

## Langgraph Tutorial | Agentic AI crash course



3,591 views Premiered 8 hours ago [\[embed\]](#)  
Learn Agentic AI using a popular framework langgraph. In this video, we will start with agentic AI basics and then we will dive deeper into langgraph by covering a wide range of topics.

Check out PyCharm, the IDE built for data science and AI/ML professionals: <https://jb.gg/try-pycharm-now>  
Free 3-Month PyCharm Pro subscription with coupon code "PyCharm2codebasics": [https://jb.gg/pc\\_codebasics](https://jb.gg/pc_codebasics)  
Redeem on new purchases before Feb 01, 2026

Code: <https://github.com/codebasics/langgra...>

00:00 Introduction  
00:40 Agentic AI Basics  
00:00 What is Langgraph  
07:41 Langchain vs Langgraph  
13:08 Installation and Setup  
19:00 Simple Graph  
31:36 Graph with Condition  
36:55 Chatbot in Langgraph  
45:38 Chatbot with Tool  
56:03 Memory  
1:05:05 Tracing with Langsmith  
1:10:11 Human in the Loop

<https://github.com/codebasics/langgraph-crash-course>

Codebasics / langgraph-crash-course

Type  to search

Code Issues Pull requests Actions Projects Security Insights

**langgraph-crash-course** Public

Watch 0 Fork 0 Star 1

**main** 1 Branch 0 Tags

Go to file Add file

**About**

Agentic AI crash course using langgraph framework

Readme Activity Custom properties 1 star 0 watching 0 forks Report repository

**Releases**

No releases published

**Packages**

No packages published

**Languages**

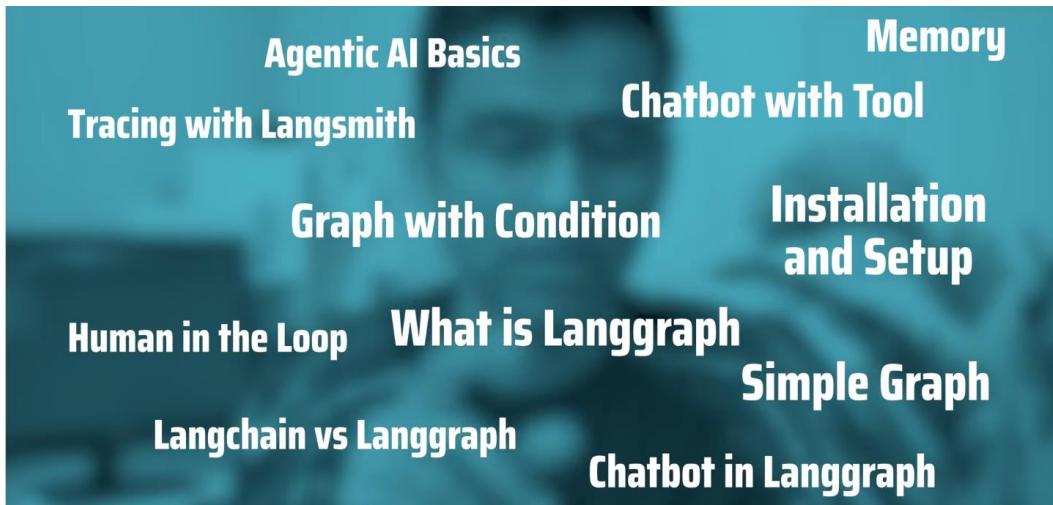
Jupyter Notebook 98.1% Python 1.9%

**README**

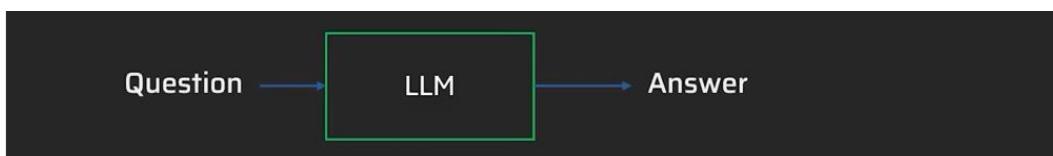
### Agentic AI Crash Course using Langgraph

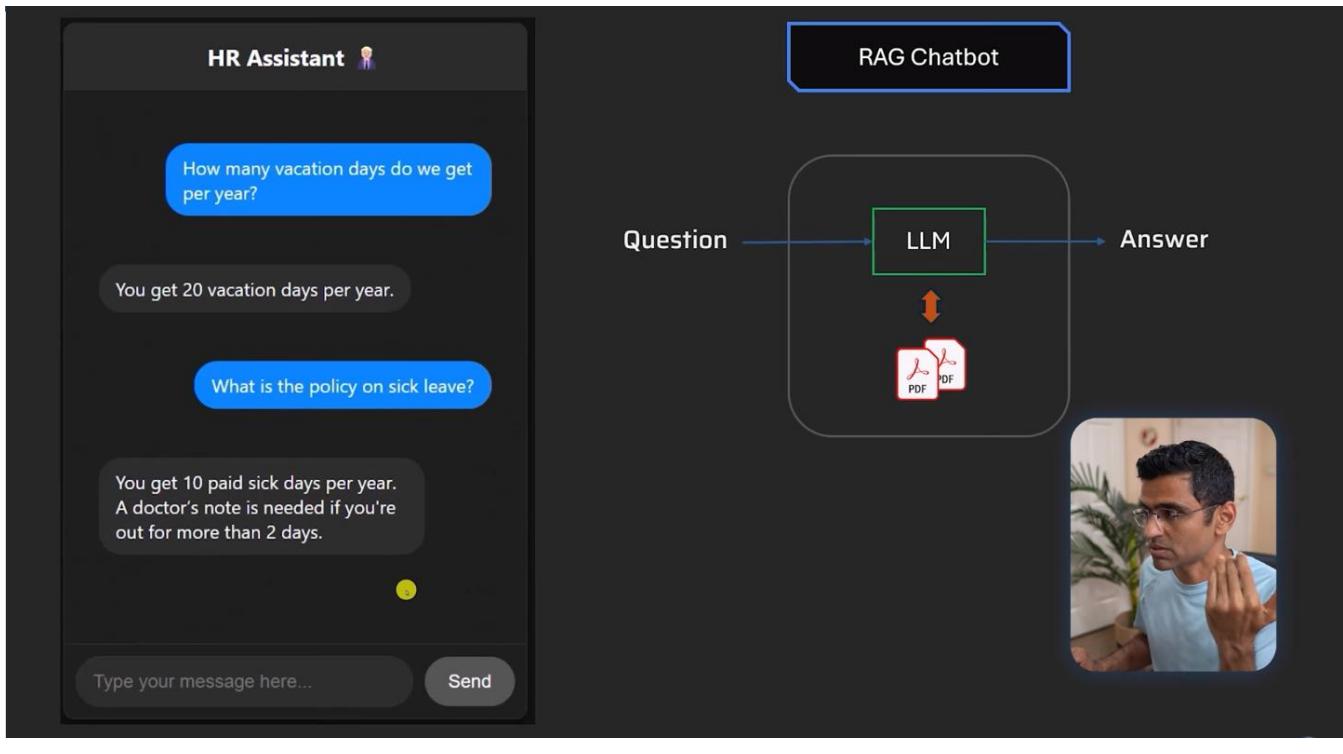
#### Setup Instructions

1. Rename sample.env to .env file and add your keys. You can find the instructions on key generation through the YouTube video on codebasics channel
2. Follow YouTube video instructions to install uv, PyCharm pro and other packages



## What is an AI Agent





[anthropic.com/engineering/building-effective-agents](https://anthropic.com/engineering/building-effective-agents)

A

Claude API Solutions Research Commitments Learn News

Try Claude

"Agent" can be defined in several ways. Some customers define agents as fully autonomous systems that operate independently over extended periods, using various tools to accomplish complex tasks. Others use the term to describe more prescriptive implementations that follow predefined workflows. At Anthropic, we categorize all these variations as **agentic systems**, but draw an important architectural distinction between **workflows** and **agents**:

- Workflows are systems where LLMs and tools are orchestrated through predefined code paths.
- Agents, on the other hand, are systems where LLMs dynamically direct their own processes and tool usage, maintaining control over how they accomplish tasks.

Below, we will explore both types of agentic systems in detail. In Appendix 1 ("Agents in Practice"), we describe two domains where customers have found particular value in using these kinds of systems.



**HR Assistant** 🧑

How many leaves do I have left this year?

You have 7 leave days remaining this year

Apply for 2 days of casual leave on May 5 and 6

Your 2-day casual leave for May 5 and 6 has been successfully applied.

Type your message here... **Send**

**Tool Augmented Chatbot**

The diagram illustrates the architecture of the Tool Augmented Chatbot. A 'Question' is input into an 'LLM' (Large Language Model). The LLM generates an 'Answer'. The process involves interactions with external tools: a PDF file icon, an 'API' box, and a database cylinder icon, represented by double-headed arrows indicating bidirectional communication between the LLM and these tools.

**HR Assistant** 🧑

Prepare for Sarah's maternity leave.

I've reviewed Sarah's due date and workload. A transition plan has been drafted, backup resources identified, and a handover meeting scheduled for next Monday.

Send

**HR Assistant**

I've reviewed Sarah's due date and workload. A transition plan has been drafted, backup resources identified, and a handover meeting scheduled for next Monday.

Onboard the new intern joining next Monday.

I've prepared the onboarding checklist, scheduled a welcome meeting, and sent access requests to IT. The intern's welcome kit has also been ordered.

Type your message here... Send

**Agentic Chatbot**

Question → LLM → API, Calendar, One Drive, Database → Answer

Onboard the new intern joining next Monday.

Schedule welcome meeting

Create intern's profile in HR Management System

IT Helpdesk (Wi-Fi credentials, Email, Slack access)

Order Laptop, ID Card

LLM

Goal Oriented Planning

Multi-step Reasoning

Autonomous Decision Making

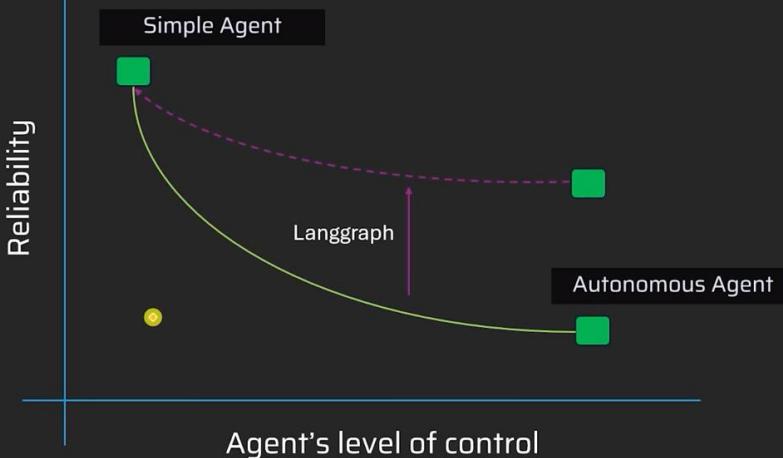
Tool, Knowledge, Memory

**AI Agent** can make decisions and take actions on its own to achieve a goal without being told exactly what to do at every step.

**Agentic AI** is a system using one or more agents

Type	Reactive	Tool Use	Reasoning	Planning	Proactivity
RAG Chatbot	✓	✗	✗	✗	✗
Tool-Augmented Chatbot	✓	✓	✗	✗	✗
Agentic AI	✓	✓	✓	✓	✓

# Reliability



## Langgraph helps you build reliable agents

### Langgraph

Provides a more expressive framework to build highly customizable, complex agents

Langgraph excels at graph-based, stateful orchestration - e.g. multi-step, workflows with memory, streaming, human-in-the-loop control

### Other Frameworks

#### Agno

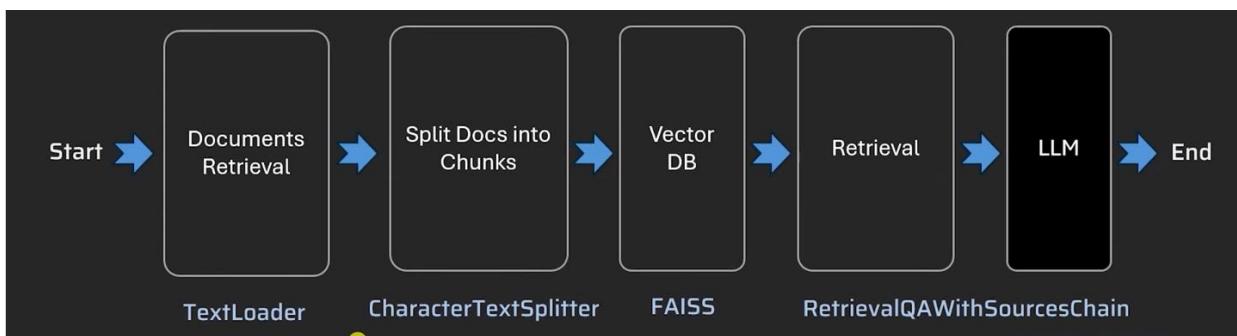
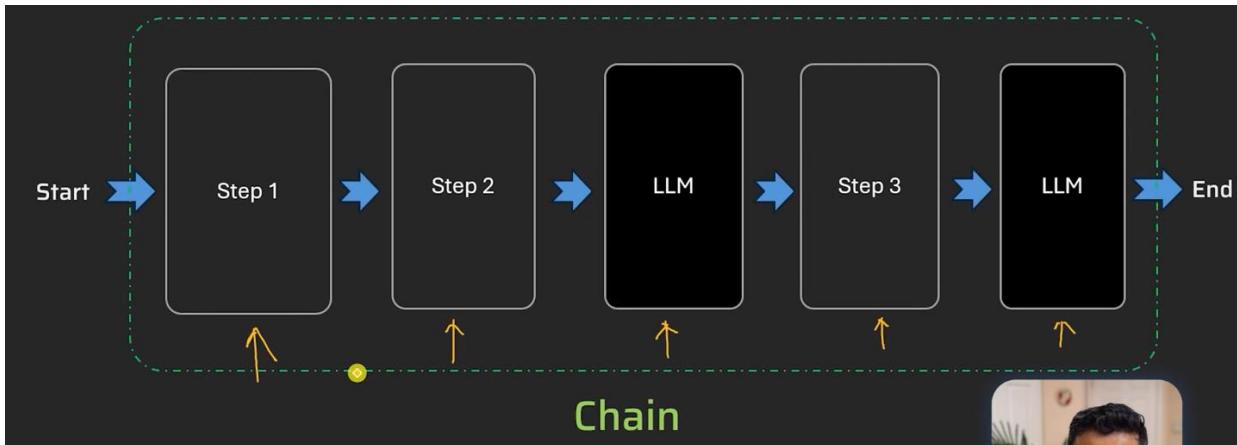
Lightweight, fast

