
  + The original input
  + The result from the tool
* This is the **Observe** phase: the model observes the outcome of its action.

**3. Reason**

* The LLM now **reasons**:*"Okay, I’ve added five plus five and got ten. The next part of the input says 'multiply by three'."*
* It decides to call another tool (multiply) with inputs 10 and 3. The result is 30.
* Again, it observes this result and reasons: *"Is there anything else to do?"*
* Since the input is fully processed, it **ends the conversation**.

|  |
| --- |
| Simplified flow:  **Start → LLM Node → Tool Call → Tool Output → LLM Node → (Repeat or End)**  Each time the LLM receives new information (either input or tool output), it **reasons again**. |

Example Walkthrough

|  |  |
| --- | --- |
| **A diagram of a software  AI-generated content may be incorrect.** | **Input:***"Please add five plus five and then multiply by three."*  **Step-by-step:**   1. **Act**: LLM sees "add five plus five" → calls add(5, 5) → gets 10. 2. **Observe**: Tool returns 10 → LLM receives this and checks the rest of the input. 3. **Reason**: Sees "multiply by three" → calls multiply(10, 3) → gets 30. 4. **Observe Again**: Tool returns 30 → LLM checks if anything else is needed. 5. **Reason Again**: Input is fully processed → ends the conversation. |

## ReACT AGENT ARCHITECTURE IMPLEMENTATION

|  |
| --- |
| from langchain\_core.messages import HumanMessage, AnyMessage  from langchain\_core.tools import tool  from langchain\_openai import AzureChatOpenAI  import os  from dotenv import load\_dotenv  from typing import Annotated  from langgraph.graph.message import add\_messages  from langgraph.graph import StateGraph, START, END  from langgraph.prebuilt import ToolNode, tools\_condition  from typing import TypedDict  from langchain\_community.tools import WikipediaQueryRun  from langchain\_community.utilities import WikipediaAPIWrapper  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"]  tavily\_api\_key = os.environ["TAVILY\_API\_KEY"]  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  ######################### Tool Defination  Start ##################################  @tool  def add(op1: int, op2: int) -> int:      """      Adds two integers and returns their sum.      Args:          op1 (int): The first operand.          op2 (int): The second operand.      Returns:          int: The sum of op1 and op2.      """      return op1 + op2  @tool  def multiply(op1: int, op2: int) -> int:      """      Multiplies two integers and returns the result.      Args:          op1 (int): The first operand.          op2 (int): The second operand.      Returns:          int: The product of op1 and op2.      """      return op1 \* op2  @tool  def divide(op1: int, op2: int) -> float:      """      Divides the first integer by the second and returns the result.      Args:          op1 (int): The numerator.          op2 (int): The denominator.      Returns:          float: The result of division.      Raises:          ValueError: If op2 is zero.      """      if op2 == 0:          raise ValueError("Division by zero is not allowed.")      return op1 / op2  @tool  def wikipedia\_search(query: str) -> str:      """      Searches Wikipedia for the given query.      """      wiki\_api\_wrapper = WikipediaAPIWrapper(top\_k\_results=2, doc\_content\_chars\_max=500)      wiki\_tool = WikipediaQueryRun(api\_wrapper=wiki\_api\_wrapper, verbose=True)      return wiki\_tool.invoke(query)  # Return the result directly  ######################### Tool Defination  End ##################################  tools = [add, multiply, divide, wikipedia\_search]  llm\_with\_tools = llm.bind\_tools(tools)  class State(TypedDict):      messages: Annotated[list[AnyMessage], add\_messages]  def llm\_tool(state: State):      return {"messages": [llm\_with\_tools.invoke(state["messages"])]}  graph = StateGraph(State)  graph.add\_node("llm\_tool", llm\_tool)  graph.add\_node("tools", ToolNode(tools))  graph.add\_edge(START, "llm\_tool")  graph.add\_conditional\_edges("llm\_tool", tools\_condition)  graph.add\_edge("tools", "llm\_tool") 🡨 HERE THE TOOL RESPONSE IS SENT BACK TO THE LLM  runnable = graph.compile()  messages = runnable.invoke(      {          "messages": [              HumanMessage(                  "What is 5 + 3? and the product of 4 and 6? and the division of 20 by 4? Also, who is Sachin Tendulkar?"              )          ]      }  )  print(messages["messages"][-1].content) |
| 1. The sum of 5 and 3 is \*\*8\*\*.  2. The product of 4 and 6 is \*\*24\*\*.  3. The division of 20 by 4 is \*\*5.0\*\*.  Regarding Sachin Tendulkar:  Sachin Ramesh Tendulkar, born on April 24, 1973, is an Indian former international cricketer who captained the Indian national team. Often dubbed the "God of Cricket" in India, he is widely regarded as one of the greatest cricketers and batsmen of all time. He holds several world records, including being the all-time highest run-scorer in cricket. |