#### Google ADK

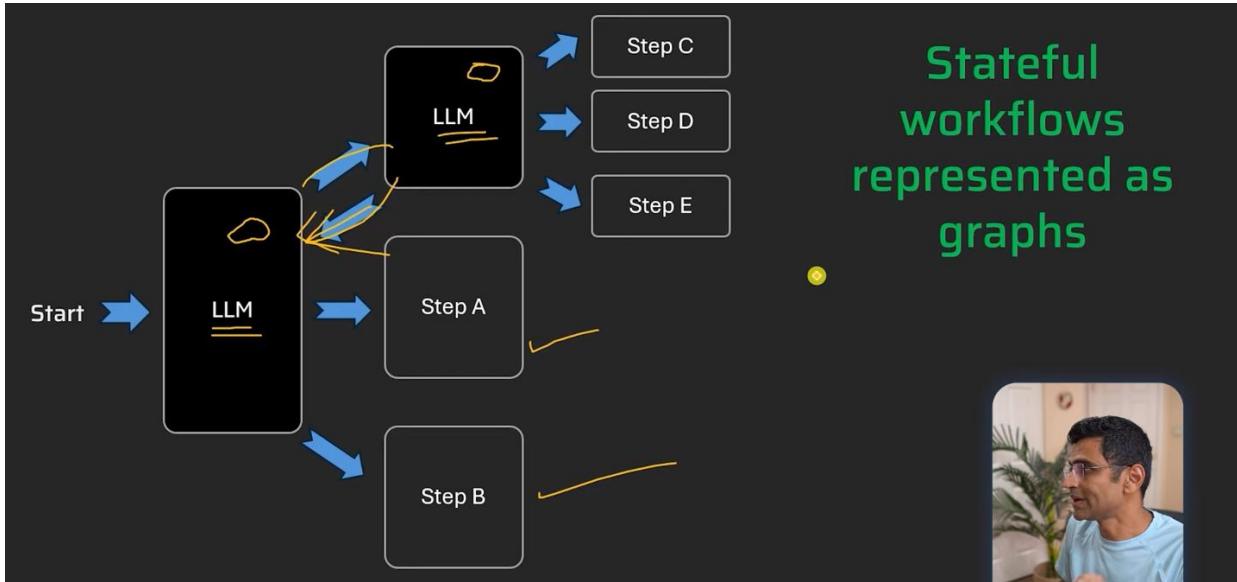
Tight GCP, Multi-agent



# Langchain vs Langgraph



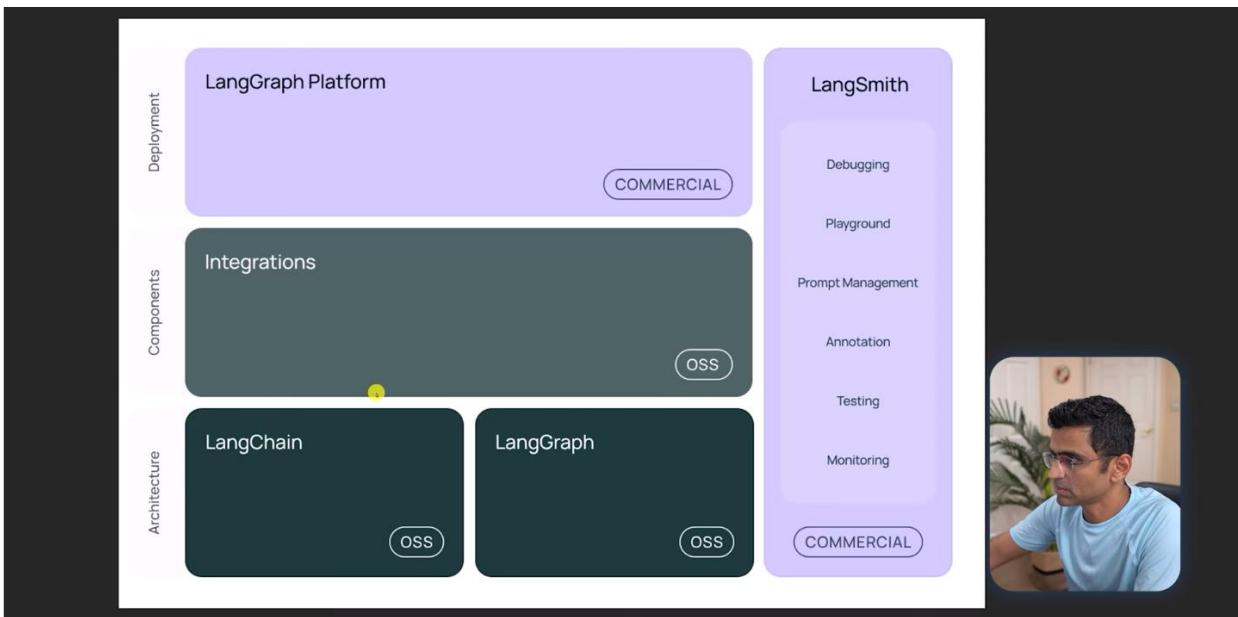
These are some of the classes and tools provided in the Langchain framework as utilities.



Autonomy is important here and Langgraph framework is best for these graph-like, stateful workflow solutions.

Feature	LangChain	LangGraph
Purpose	Toolkit to build LLM apps (chains, tools, agents)	Framework to manage complex workflows with state
Style	Linear or reactive chains	Graph-based, supports loops, retries, memory
Best Use Case	Simple chatbots, RAG apps, tool usage	Multi-step workflows, agents with memory, conditional paths
State Handling	Stateless or partially stateful	Fully stateful; remembers and transitions based on logic
Example Use	"Book a flight" using a flight API	"Plan a vacation" (ask budget → choose flights → book hotel → loop if error)





PyCharm JetBrains IDEs

Use Cases ▾ EAP What's New Features ▾ Learn ▾ Pricing Download

# PyCharm

## The only Python IDE you need

Built for web, data, and AI/ML professionals. Supercharged with an AI-enhanced IDE experience.

Download

Free forever, plus one month of Pro included




**JETBRAINS**

AI Developer Tools Team Tools Education Solutions Support Store

PyCharm JetBrains IDEs Use Cases ▾ EAP What's New Features ▾ Learn ▾ Pricing Download

Windows macOS Linux

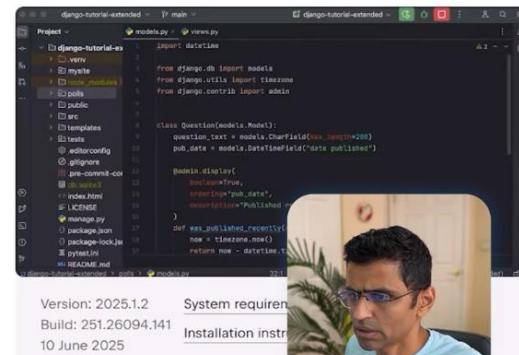


The only Python IDE you need

[Download](#)

.exe (Windows) ▾

Free forever, plus one month of Pro included



## PyCharm is now one unified product!

All users now automatically start with a free one-month Pro trial. After

## Thank you for downloading PyCharm!

Your download should start shortly. If it doesn't, please use the [direct link](#).

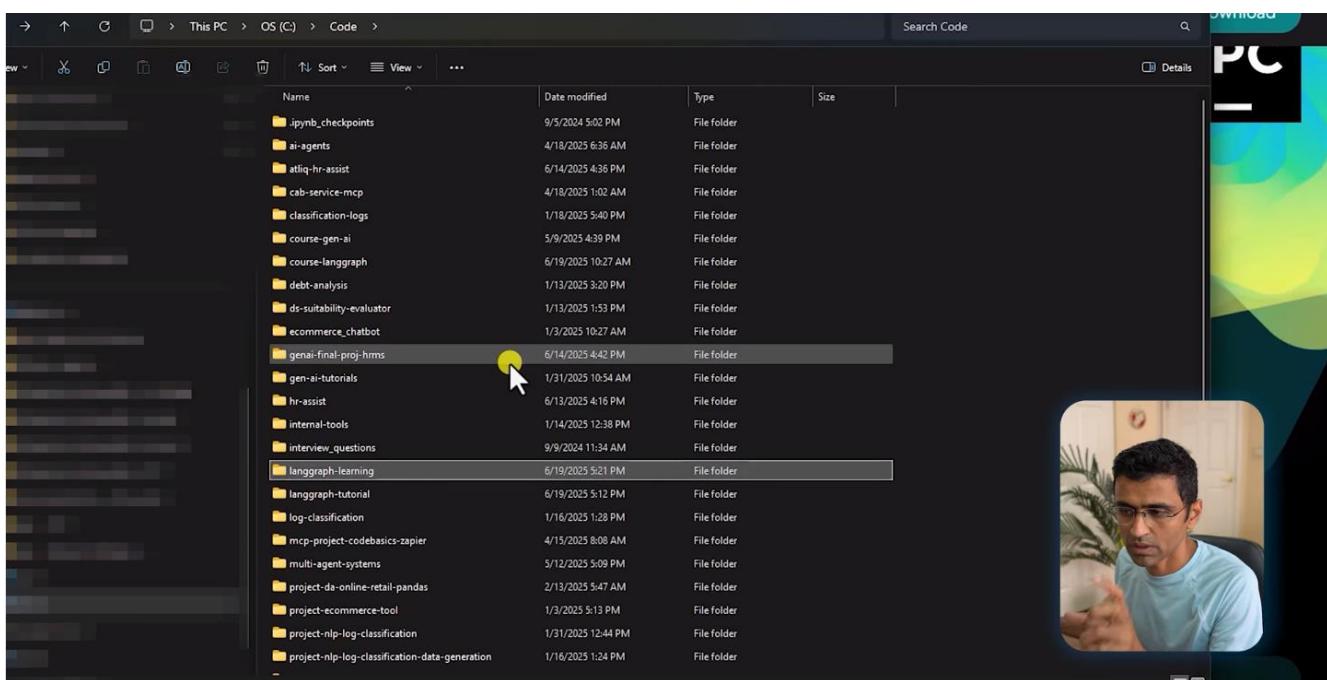
Download and verify the file [SHA-256 checksum](#).

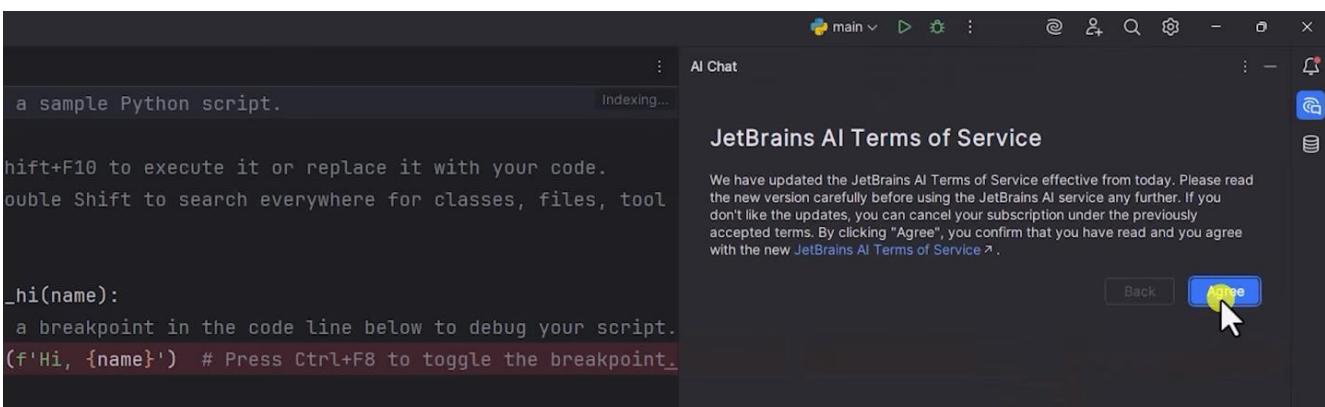
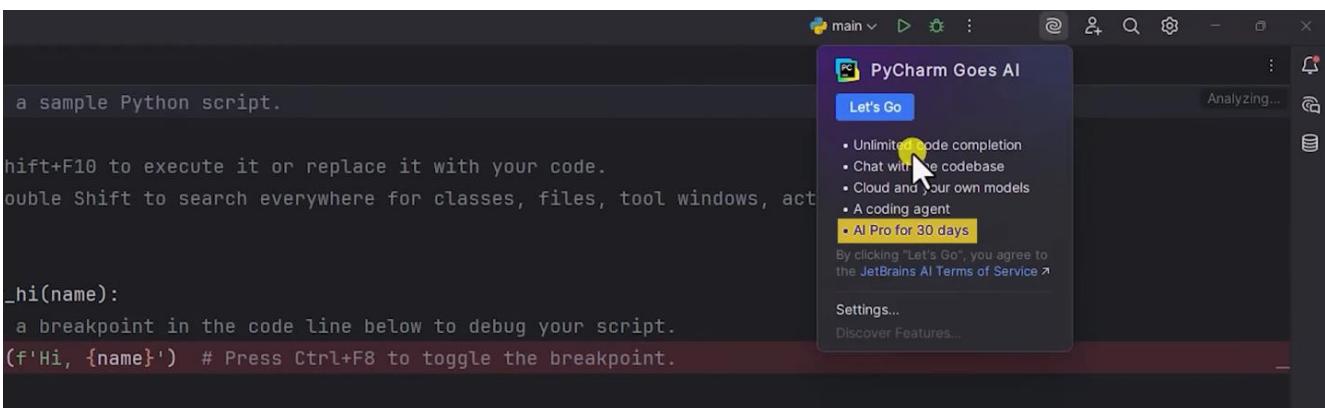
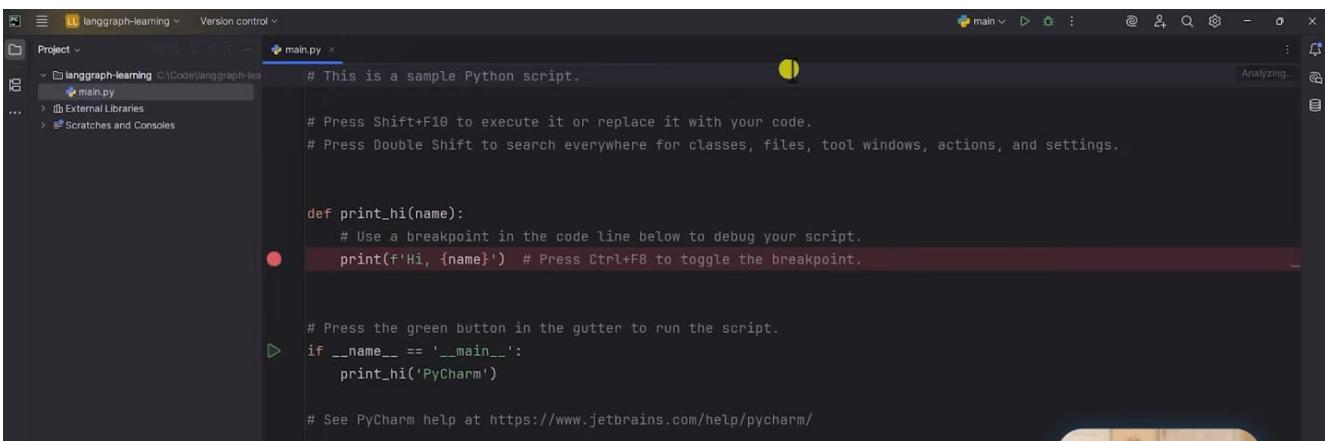
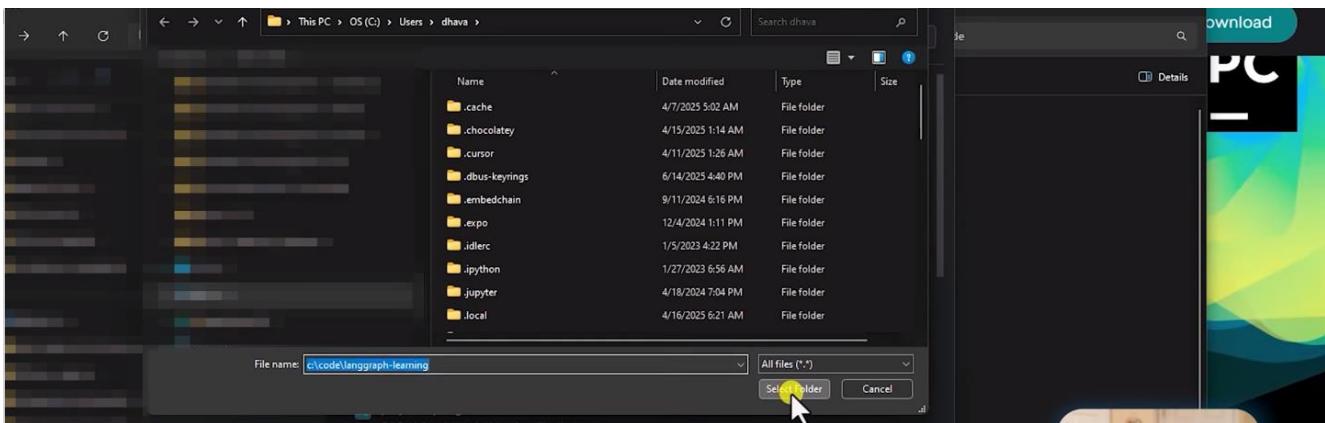
Learn more about the [digital signatures of JetBrains binaries](#).