FOR MORE DETAILS AND EXAMPLE REFER AGENTIC AI DOCUMENT

# MEMORY

* **Memory** plays a crucial role in enabling **context-aware and persistent interactions** across nodes and steps.
* **Memory** in LangGraph refers to the **mechanism for storing and retrieving information** across different steps or nodes in a graph-based workflow.
* It allows the system to **remember past interactions, decisions, tool outputs, and user inputs**, enabling more intelligent and coherent behavior.

Example Scenario

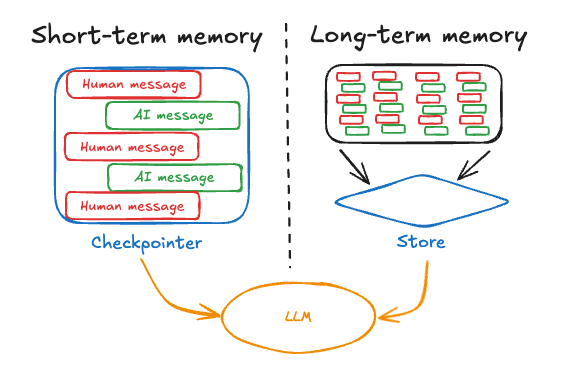
Imagine a LangGraph workflow for booking travel:

1. **User asks to book a flight** → HumanMessage
2. **LLM decides to call a flight search tool** → ToolNode
3. **Tool returns flight options** → stored in memory
4. **LLM uses memory to summarize options** → AIMessage
5. **User selects one** → memory updated
6. **LLM calls booking tool with selected flight** → uses memory to pass parameters

EXAMPLE – ISSUE WITHOUT MEMORY

|  |  |
| --- | --- |
|  | Without memory the LLM will not remember the previous interaction |
| from langgraph.graph import END, START, StateGraph, MessagesState  from langchain\_openai import AzureChatOpenAI  from langchain\_core.tools import tool  from langgraph.prebuilt import ToolNode  from langchain\_core.messages import HumanMessage  from langgraph.checkpoint.memory import MemorySaver  import os  from dotenv import load\_dotenv  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,  )  @tool  def get\_restaurant\_recommendations(location: str):      """Provides a single top restaurant recommendation for a given location."""      recommendations = {          "munich": ["Hofbräuhaus", "Augustiner-Keller", "Tantris"],          "new york": ["Le Bernardin", "Eleven Madison Park", "Joe's Pizza"],          "paris": ["Le Meurice", "L'Ambroisie", "Bistrot Paul Bert"],      }      return recommendations.get(location.lower(), ["No recommendations available."])  @tool  def book\_table(restaurant: str, time: str):      """Books a table at a specified restaurant and time."""      return f"Table booked at {restaurant} for {time}."  # Bind the tool to the model  tools = [get\_restaurant\_recommendations, book\_table]  model = llm.bind\_tools(tools)  tool\_node = ToolNode(tools)  # TODO: Define functions for the workflow  def call\_model(state: MessagesState):      messages = state["messages"]      response = model.invoke(messages)      return {"messages": response}  # TODO: Define Conditional Routing  def should\_continue(state: MessagesState):      messages = state["messages"]      last\_mesage = messages[-1]      if last\_mesage.tool\_calls:          return "tools"      return END  # TODO: Define the workflow  workflow = StateGraph(MessagesState)  workflow.add\_node("agent", call\_model)  workflow.add\_node("tools", tool\_node)  workflow.add\_edge(START, "agent")  workflow.add\_conditional\_edges("agent", should\_continue)  workflow.add\_edge("tools", "agent")  graph = workflow.compile()  # First invoke - Get one restaurant recommendation  response = graph.invoke(      {          "messages": [              HumanMessage(                  content="Can you recommend just one top restaurant in Munich? "                  "The response should contain just the restaurant name"              )          ]      }  )  # TODO: Extract the recommended restaurant  recommended\_resturant = response["messages"][-1].content  print(recommended\_resturant)  response = graph.invoke(      {          "messages": [              HumanMessage(                  content="Book a table on the same restaurant for tomorrow at 7 PM."              )          ]      }  )  # TODO: Extract the recommended restaurant  book\_table = response["messages"][-1].content  print(book\_table) | |
| **OUTPUT**  Hofbräuhaus  It seems there are no restaurant recommendations available for Mechanicsburg at the moment. Would you like to try a different location or modify your request? | |

## HOW MEMORY WORKS



### SHORT TERM MEMORY

|  |  |
| --- | --- |
|  | * Stored in-memory during the runtime of the program. * Managed using tools like **MemorySaver**. * Volatile: Lost when the program stops or restarts. * Useful for:   + Session-based workflows   + Temporary state tracking   + Quick prototyping |

#### THREAD IN LANGGRAPH?

* A thread is like a conversation session.
* Each thread has a unique thread\_id.
* If we use the same thread\_id across multiple turns, LangGraph knows *“****this is the same conversation****”* and retrieves the state for that thread(think of thread\_id like a chat session ID)

#### CHECKPOINT

A checkpoint is a snapshot of the graph’s state at a point in execution. Each step in execution (like calling a tool, responding to the user) may create a checkpoint.

It includes:

1. The messages so far (chat history).
2. The node states (what tools or agents have run).
3. Any intermediate values.

LangGraph uses checkpoints to:

1. Resume a graph if it was interrupted.
2. Rollback to a previous state if needed.
3. Continue conversations by loading the last checkpoint of a thread.

**Checkpoints are stored using a checkpointer (e.g., MemorySaver, SqliteSaver, RedisSaver).**

#### WHAT IS A THREAD-ID?

* When we use memory in LangGraph, every conversation or “session” is tied to a thread id.
* The thread id is essentially the unique identifier for a specific conversation context(think of it like a chat session ID in a messaging app).
* If we use the same thread id, the agent will recall the past conversation. If we use a new thread id, it starts fresh.

#### RELATIONSHIP BETWEEN THREAD ID AND CHECKPOINT

1. **Thread ID → groups all checkpoints that belong to the same conversation/session.**
2. **Checkpoint → represents a specific state in that conversation.**
3. Example:

You create a conversation with thread\_id = "user123". The graph executes step by step, storing checkpoints:

* *checkpoint\_1: after user’s first message.*
* *checkpoint\_2: after model reasoning.*
* *checkpoint\_3: after calling a tool.*

All these checkpoints are tied to "user123" thread id.

#### CHECKPOINTERS AND THREAD

|  |
| --- |
| * In LangGraph the state is passed across nodes as the graph is executed and in Langgraph handles memory using state. * Langgraph can use a checkpointer (MemorySaver)to automatically save the graph state after each step * This is very useful if we want to share this state across graph invocations, or we want to do an interrupt, we will do it when we use the human in the loop (i.e. interruption at a certain point, get some feedback, and then continue.) * From that point, we will store that state into a permanent store when the interrupt happens, and can retrieve back that state at a later point and continue with the graph execution.To accomplish this, we will use the checkpointer classes, namely **Memorysaver**. |

* **The Checkpoint class is an abstract class and Memorysaver is an implementation class.**
* **The instance of Memorysaver class is passed to the compile method**, which will be used to store our state from this point.

|  |
| --- |
| **checkpointer = MemorySaver()**  **graph = workflow.compile(checkpointer=checkpointer**) |