PyCharm goes AI! Less routine, more coding joy. All refined JetBrains AI tools – right in your IDE, for free.

[Get started](#)





```
he green button in the gutter to run the script.  
__ == '__main__':  
    hi('PyCharm')  
  
PyCharm help at https://www.jetbrains.com/help/pycharm/
```

Model

- Claude 3.5 Sonnet
- GPT-4o
- o1
- o3
- o3-mini
- o4-mini
- GPT-4.1
- GPT-4.1 mini
- GPT-4.1 nano
- Gemini 2.5 Pro (Experimental)
- Gemini 2.5 Flash
- Gemini 2.0 Flash
- Claude 3.5 Haiku
- Claude 3.7 Sonnet
- Claude 4 Sonnet

LM Studio

Ask A... Connect... #mentions) or /commands

Ollama

Code Connect... Instructions Empty

+ Claude...Sonnet Chat Share feedback

Waiting for "Scanning" to complete... Show all (2) 1:1 CRLF UTF-8 4 spaces Python 3.10 (langgraph-tutorial)

```
Waiting for "Scanning" to complete... Show all (2) 1:1 CRLF UTF-8 4 spaces Python 3.10 (langgraph-tutorial)
```

Ask AI Assistant, use @mentions (#mentions) or /commands

Codebase Off main.py Current Instructions Empty

+ Claude...Sonnet Chat Share feedback

## Installation

Installation methods

Install uv with our standalone installers or your package manager of choice.

### Standalone installer

uv provides a standalone installer to download and install uv:

macOS and Linux Windows

Use `irm` to download the script and execute it with `iex`:

```
PS> powershell -ExecutionPolicy Bypass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

Changing the `execution policy` allows running a script from the internet.

Request a specific version by including it in the URL:

```
PS> powershell -ExecutionPolicy Bypass -c "irm https://astral.sh/uv/0.7.13/install.ps1 | iex"
```

**Tip**

The installation script may be inspected before use:

macOS and Linux Windows

```
PS> powershell -ExecutionPolicy Bypass -c "irm https://astral.sh/uv/install.ps1 | more"
```

Alternatively, the installer or binaries can be downloaded directly from GitHub.

See the reference documentation on the [installer](#) for details on customizing your uv installation.

uv 0.7.13 58.4k 1.7k

Installation methods

- Standalone installer
- PyPI
- Cargo
- Homebrew
- WinGet
- Scoop
- Docker
- GitHub Releases

Upgrading uv

Shell autocompletion

Uninstallation

Next steps

A video feed of a man in a red shirt is visible in the bottom right corner.

```
Microsoft Windows [Version 10.0.22631.5472]
(c) Microsoft Corporation. All rights reserved.

C:\Users\dhava>powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/u/v/install.ps1 | iex"
Downloading uv 0.7.13 (x86_64-pc-windows-msvc)
Installing to C:\Users\dhava\.local\bin
We encountered an error trying to perform the installation;
please review the error messages below.

The process cannot access the file 'C:\Users\dhava\.local\bin\uv.exe' because
it is being used by another process.

C:\Users\dhava>
```

uv 0.7.13 (x86\_64-pc-windows-msvc)

Installing to C:\Users\dhava\.local\bin

We encountered an error trying to perform the installation;  
please review the error messages below.

The process cannot access the file 'C:\Users\dhava\.local\bin\uv.exe' because  
it is being used by another process.

We will use the installed **uv** to install other python packages

```
# Press Shift + F11 to start the debugger
# Press Ctrl + Shift + F11 to start the debugger with breakpoints
# Press Ctrl + Shift + F11 to start the debugger with breakpoints
dhava@DP MINGW64 /c/Code
$ cd langgraph-learning/
dhava@DP MINGW64 /c/Code/langgraph-learning
$ uv init
Initialized project `Langgraph-learning`
```

dhava@DP MINGW64 /c/Code/langgraph-learning (master)

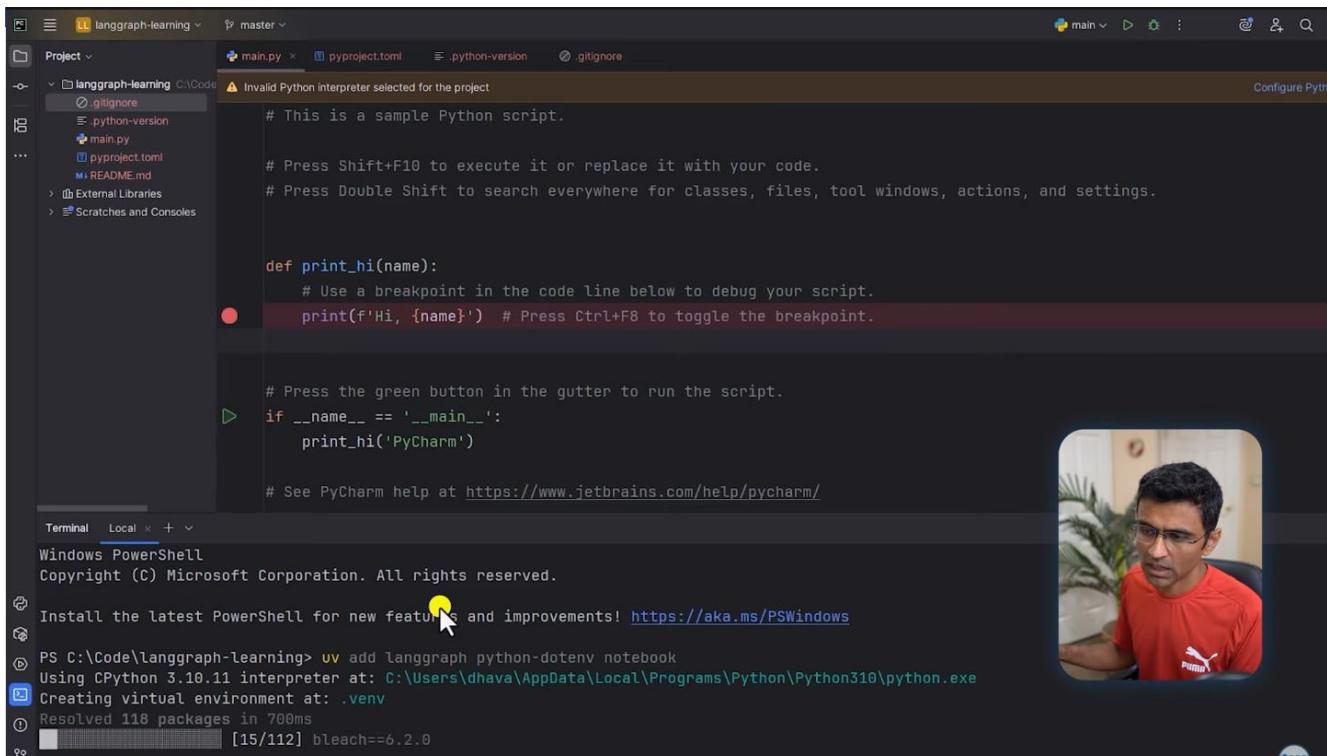
dhava@DP MINGW64 /c/Code/langgraph-learning (master)

```
[project]
name = "langgraph-learning"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.10"
dependencies = []
```

```
3.10
```

```
# Python-generated files
__pycache__/
*.py[oc]
build/
dist/
wheels/
*.egg-info

# Virtual environments
.venv/
```



```
# This is a sample Python script.

# Press Shift+F10 to execute it or replace it with your code.
# Press Double Shift to search everywhere for classes, files, tool windows, actions, and settings.

def print_hi(name):
    # Use a breakpoint in the code line below to debug your script.
    print(f'Hi, {name}') # Press Ctrl+F8 to toggle the breakpoint.

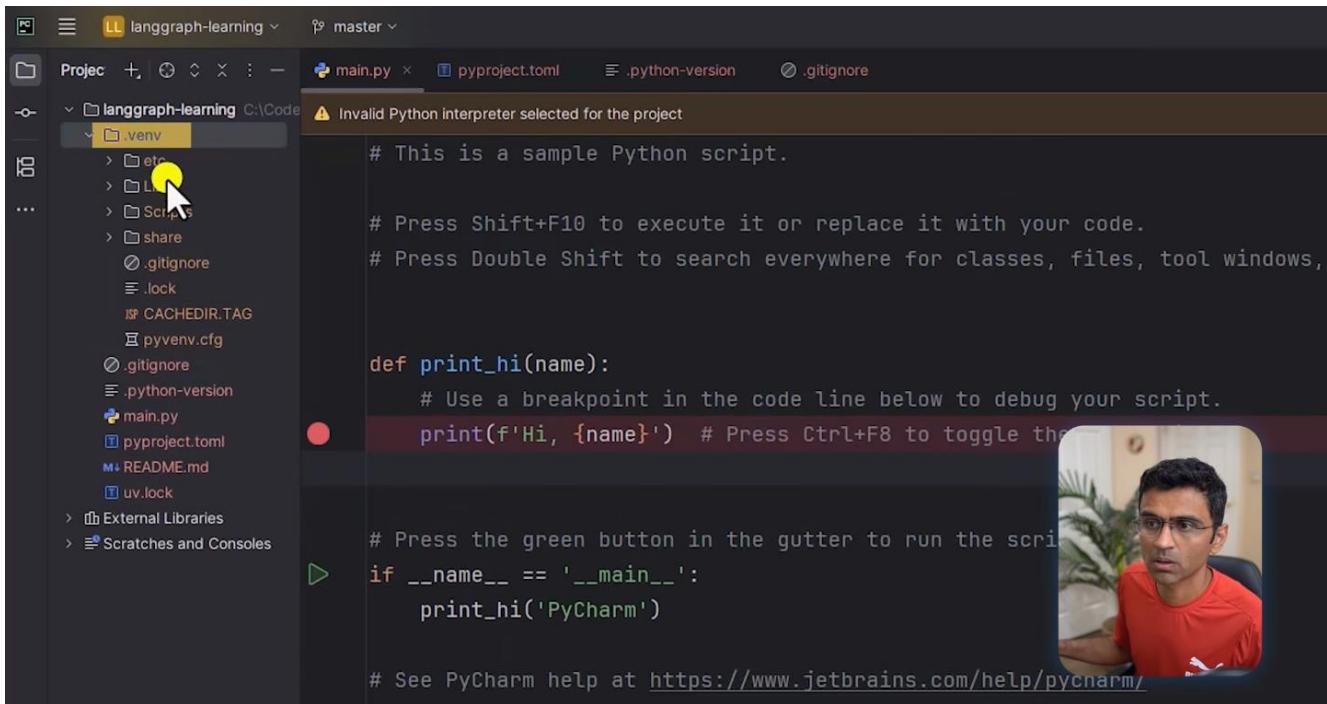
# Press the green button in the gutter to run the script.
if __name__ == '__main__':
    print_hi('PyCharm')

# See PyCharm help at https://www.jetbrains.com/help/pycharm/
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows
PS C:\Code\langgraph-learning> uv add langgraph python-dotenv notebook
Using CPython 3.10.11 interpreter at: C:\Users\dhava\AppData\Local\Programs\Python\Python310\python.exe
Creating virtual environment at: .venv
Resolved 118 packages in 700ms
[15/112] bleach==6.2.0
```

Next, we need to create a virtual environment



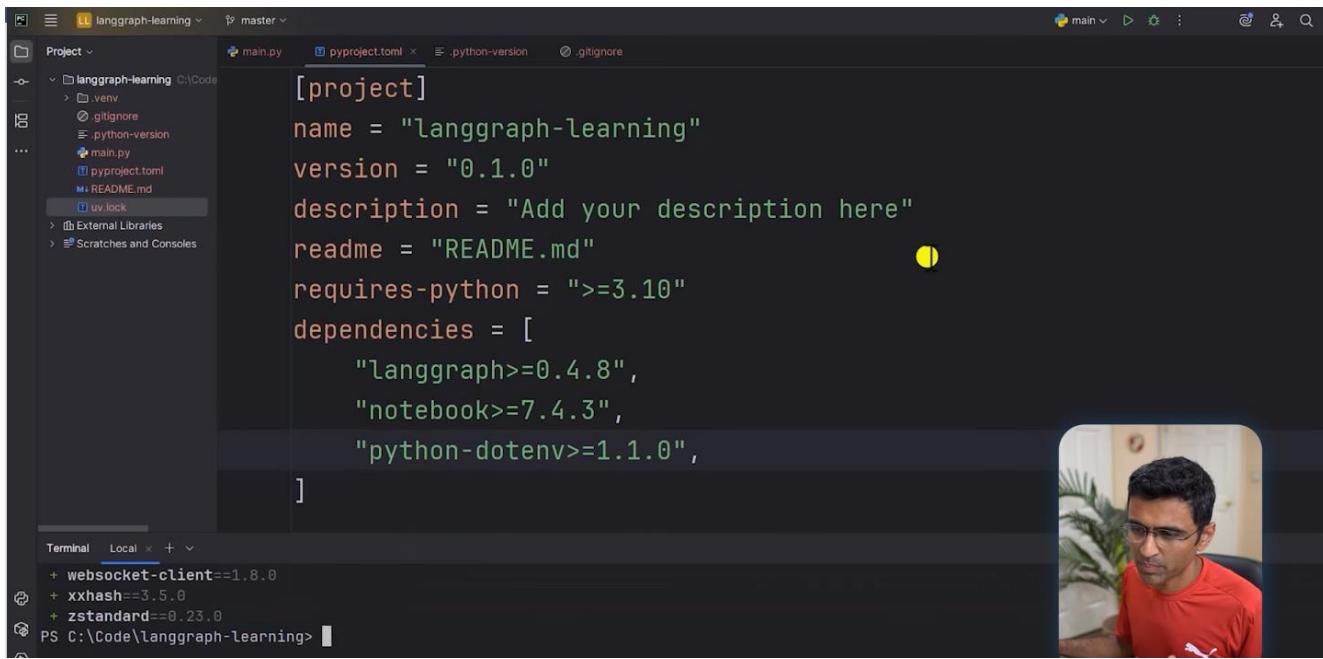
```
# This is a sample Python script.

# Press Shift+F10 to execute it or replace it with your code.
# Press Double Shift to search everywhere for classes, files, tool windows,
# actions, and settings.

def print_hi(name):
    # Use a breakpoint in the code line below to debug your script.
    print(f'Hi, {name}') # Press Ctrl+F8 to toggle the breakpoint.

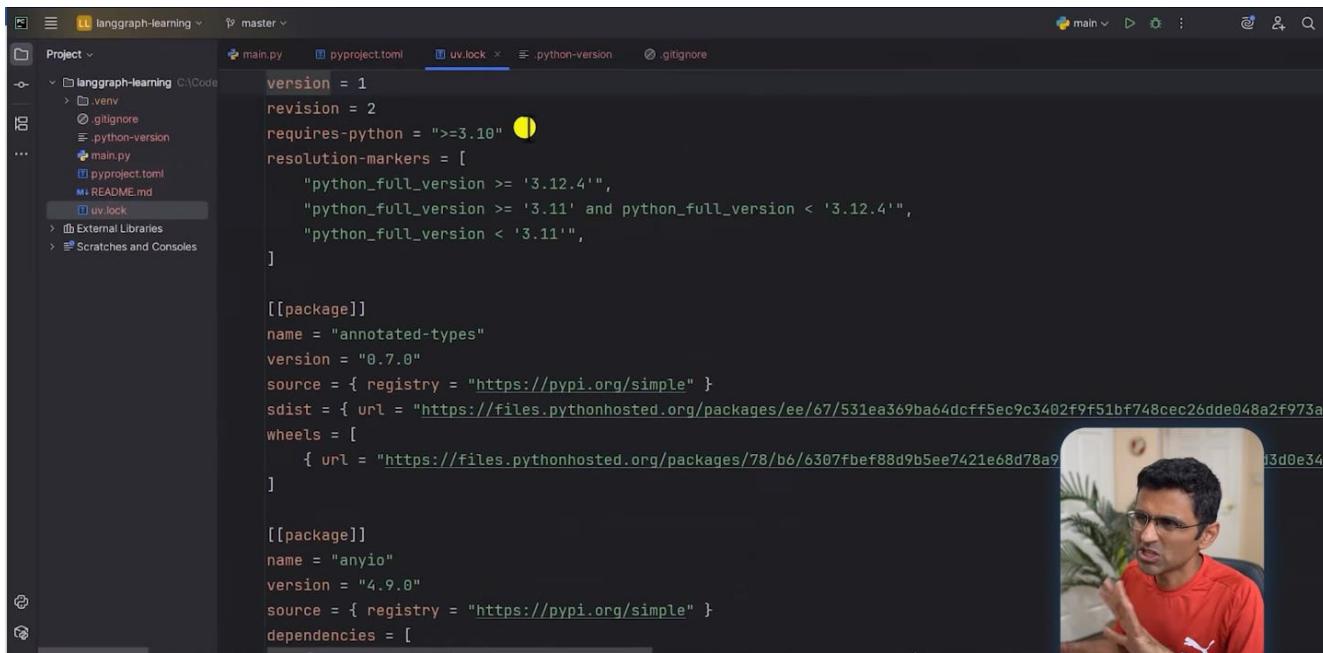
# Press the green button in the gutter to run the script.
if __name__ == '__main__':
    print_hi('PyCharm')

# See PyCharm help at https://www.jetbrains.com/help/pycharm/
```



```
[project]
name = "langgraph-learning"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.10"
dependencies = [
    "langgraph>=0.4.8",
    "notebook>=7.4.3",
    "python-dotenv>=1.1.0",
]
```

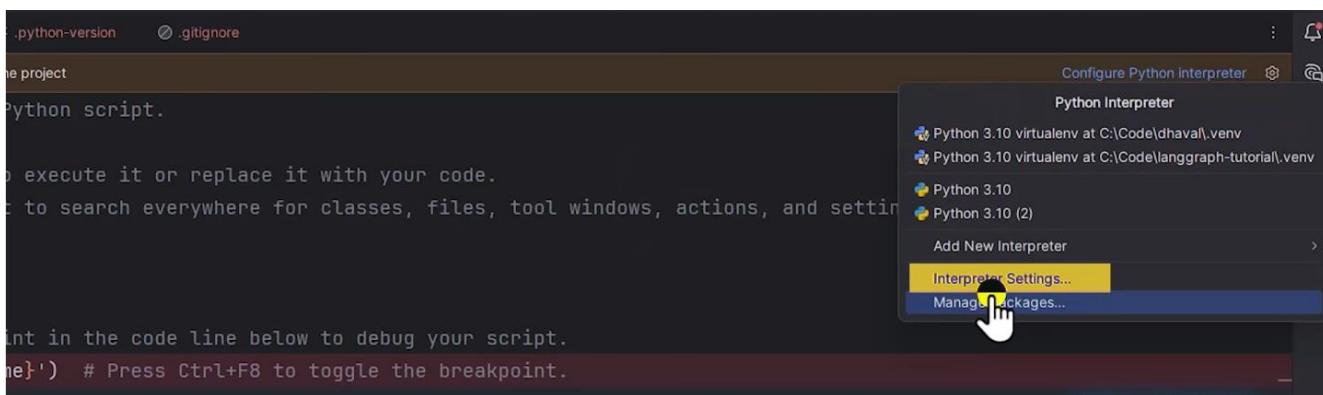
```
+ websocket-client==1.8.0
+ xxhash==3.5.0
+ zstandard==0.23.0
PS C:\Code\langgraph-learning>
```



```
version = 1
revision = 2
requires-python = ">=3.10"
resolution-markers = [
    "python_full_version >= '3.12.4'",
    "python_full_version >= '3.11' and python_full_version < '3.12.4'",
    "python_full_version < '3.11'",
]

[[package]]
name = "annotated-types"
version = "0.7.0"
source = { registry = "https://pypi.org/simple" }
sdist = { url = "https://files.pythonhosted.org/packages/ee/67/531ea369ba64dcffSec9c3402f9f51bf748cec26dde048a2f973a4" }
wheels = [
    { url = "https://files.pythonhosted.org/packages/78/b6/6307fbef88d9b5ee7421e68d78a9" }
]

[[package]]
name = "anyio"
version = "4.9.0"
source = { registry = "https://pypi.org/simple" }
dependencies = [
```



```
Python script.

execute it or replace it with your code.
to search everywhere for classes, files, tool windows, actions, and settings.