* Along with this while invoking the graph we need to pass in a configuration with a unique **thread id**(It will behave like a user session). The thread id can be any alphanumeric value.
* Pass configuration to the invoke() method along with message parameters.
* Include a thread ID, which acts like a unique session identifier.
* The thread ID ensures that the state is continuously saved using the MemorySaver.
* MemorySaver stores the state in-memory, not in a permanent database.
* As long as the program is running, the state tied to the thread ID remains accessible.
* When invoking the graph multiple times with the same thread ID:
* The graph remembers the previous state.
* You can resume execution from where it left off.
* This enables session continuity across invocations

|  |
| --- |
| **CREATE CONFIGURATION** |
| config = {      "configurable": {          "thread\_id": "1",🡨 It can be alpha-numeric      }  }  # Define configuration for memory |
| **PASS CONFIGURATION WHILE INVOKING** |
| **# First invoke - Get one restaurant recommendation**  response = graph.invoke(      {          "messages": [              HumanMessage(                  content="Can you recommend jst one top restaurant in Munich? "                  "The response should contain just the restaurant name",              )          ]      },      config=config,  )  **# Second invoke - Book Table**  response = graph.invoke(      {"messages": [HumanMessage(content="Book a table at this restaurant for 7 PM")]},      config=config,  ) |

#### HOW MEMORY WORKS WITH THREADS(ID) + CHECKPOINTS

When you call .invoke(...) with a thread\_id, LangGraph checks: Do we already have a checkpoint for this thread?

* ✅ Yes → Load checkpoint (restore history + state).
* ❌ No → Start a fresh state.

**After the agent runs, LangGraph saves a new checkpoint for that thread.Next time you invoke with the same thread\_id, the graph resumes from that checkpoint.**

##### Example – WITH THREAD ID

|  |
| --- |
| import os  from dotenv import load\_dotenv  from langchain\_openai import AzureChatOpenAI  from langgraph.checkpoint.memory import MemorySaver  from langgraph.prebuilt import create\_react\_agent  from langchain\_core.messages import HumanMessage, SystemMessage  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"]  embedding\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]  model = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,      temperature=0,  )  # Checkpointer (stores checkpoints in memory)  checkpointer = MemorySaver()  # Agent with memory  agent = create\_react\_agent(model, tools=[], checkpointer=checkpointer)  # Config with thread\_id (session)  config = {"configurable": {"thread\_id": "session\_1"}}  # First turn  response1 = agent.invoke(      {"messages": [HumanMessage(content="My favorite color is blue.")]}, config=config  )  print(response1["messages"][-1].content)  # Second turn (same thread\_id → memory works)  response2 = agent.invoke(      {"messages": [HumanMessage(content="What’s my favorite color?")]}, config=config  )  print(response2["messages"][-1].content)  # Third turn (new thread\_id → no memory)  config2 = {"configurable": {"thread\_id": "session\_2"}}  response3 = agent.invoke(      {"messages": [HumanMessage(content="What’s my favorite color?")]}, config=config2  )  print(response3["messages"][-1].content) |
| OUTPUT  That's great! Blue is often associated with calmness, stability, and tranquility. Do you have a specific shade of blue that you like the most?  Your favorite color is blue!  I don't have access to personal information about you, so I can't know your favorite color. However, if you tell me what it is, I'd be happy to chat about it! |