Int in the code line below to debug your script.
ne}') # Press Ctrl+F8 to toggle the breakpoint.
```

Configure Python interpreter

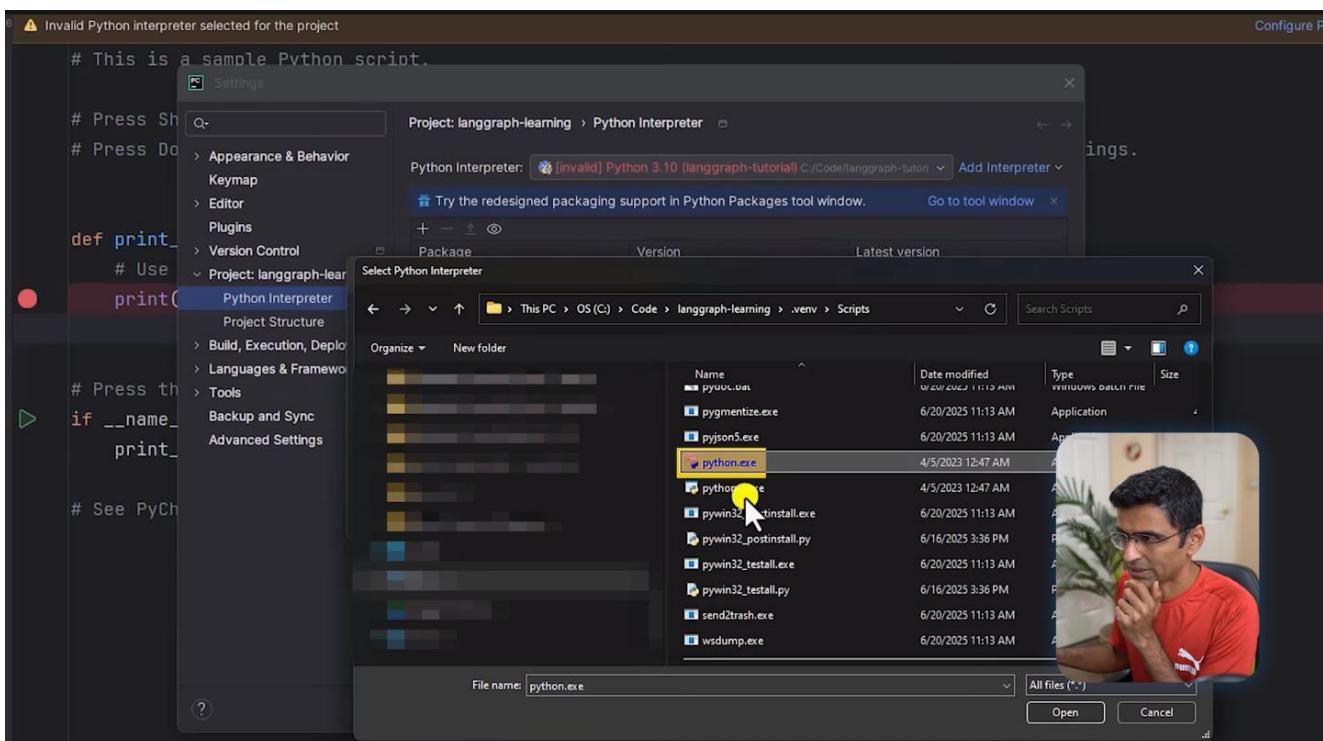
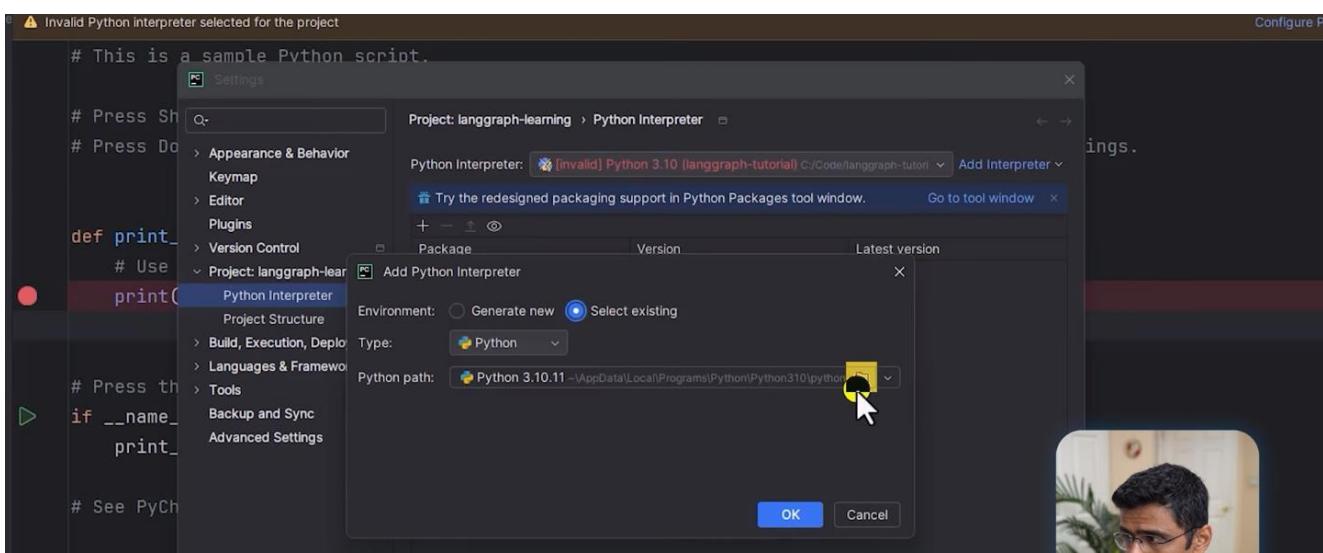
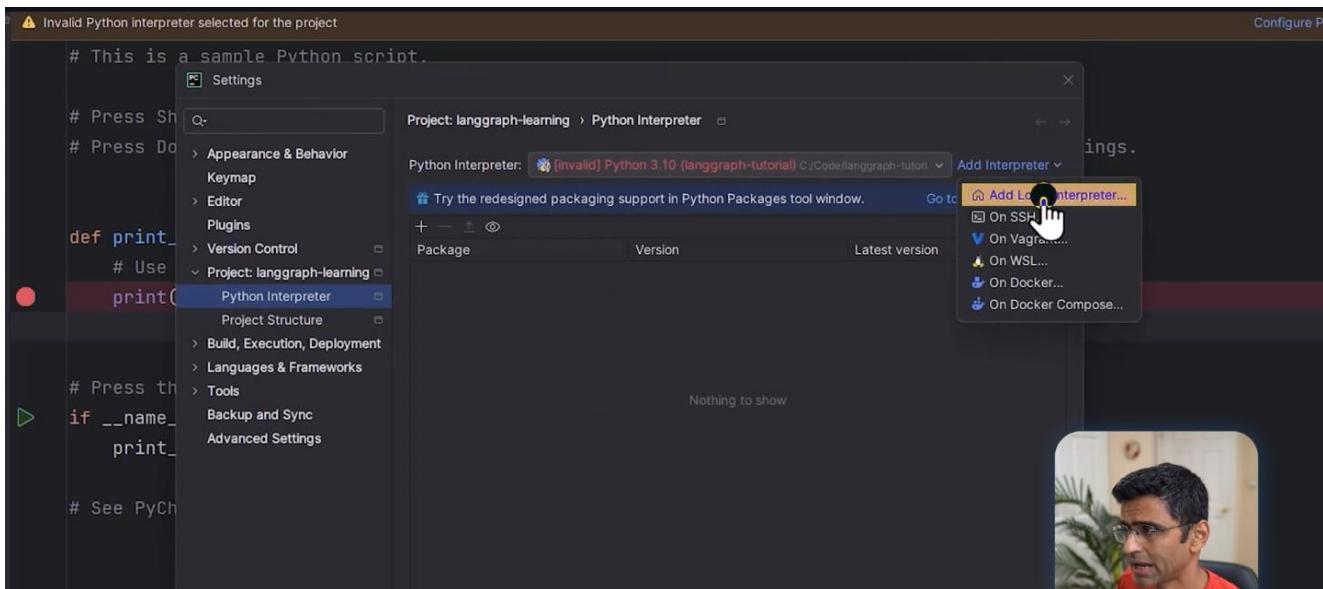
Python Interpreter

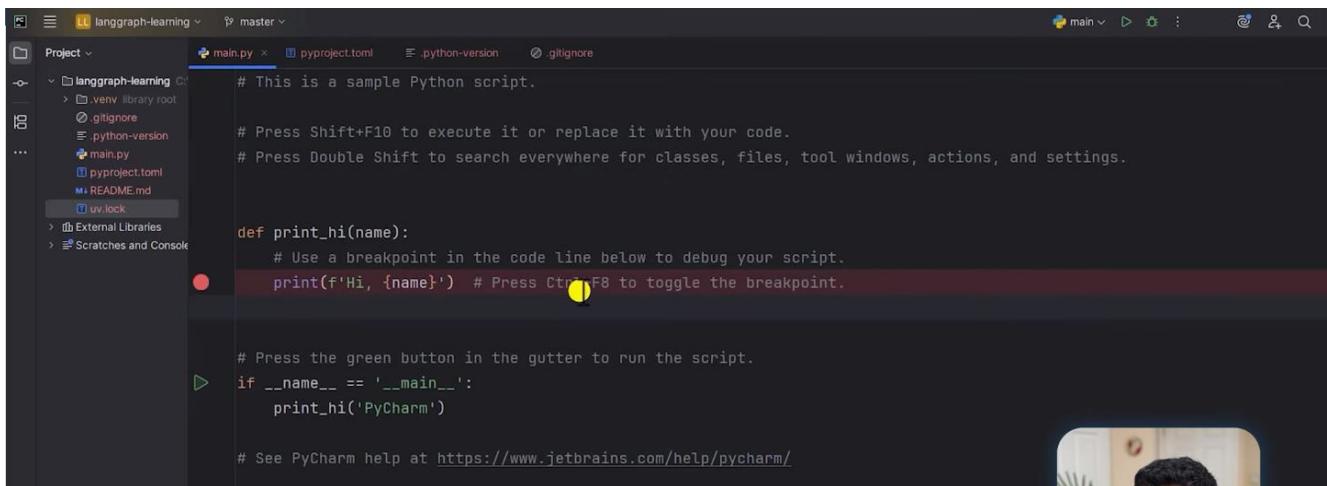
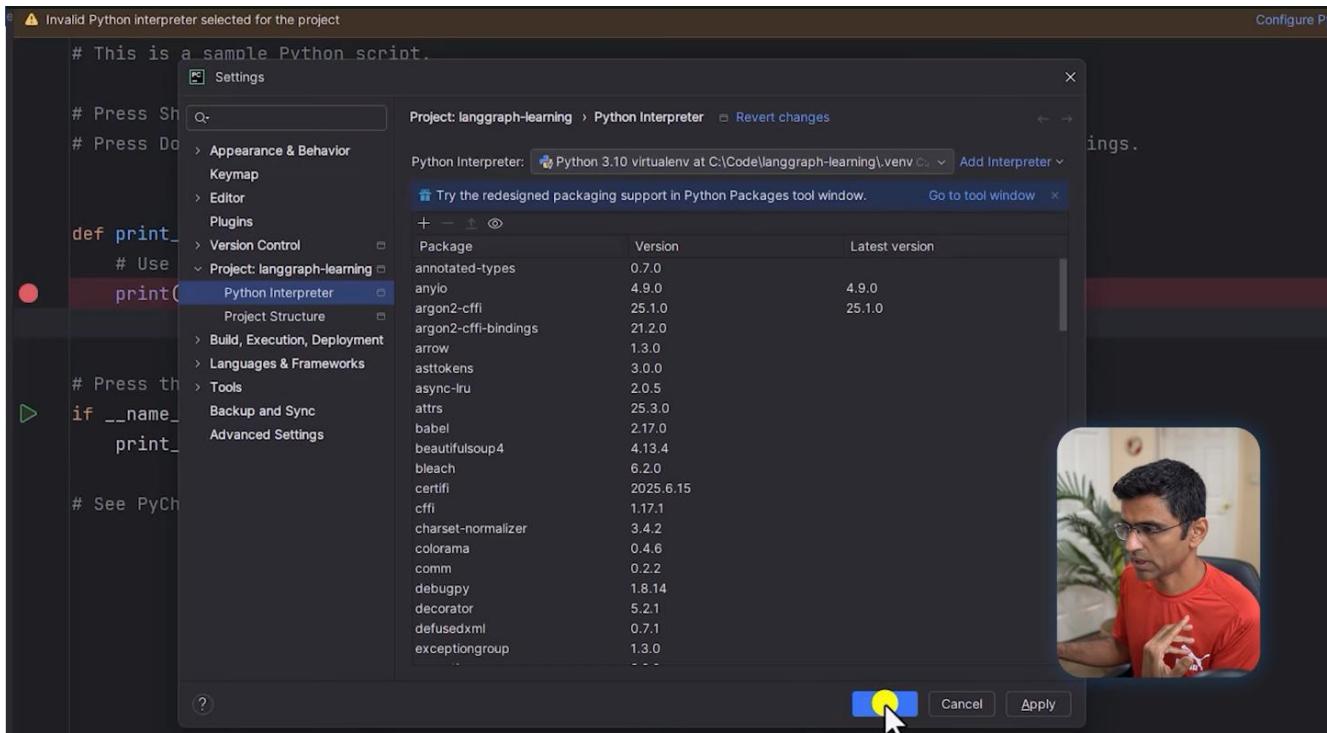
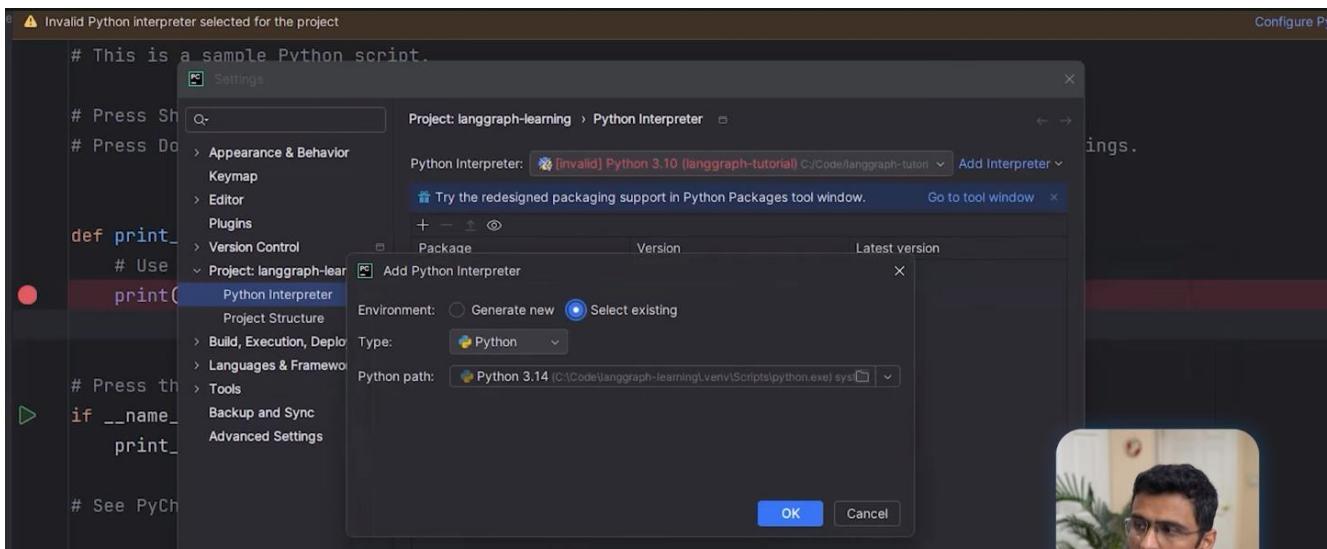
- Python 3.10 virtualenv at C:\Code\dhaval\.venv
- Python 3.10 virtualenv at C:\Code\langgraph-tutorial\.venv
- Python 3.10
- Python 3.10 (2)

Add New Interpreter

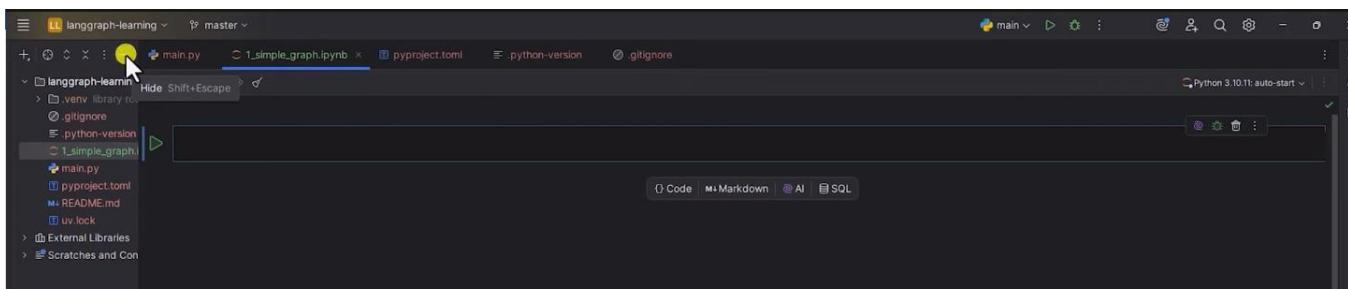
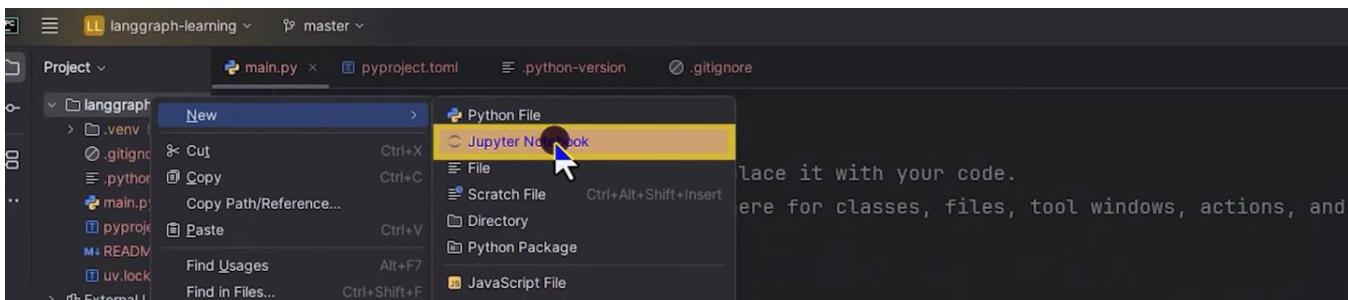
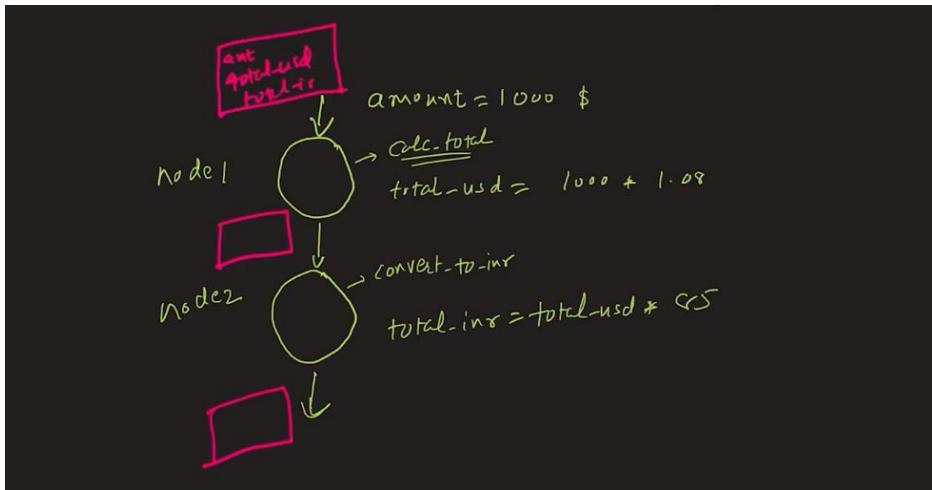
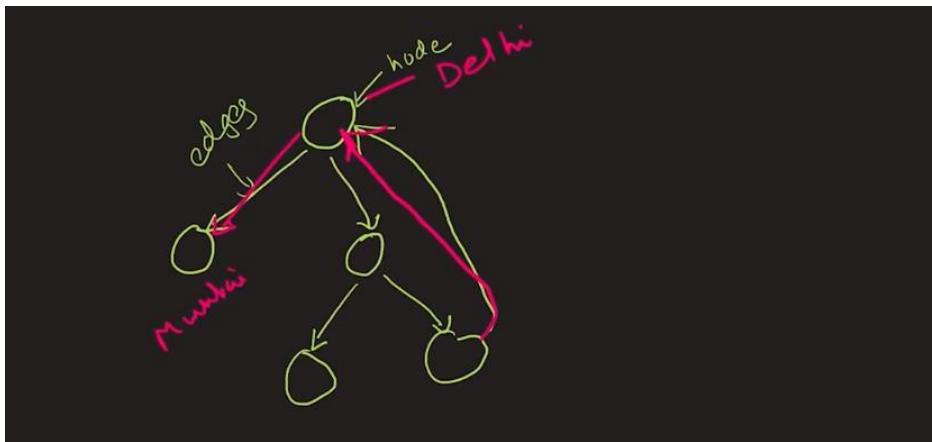
Interpreter Settings... (highlighted)

Manage Packages... (highlighted)





Now your environment is all set.