#### EXAMPLE – RESTAURANT BOOKING

|  |
| --- |
| from langgraph.graph import END, START, StateGraph, MessagesState  from langchain\_openai import AzureChatOpenAI  from langchain\_core.tools import tool  from langgraph.prebuilt import ToolNode  from langchain\_core.messages import HumanMessage  # from util.langgraph\_util import display  from langgraph.checkpoint.memory import MemorySaver  import os  from dotenv import load\_dotenv  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,  )  @tool  def get\_restaurant\_recommendations(location: str):      """Provides a single top restaurant recommendation for a given location."""      recommendations = {          "munich": ["Hofbräuhaus", "Augustiner-Keller", "Tantris"],          "new york": ["Le Bernardin", "Eleven Madison Park", "Joe's Pizza"],          "paris": ["Le Meurice", "L'Ambroisie", "Bistrot Paul Bert"],      }      return recommendations.get(location.lower(), ["No recommendations available."])  @tool  def book\_table(restaurant: str, time: str):      """Books a table at a specified restaurant and time."""      return f"Table booked at {restaurant} for {time}."  # Bind the tool to the model  tools = [get\_restaurant\_recommendations, book\_table]  model = llm.bind\_tools(tools)  tool\_node = ToolNode(tools)  # TODO: Define functions for the workflow  def call\_model(state: MessagesState):      messages = state["messages"]      response = model.invoke(messages)      return {"messages": response}  # TODO: Define Conditional Routing  def should\_continue(state: MessagesState):      messages = state["messages"]      last\_mesage = messages[-1]      if last\_mesage.tool\_calls:          return "tools"      return END  # TODO: Define the workflow  workflow = StateGraph(MessagesState)  workflow.add\_node("agent", call\_model)  workflow.add\_node("tools", tool\_node)  workflow.add\_edge(START, "agent")  workflow.add\_conditional\_edges("agent", should\_continue)  workflow.add\_edge("tools", "agent")  checkpointer = MemorySaver()  graph = workflow.compile(checkpointer=checkpointer)  config = {      "configurable": {          "thread\_id": "1",      }  }  # Define configuration for memory  # First invoke - Get one restaurant recommendation  response = graph.invoke(      {          "messages": [              HumanMessage(                  content="Can you recommend jst one top restaurant in Munich? "                  "The response should contain just the restaurant name",              )          ]      },      config=config,  )  # TODO: Extract the recommended restaurant  recommended\_resturant = response["messages"][-1].content  print(recommended\_resturant)  # Second invoke - Book Table  response = graph.invoke(      {"messages": [HumanMessage(content="Book a table at this restaurant for 7 PM")]},      config=config,  )  # TODO: Extract the recommended restaurant  book\_table = response["messages"][-1].content  print(book\_table) |
| **OUTPUT**  Hofbräuhaus  Table booked at Hofbräuhaus for 7 PM. |

### LONG-TERM MEMORY

* Persisted to external storage like databases (e.g., PostgreSQL).
* Enables state recovery across sessions, even after restarts.
* Useful for:
  + Human-in-the-loop workflows
  + Multi-session applications
  + Audit trails and history tracking

# 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

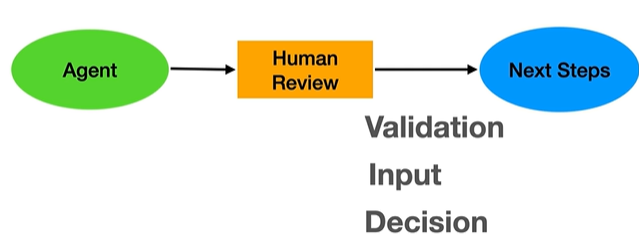
LangGraph supports both **synchronous** and **asynchronous** streaming:

* **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

# HUMAN IN THE LOOP(HITL)

* Human-in-the-Loop (HITL) refers to a design pattern where human intervention is intentionally introduced into an automated workflow.



* In LangGraph, this means pausing the graph execution at specific points to allow a human to:
  + Review or validate outputs
  + Provide feedback or corrections
  + Make decisions before continuing

How It WORKS?

1. Graph Execution Begins: The graph runs through its nodes, passing state and messages.
2. Interrupt Point Defined: A node is configured to pause execution and wait for human input.
3. State Is Saved: Using a checkpointer (e.g., MemorySaver), the current state is stored.
4. Human Provides Input: The user reviews the state/output and provides feedback or a decision.
5. Graph Resumes: Execution continues from the saved state using the same thread ID.

## USECASE – HITL

Use Case 1: Human Review Before Sending an AI-Generated Email

A diagram of a diagram

AI-generated content may be incorrect.

* An **AI agent summarizes an internal report** intended for stakeholders within a company.
* Before the email is sent, the **summary undergoes human review** — a perfect example of Human-in-the-Loop.
* A **manager reviews the summary** and chooses to either:
  + ✅ **Accept** it — the email is sent.
  + ❌ **Reject** it — the agent re-summarizes the report.
* The agent may then generate **multiple revised summaries**.
* These options are presented using a **review tool**, and a human selects the most appropriate one.
* This ensures **quality control**, **accountability**, and **collaboration** between AI and humans

## EXAMPLE

|  |  |
| --- | --- |
|  | * In the example - we are going to work on a **code and test generation** use case, and the test will be generated only if we as human approve the code that is generated by the LLM. * To implement we use the interrupt function and the command object from lang graph |

# RAG WITH LANGGRAPH

Let us proceed systematically to understand the operational mechanisms of a Generative AI System.

A diagram of a question

AI-generated content may be incorrect.

1. Imagine a user ask questions in an AI system.
2. Each question gets incorporated with a prompt (system prompt) to give the AI model instructions.
3. Note: The prompt helps the AI know how to answer (like behave like Ai Assistant)

|  |
| --- |
| what is RAG?  RAG stands for **Retrieval-Augmented Generation**. It’s a way to make LLMs smarter by giving them **extra context** from external sources (like a vector database) before they generate a response |

**Now let’s see how a Traditional RAG works**

|  |  |
| --- | --- |
|  | * Imagine a user asks: **“What is the company’s holiday policy?”**   In a **traditional RAG system**:   1. **User sends a query** 2. The query goes to a **vector database** (which stores documents as vectors) 3. The database finds relevant info (like the company’s policy) 4. That info is combined with a **prompt** (which tells the LLM how to behave) 5. The LLM uses this combo to generate a response 6. The user gets an answer.   Key point: In traditional RAG, the LLM is used **only once**—just to generate the final output. |

## AGENTIC RAG

A screen shot of a diagram

AI-generated content may be incorrect.

**Agentic RAG** is like traditional RAG—but smarter. It uses **agents** (powered by LLMs) to:

* Handle **complex tasks**
* Make **dynamic decisions**
* Choose **which database to query**
* Possibly **loop or retry** if needed
* So instead of a fixed flow, the system **thinks** about what to do next.

### EXAMPLE: MULTIPLE VECTOR DATABASES

Let’s say you have two vector DBs:

1. **DB1**: Company policies
2. **DB2**: Legal documents

Now a user asks**:“What are the legal implications of remote work?”**

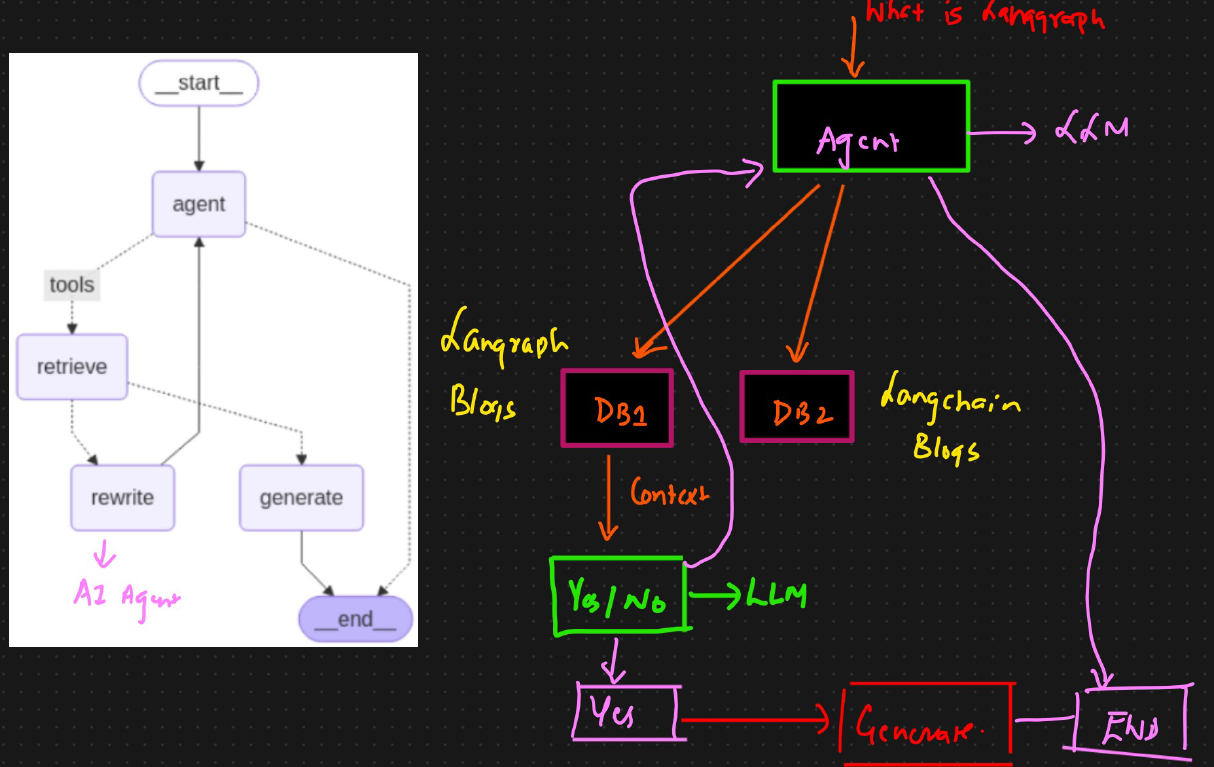
* In traditional RAG
  + We have have to manually decide which DB to query.
* But in **Agentic RAG**,
  + The **agent (LLM)** looks at the query and says: Since this is legal-related. Let me query DB2.”
  + It **routes the query intelligently**. That’s the magic of Agentic RAG.
  + **if the query doesn’t match any DB.** Let’s say the user asks: “Who won the recent World Cup?” for which agentic AI does not have answer. To cater this we can define a **fail route**:
    - 1. The agent checks all DBs
      2. Finds nothing relevant
      3. Responds: “Sorry, I don’t have that information.”
      4. This keeps your assistant **focused and accurate**.