```

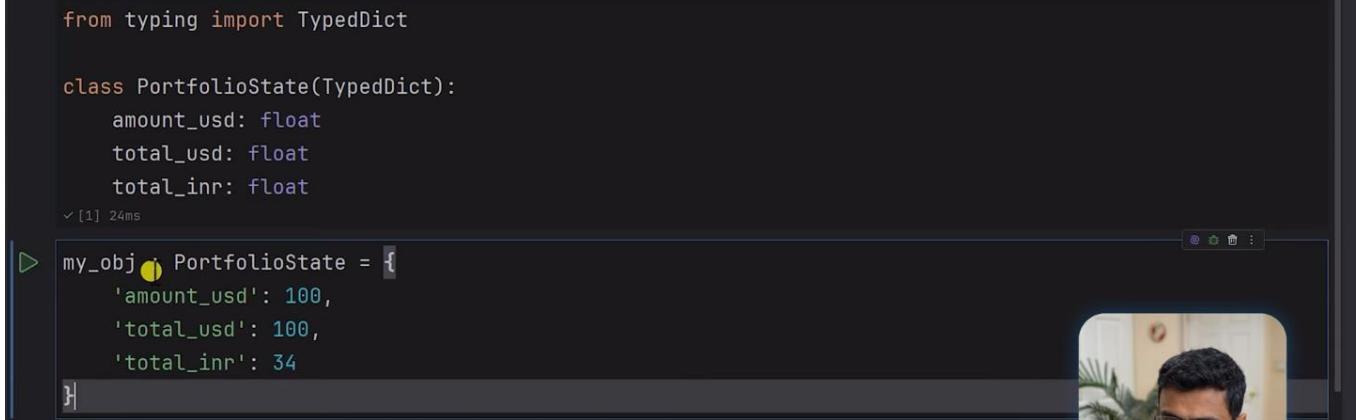
class PortfolioState():

```

```
from typing import TypedDict

class PortfolioState(TypedDict):
    amount_usd: float
    total_usd: float
    total_inr: float
✓ [1] 24ms
```

▶ my\_obj = PortfolioState({  
 'amount\_usd': 100,  
 'total\_usd': 100,  
 'total\_inr': 34  
})

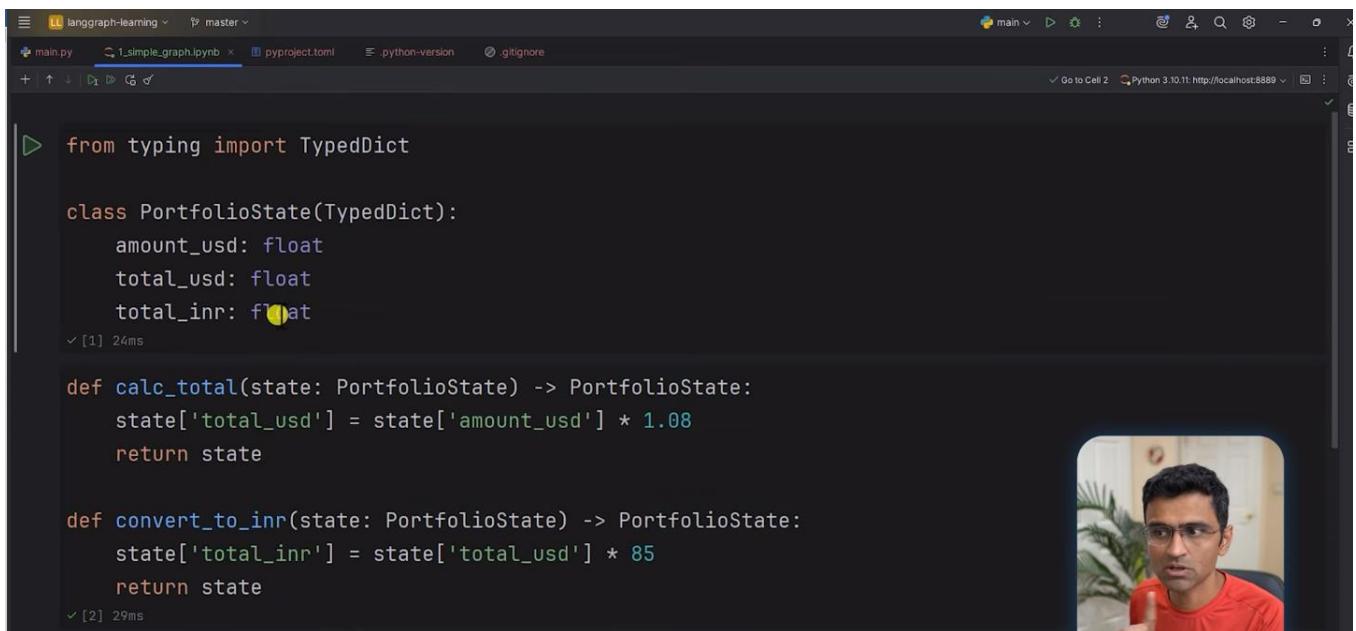


```
from typing import TypedDict

class PortfolioState(TypedDict):
    amount_usd: float
    total_usd: float
    total_inr: float
✓ [1] 24ms
```

```
def calc_total(state: PortfolioState) -> PortfolioState:  
    state['total_usd'] = state['amount_usd'] * 1.08  
    return state
```

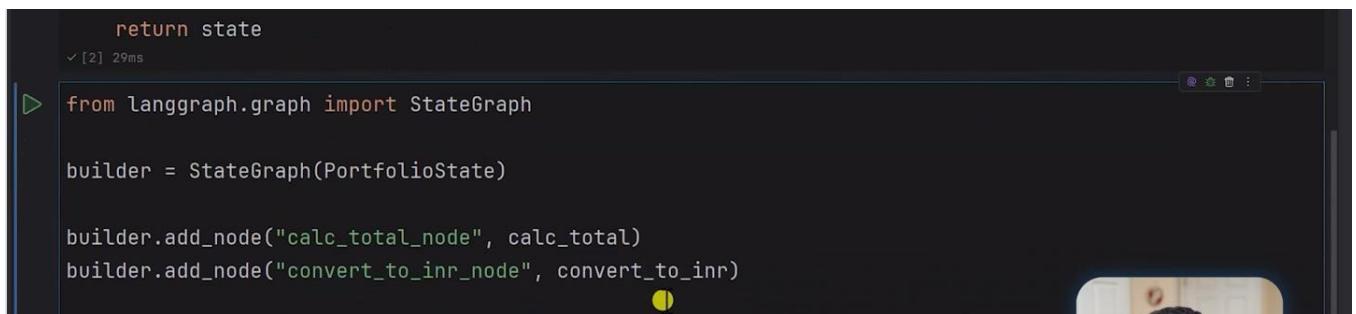
```
def convert_to_inr(state: PortfolioState) -> PortfolioState:  
    state['total_inr'] = state['total_usd'] * 85  
    return state
✓ [2] 29ms
```

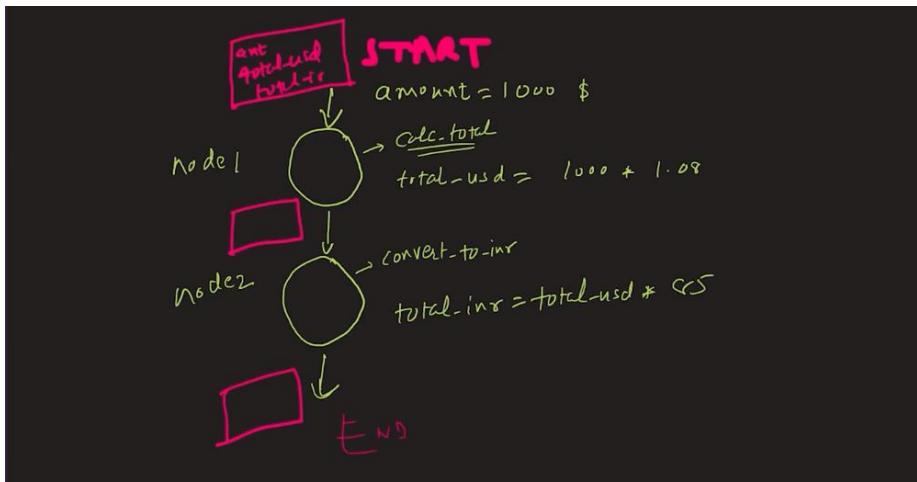


Next, we can define our graph as below

```
return state
✓ [2] 29ms
```

▶ from langgraph.graph import StateGraph  
  
builder = StateGraph(PortfolioState)  
  
builder.add\_node("calc\_total\_node", calc\_total)  
builder.add\_node("convert\_to\_inr\_node", convert\_to\_inr)





Next, we need to add the edges that define the relationships between the START and END edges.

```

from langgraph.graph import StateGraph, START, END

builder = StateGraph(PortfolioState)

builder.add_node("calc_total_node", calc_total)
builder.add_node("convert_to_inr_node", convert_to_inr)

builder.add_edge(START, "calc_total_node")
builder.add_edge("calc_total_node", "convert_to_inr_node")
builder.add_edge("convert_to_inr_node", END)

graph = builder.compile()

```

```

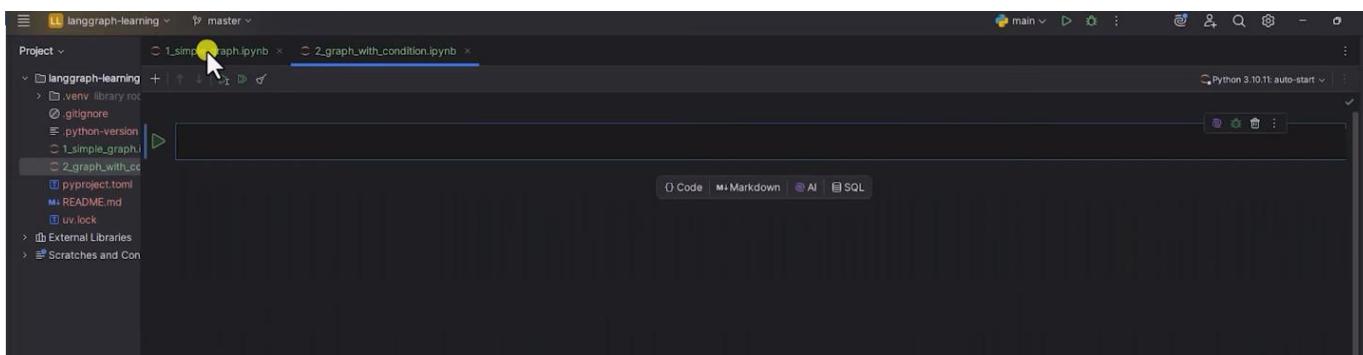
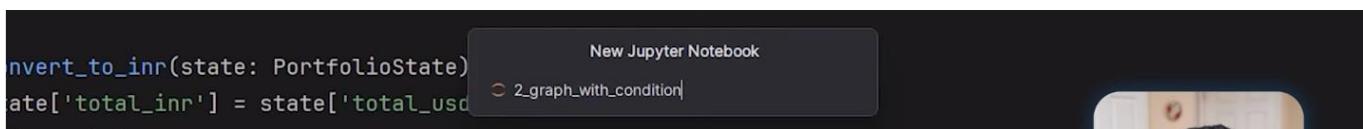
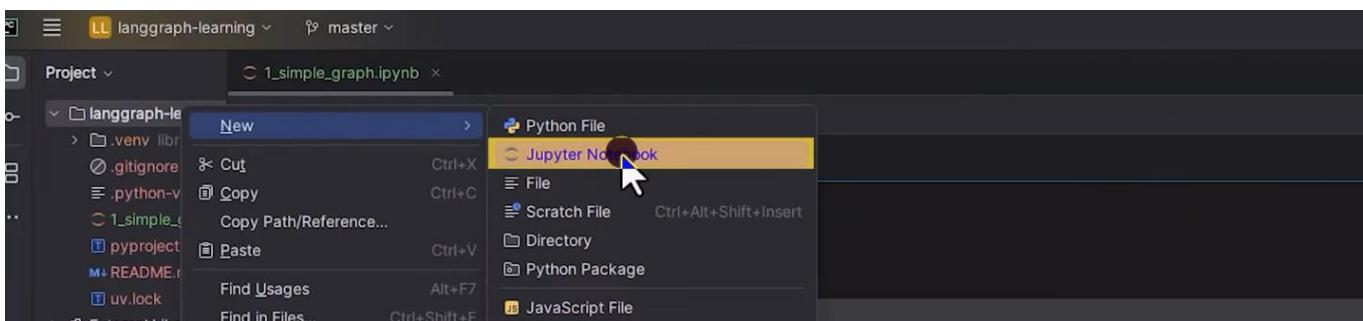
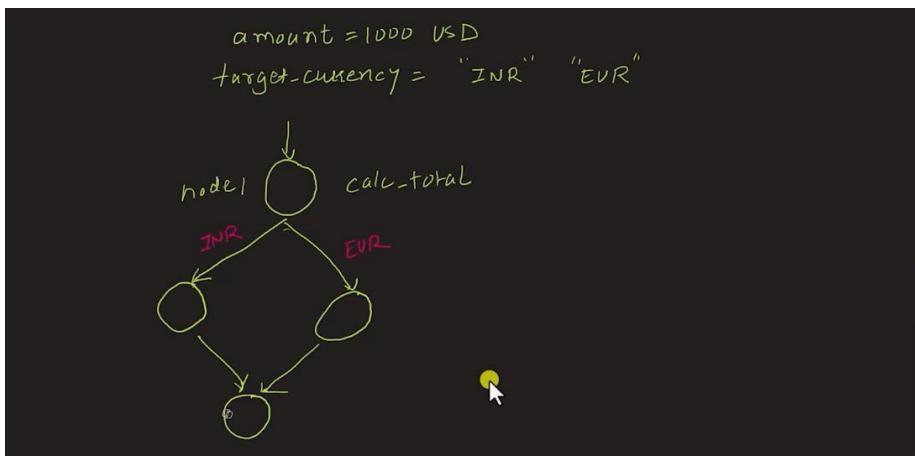
graph TD
    START((start)) --> calc_total_node[calc_total_node]
    calc_total_node --> convert_to_inr_node[convert_to_inr_node]
    convert_to_inr_node --> END((end))

```

convert\_to\_inr\_node  
\_end