HOW LANGGRAPH HELPS

For the above use case - LangGraph helps in building these Agentic System

* Step 1: Agents make decisions
* Step 2: Tool calls are made to query databases
* Step 3: Function calls process the data
* Step 4: Conditional paths, such as retry or fail, are handled

## AGENTIC RAG IMPLEMENTATION



### OBJECTIVE

* **Purpose**: To build a graph-based, agent-driven Retrieval-Augmented Generation (RAG) system that intelligently retrieves, evaluates, rewrites, and summarizes user queries using multiple vector databases and LLMs.
* **Goal**: Demonstrate how agentic workflows can enhance precision and relevance in responses by dynamically selecting and interacting with context sources.

|  |
| --- |
| Highlights of Agentic RAG Design   * **Agentic Intelligence**: Agents autonomously decide which vector DB to query and whether to rewrite or summarize. * **Multi-DB Integration**: Supports multiple vector databases for richer and more diverse context retrieval. * **LLM-Driven Decisions**: Uses language models to evaluate context relevance and perform query rewriting. * **Graph-Based Workflow**: Entire system is orchestrated using a graph structure for modularity and clarity. |

### KEY COMPONENTS & WORKFLOW

1. Agent Node (Start Node)

* Acts as the entry point for user queries.
* Integrated with LLMs to understand and route queries intelligently.
* Connected to two vector databases:
  + **DB1**: Contains LangGraph blocks.
  + **DB2**: Contains LangChain blocks.

2. Retrieval Node

* Triggered by the agent based on query analysis.
* Retrieves context from the appropriate vector DB (DB1 or DB2).
* Selection logic:
  + If the query is about LangGraph → retrieve from DB1.
  + If the query is about LangChain → retrieve from DB2.

3. Context Evaluation Node

* Uses an LLM to compare the retrieved context with the original query.
* Decision output:
  + **Yes** → Context is relevant.
  + **No** → Context is not relevant.

4. Rewrite Node (Conditional)

* Activated if context is deemed irrelevant.
* Rewrites the user query to better align with available context.
* Example: Original query “What is LangGraph?” → rewritten to “What are the key features of LangGraph?”

5. Summary Generation Node

* Activated if context is relevant (Yes).
* Generates a concise, context-aware summary using the retrieved information.

6. End Node

* Final output is either:
* A rewritten query (if context was bad).
* A generated summary (if context was good).
* Response is sent back to the user.