```
graph.invoke({'amount_usd': 1000})| graph
✓ [15] 40ms
{'amount_usd': 1000, 'total_usd': 1080.0, 'total_inr': 91800.0}
```

Code Markdown AI SQL



```
From typing import TypedDict

class PortfolioState(TypedDict):
    amount_usd: float
    total_usd: float
    target_currency: str
    total: float
```

```
from typing import TypedDict, Literal

class PortfolioState(TypedDict):
    amount_usd: float
    total_usd: float
    target_currency: Literal["INR", "EUR"]
    total: float

[1] 15ms

def calc_total(state: PortfolioState) -> PortfolioState:
    state['total_usd'] = state['amount_usd'] * 1.08
    return state

def convert_to_inr(state: PortfolioState) -> PortfolioState:
    state['total'] = state['total_usd'] * 85
    return state

def convert_to_eur(state: PortfolioState) -> PortfolioState:
    state['total'] = state['total_usd'] * 0.9
    return state
```

```
def convert_to_eur(state: PortfolioState) -> PortfolioState:
    state['total'] = state['total_usd'] * 0.9
    return state

def choose_conversion(state: PortfolioState) -> str:
    return state["target_currency"]

[2] 11ms

from langgraph.graph import StateGraph, START, END

builder = StateGraph(PortfolioState)

builder.add_node("calc_total_node", calc_total)
builder.add_node("convert_to_inr_node", convert_to_inr)
builder.add_node("convert_to_eur_node", convert_to_eur)
```

Next, we add the edges

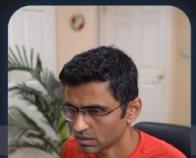
```
langraph-learning - master ~
1.simple_graph.ipynb 2.graph_with_condition.ipynb
+ ↑ ↓ ⌂ ⌂ builder.add_node("calc_total_node", calc_total_node)
builder.add_node("convert_to_inr_node", convert_to_inr)
builder.add_node("convert_to_eur_node", convert_to_eur)

builder.add_edge(START, "calc_total_node")
builder.add_conditional_edges(
    "calc_total_node",
    choose_conversion,
{
    "INR": "convert_to_inr_node",
    "EUR": "convert_to_eur_node",
}
)
builder.add_edge("convert_to_inr_node", END)
builder.add_edge("convert_to_eur_node", END)|
```



```
langraph-learning - master ~
1.simple_graph.ipynb 2.graph_with_condition.ipynb
+ ↑ ↓ ⌂ ⌂ builder.add_node("calc_total_node", calc_total_node)
builder.add_node("convert_to_inr_node", convert_to_inr)
builder.add_node("convert_to_eur_node", convert_to_eur)

builder.add_edge(START, "calc_total_node")
builder.add_conditional_edges(
    "calc_total_node",
    choose_conversion,
{
    "INR": "convert_to_inr_node",
    "EUR": "convert_to_eur_node",
}
)
builder.add_edge(["convert_to_inr_node", "convert_to_eur_node"], END)
graph = builder.compile()|
```

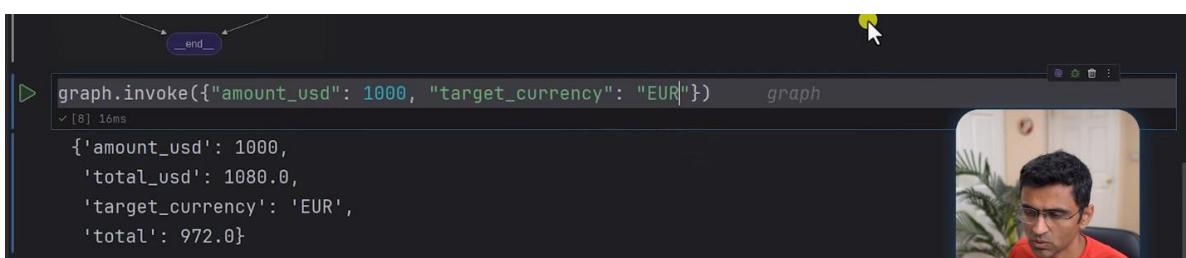
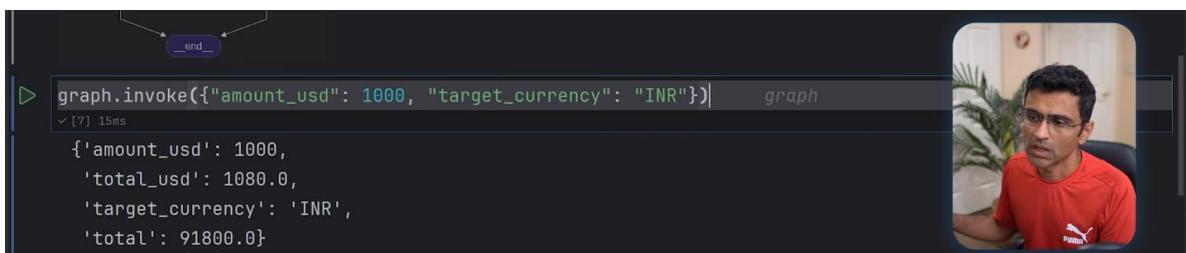
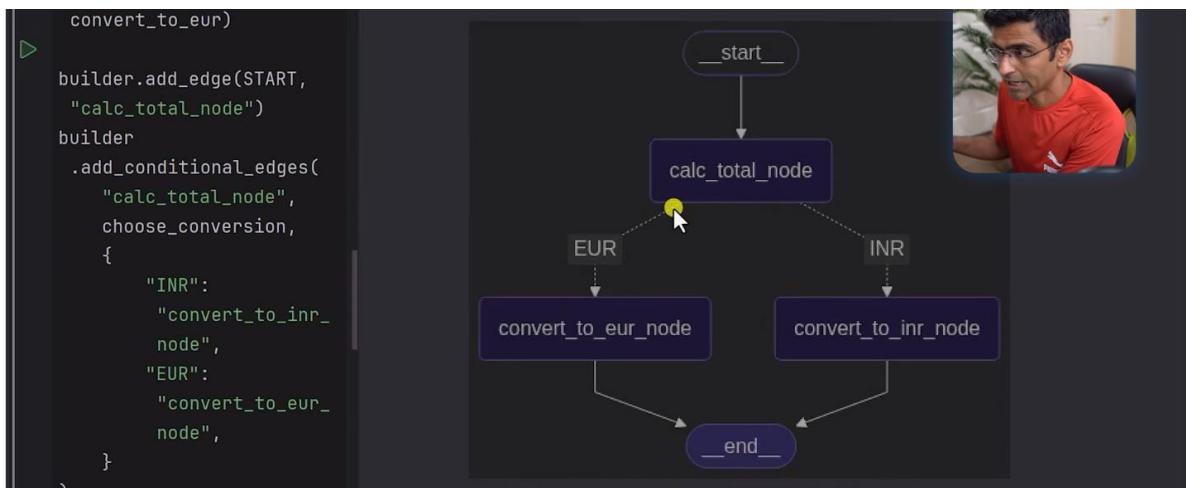


```
graph = builder.compile()
✓ [5] 16ms
from IPython.display import Image, display

display(Image(graph.get_graph().draw_mermaid_png()))| graph
✓ [6] 742ms
```

```
graph TD; start((start)) --> calcTotalNode[calc_total_node]; calcTotalNode -- EUR --> convertToEurNode[convert_to_eur_node]; calcTotalNode -- INR --> convertToInrNode[convert_to_inr_node]; convertToEurNode --> end((end)); convertToInrNode --> end;
```





It is now correctly going through the correct conditional path based on the selected output currency.

The screenshot shows the Google AI Studio interface. At the top, there are navigation links: `Get API key`, `Studio`, and `Dashboard`. Below this, there is a sidebar with links: `API Keys` (which is currently selected), `Usage & Billing`, and `Changelog`. The main content area is titled `API Keys` and contains a section titled `Quickly test the Gemini API` with a yellow circular button for testing. Below this is a section titled `API quickstart guide` containing a curl command:

```

curl "https://generativelanguage.googleapis.com/v1beta/models/gemini-2.0-flash:generateContent?key=GEMINI_API_KEY" \
-H 'Content-Type: application/json' \
-X POST \
-d '{
  "contents": [
    {
      "parts": [
        {
          "text": "Explain how AI works in a few words"
        }
      ]
    }
}'

```

At the bottom of the main content area, it says: `Your API keys are listed below. You can also view and manage your project and API keys in Google Cloud.`

Next, let us create a chatbot using Gemini

Google Cloud google maps project

Search (/) for resources, docs, products, and more Search

Welcome

You're working in [google maps project](#)

Project number: 106713121410 Project ID: savvy-scion-447817-f3

Dashboard [Cloud Hub](#) New

**Create a VM** Streamline cloud operations with centralized insights for your projects and applications. Monitor key metrics for performance, costs, issues, and support cases. Deploy an application

**Create a storage bucket** Learn more

Quick access

[API APIs & Services](#) [IAM & Admin](#) [Billing](#)

[Cloud Storage](#) [BigQuery](#) [VPC network](#)

[View all products](#)

Try Gemini 2.0 Flash [Try Gemini](#)

A yellow circle highlights the 'google maps project' link in the top navigation bar.

Welcome

You're working in [google maps project](#)

Project number: 106713121410

Dashboard [Cloud Hub](#) New

**Create a VM** **Create a storage bucket**

Recent Starred All

Name	Type	ID	Star	Action
<a href="#">google maps project</a>	Project	savvy-scion-447817-f3	★	<a href="#">Gemini 2.0 Flash</a>
<a href="#">MCP Server</a>	Project	mcp-server-457202	★	<a href="#">Gemini</a>
<a href="#">AI Agents Tutorials</a>	Project	ai-agents-tutorials	★	<a href="#">Gemini</a>
<a href="#">Generative Language Client</a>	Project	gen-lang-client-0534822434	★	<a href="#">Gemini</a>
<a href="#">Potato disease classification</a>	Project	potato-disease-classification	★	<a href="#">Gemini</a>
<a href="#">My First Project</a>	Project	mineral-proton-324215	★	<a href="#">Gemini</a>

New project

A yellow circle highlights the 'New project' button in the 'Select a project' modal.

Welcome

You're working in [google maps project](#)

Project number: 106713121410

Dashboard [Cloud Hub](#) New

**Create a VM** **Create a storage bucket**

Recent Starred All

Name	Type	ID	Star	Action
<a href="#">google maps project</a>	Project	savvy-scion-447817-f3	★	<a href="#">Gemini 2.0 Flash</a>
<a href="#">MCP Server</a>	Project	mcp-server-457202	★	<a href="#">Gemini</a>
<a href="#">AI Agents Tutorials</a>	Project	ai-agents-tutorials	★	<a href="#">Gemini</a>
<a href="#">Generative Language Client</a>	Project	gen-lang-client-0534822434	★	<a href="#">Gemini</a>
<a href="#">Potato disease classification</a>	Project	potato-disease-classification	★	<a href="#">Gemini</a>
<a href="#">My First Project</a>	Project	mineral-proton-324215	★	<a href="#">Gemini</a>

New project

A yellow circle highlights the 'AI Agents Tutorials' project row in the 'Select a project' modal.

Google AI Studio

Get API key Studio Dashboard

API Keys Usage & Billing Changelog

## API Keys

Quickly test the Gemini API

API quickstart guide

```
curl "https://generativelang...
```

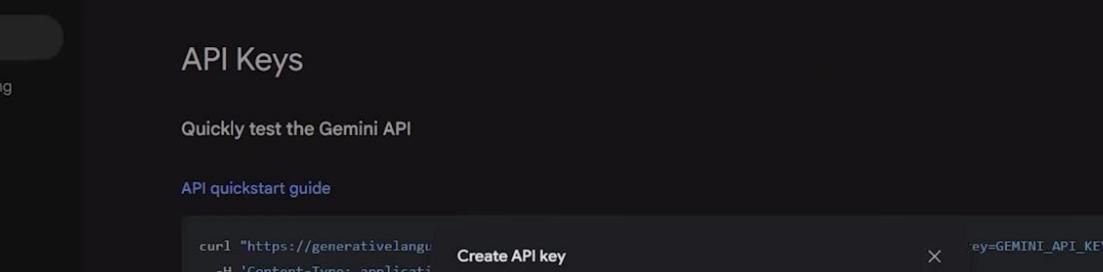
Create API key

Select a project from your existing Google Cloud projects

Search Google Cloud projects

- MCP Server mcp-server-457202
- AI Agents Tutorials ai-agents-tutorials
- google maps project savvy-scion-447817-f3
- Generative Language Client gen-lang-client-0534822434
- Generative Language Client gen-lang-client-0187384934

Use code with caution.



API quickstart guide

```
curl "https://generativelangu...ey=GEMINI_API_KEY" \
```

Create API key

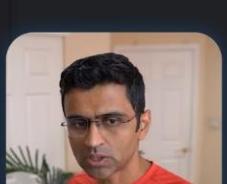
Select a project from your existing Google Cloud projects

Search Google Cloud projects

AI Agents Tutorials ai-agents-tutorials

⟳ Create API key in existing project

Use code with caution.



Your API keys are listed below. You can also view and manage your project and API keys in Google Cloud.



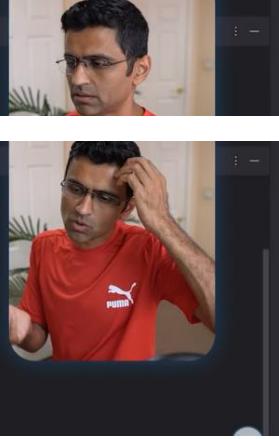
A screenshot of the Visual Studio Code interface. The top bar shows the project name "langgraph-learning" and the branch "master". The left sidebar shows a tree view of the project structure under "Project": "langgraph-learning" (with a sub-item ".venv library root"), ".env", ".gitignore", and ".python-version". The main editor area has three tabs: "1\_simple\_graph.ipynb" (highlighted), ".env" (selected), and "2\_graph\_with\_condition.ipynb". The ".env" tab contains the environment variable "GOOGLE\_API\_KEY=AI".

The screenshot shows a Jupyter Notebook interface with the following details:

- Project:** langgraph-learning
- File List:** 1\_simple\_graph.ipynb, 3\_chatbot.ipynb, 2\_graph\_with\_condition.ipynb
- Cells:** The current cell contains the following Python code:

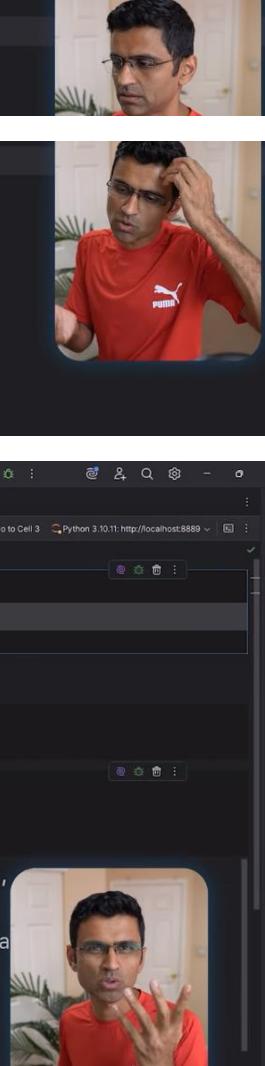
```
from dotenv import load_dotenv
load_dotenv()
```
- Output:** The output of the cell is "True".
- Environment:** Python 3.10.11: http://localhost:8888

Create a new project file as above



Terminal Local x + v  
(langgraph-learning) PS C:\Code\langgraph-learning> uv add langchain langchain-google-genai  
Resolved 137 packages in 388ms

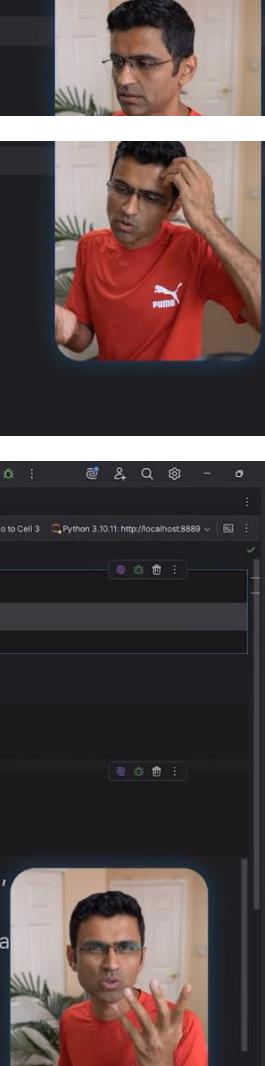
Terminal Local x + v  
+ google-ai-generative-language==0.6.18  
+ google-api-core==2.25.1  
+ google-auth==2.40.3  
+ googleapis-common-protos==1.70.0  
+ grpcio==1.73.0  
+ grpcio-status==1.73.0  
+ langchain-google-genai==2.1.5  
+ proto-plus==1.26.1  
+ protobuf==6.31.1  
+ pyasn1==0.6.1  
+ pyasn1-modules==0.4.2  
+ rsa==4.9.1  
(langgraph-learning) PS C:\Code\langgraph-learning>



langgraph-learning master  
1\_simple\_graph.ipynb 3\_chatbot.ipynb 2\_graph\_with\_condition.ipynb  
+ | - | D G ⌂

```
> from dotenv import load_dotenv  
| load_dotenv()  
| ✓ [1] 24ms  
| True  
  
> from langchain.chat_models import init_chat_model  
| ✓ [2] 5s 76ms  
  
> llm = init_chat_model("google_genai:gemini-2.0-flash")  
llm.invoke("Who was the first person to walk on moon?")  
| ✓ [3] 2s 142ms  
| AIMessage(content='Neil Armstrong was the first person to walk on the Moon.',  
| additional_kwargs={}, response_metadata={'prompt_feedback': {'block_reason':  
| 'safety_ratings': []}, 'finish_reason': 'STOP', 'model_name': 'gemini-2.0-fla  
| 'safety_ratings': []}, id='run--c975ccdb-6c2d-494b-9680-f6144db9fa83-0',  
| usage_metadata={'input_tokens': 10, 'output_tokens': 13, 'total_tokens': 23,  
| 'input_token_details': {'cache_read': 0}})
```

The Gemini LLM test works, let's now create our stategraph as below but we add new libraries first



langgraph-learning master  
1\_simple\_graph.ipynb 3\_chatbot.ipynb 2\_graph\_with\_condition.ipynb  
+ | - | D G ⌂

```
> from dotenv import load_dotenv  
load_dotenv()  
  
> from typing_extensions import TypedDict  
from langchain.chat_models import init_chat_model  
from langgraph.graph import StateGraph, START, END  
from langgraph.graph.message import add_messages  
| ✘ 550ms  
  
> from langchain.chat_models import init_chat_model
```

```
from langgraph.graph.message import add_messages
✓ [4] 939ms

from langchain.chat_models import init_chat_model
✓ [2] 5s 76ms

▶ llm = init_chat_model("google_genai:gemini-2.0-flash")

class State(TypedDict):
    messages: list

def chatbot(state: State) -> State:
    return {"messages": [llm.invoke(state["messages"])]}

builder = StateGraph(State)
builder.add_node("chatbot_node", chatbot)

builder.add_edge(START, "chatbot_node")
builder.add_edge("chatbot_node", END)

graph = builder.compile()
```



```
graph = builder.compile()
✓ [5] 23ms

▶ message = {"role": "user", "content": "Who walked on the moon for the first time? Print only the name"}
message
response = graph.invoke({"messages": [message]})

response["messages"]
✓ [6] 434ms

[AIMessage(content='Neil Armstrong', additional_kwargs={}, response_metadata={'prompt_feedback': {'block_reason': 0, 'safety_ratings': []}, 'finish_reason': 'STOP', 'model_name': 'gemini-2.0-flash', 'safety_ratings': []}, id='run--e85cde28-6a77-4f8e-bc0e-8acf7d28f82c-0', usage_metadata={'input_tokens': 14, 'output_tokens': 3, 'total_tokens': 17, 'input_token_details': {'cache_read': 0}})]
```

Next, we want to preserve the messages being added over the conversation

langgraph-learning master

1\_simple\_graph.ipynb 3\_chatbot.ipynb 2\_graph\_with\_condition.ipynb

Go to Cell 3 Python 3.10.11 http://localhost:8889

```
from typing import Annotated
from dotenv import load_dotenv
load_dotenv()

from typing_extensions import TypedDict
from langchain.chat_models import init_chat_model
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages
✓ [4] 939ms

from langchain.chat_models import init_chat_model
✓ [2] 5s 76ms

llm = init_chat_model("google_genai:gemini-2.0-flash")

class State(TypedDict):
    messages: Annotated[list, add_messages]

def chatbot(state: State) -> State:
    return {"messages": [llm.invoke(state["messages"])]}

builder = StateGraph(State)
builder.add_node("chatbot_node", chatbot)
```



```

builder = StateGraph(State)
builder.add_node("chatbot_node", chatbot)

builder.add_edge(START, "chatbot_node")
builder.add_edge("chatbot_node", END)

graph = builder.compile()
✓ [10] 20ms

message = {"role": "user", "content": "Who walked on the moon for the first time? Print only the name"}
response = graph.invoke([{"messages": [message]}])      response      graph      message

response["messages"]
✓ [11] 356ms
[HumanMessage(content='Who walked on the moon for the first time? Print only the name', additional_
response_metadata={}, id='e97b27be-9c23-45c7-bf0f-80c134626319'),
 AIMessage(content='Neil Armstrong', additional_kwarg..., response_metadata={'prompt_feedback': {'safety_ratings': []}, 'finish_reason': 'STOP', 'model_name': 'gemini-2.0-flash', 'safety_ratings': id='run--aa55e1f2-68c5-45c0-8cdc-3abb3091e044-0', usage_metadata={'input_tokens': 14, 'output_toke
 17, 'input_token_details': {'cache_read': 0}}}]

```

We can now see both HumanMessage and AIMessage in the `response["messages"]` list

GitHub Pages  
https://langchain-ai.github.io/how-tos/state-reducers ::

## How to update graph state from nodes

LangGraph includes a built-in reducer `add_messages` that handles these considerations: from `langgraph.graph.message import add_messages ...`

Missing: `add_message` | Show results with: `add_message`

Medium · HARSHA JS · 5 months ago

```

from langgraph.graph import StateGraph

builder = StateGraph(State)
builder.add_node(node)
builder.set_entry_point("node")
graph = builder.compile()

```

LangGraph provides built-in utilities for visualizing your graph. Let's inspect our graph. See [this section](#) for detail on visualization.

```

from IPython.display import Image, display

display(Image(graph.get_graph().draw_mermaid_png()))

```

```

graph TD
    start((__start__)) --> node((node))

```

Reducers are key to understanding how updates from nodes are applied to the `State`. Each key in the `State` has its own independent reducer function. If no reducer function is explicitly specified then it is assumed that all updates to that key should override it. There are a few different types of reducers, starting with the default type of reducer:

## Default Reducer

These two examples show how to use the default reducer:

### Example A:

```
from typing_extensions import TypedDict

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

In this example, no reducer functions are specified for any key. Let's assume the input to the graph is `1, "bar": ["hi"]`. Let's then assume the first Node returns `{"foo": 2}`. This is treated as an update to the state. Notice that the Node does not need to return the whole State schema - just an update. After applying this update, the State would then be `{"foo": 2, "bar": ["hi"]}`. If the second node returns `{"bar":`



Table of contents
Graphs
StateGraph
Compiling your
State
Schema
Multiple schema
Reducers
Default Reducers
Working with State
use me
Message Graph
Initialization
MessagesState
Nodes
START Node

## Working with Messages in Graph State

### Why use messages?

Most modern LLM providers have a chat model interface that accepts a list of messages as input. LangChain's `ChatModel` in particular accepts a list of `Message` objects as inputs. These messages come in a variety of forms such as `HumanMessage` (user input) or `AIMessage` (LLM response). To read more about what message objects are, please refer to [this](#) conceptual guide.

### Using Messages in your Graph

In many cases, it is helpful to store prior conversation history as a list of messages in your graph state. So, we can add a key (channel) to the graph state that stores a list of `Message` objects and annotate it with a reducer function (see `messages` key in the example below). The reducer function is vital to telling the graph how to update the list of `Message` objects in the state with each state update (for example, when a node provides an update). If you don't specify a reducer, every state update will overwrite the list of messages with the recently provided value. If you wanted to simply append messages to the existing list, you could use `operator.add` as a reducer.



Table of contents
Graphs
StateGraph
Compiling your
State
Schema
Multiple schema
Reducers
Default Reducers
Working with State
use me
Message Graph
Initialization
MessagesState
Nodes
START Node

### MessagesState

[Back to top](#)

Since having a list of messages in your state is so common, there exists a prebuilt state called `MessagesState` which makes it easy to use messages. `MessagesState` is defined with a single `messages` key which is a list of `AnyMessage` objects and uses the `add_messages` reducer. Typically, there is more state to track than just messages, so we see people subclass this state and add more fields, like:

```
from langgraph.graph import MessagesState

class State(MessagesState):
    documents: list[str]
```

## Nodes

In LangGraph, nodes are typically python functions (sync or async) where the **first** positional argument is `state`, and (optionally), the **second** positional argument is a "config", containing optional `configurable parameters` (such as a `thread_id`).

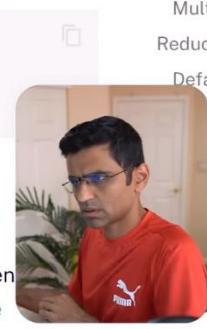


Table of contents
Graphs
StateGraph
Compiling your
State
Schema
Multiple schema
Reducers
Default Reducers
Working with State
use me
Message Graph
Initialization
MessagesState
Nodes
START Node

into LangChain Message objects whenever a state update is received on the `messages` channel. See more information on LangChain serialization/deserialization [here](#). This allows sending graph inputs / state updates in the following format:

```
# this is supported
{"messages": [HumanMessage(content="message")]}

# and this is also supported
{"messages": [{"type": "human", "content": "message"}]}
```

Since the state updates are always deserialized into LangChain Messages when using `add_messages`, you should use dot notation to access message attributes, like `state["messages"][-1].content`. Below is an example of a graph that uses `add_messages` as its reducer function.

API Reference: [AnyMessage / add\\_messages](#)

```
from langchain_core.messages import AnyMessage
from langgraph.graph.message import add_messages
from typing import Annotated
from typing_extensions import TypedDict

class GraphState(TypedDict):
```

Table of conten
Graphs
StateGraph
Compiling yo
State
Schema
Multiple sch
Reducers
Default Red
ng with
State
use me
g Mess
ph
alizati
agesSi
Nodes
START Node



```
+ 1_simple_graph.ipynb  3_chatbot.ipynb  2_graph_with_condition.ipynb
+ ↑ ↓ ⌂ G ↵
  Safety_ratings : [], TURNTURN_REASON : STOP , model_name : gpt-3.5-turbo , Safety_ratings : [], id='run--aa55e1f2-68c5-45c0-8cdc-3abb3091e044-0' , usage_metadata={'input_tokens': 14, 'output_tokens': 3, 'total_tokens': 17, 'input_token_details': {'cache_read': 0}}]

state = None
while True:
    in_message = input("You: ")
    if in_message.lower() in {"quit", "exit"}:
        break
    if state is None:
        state: State = {
            "messages": [{"role": "user", "content": in_message}]
        }
    else:
        state["messages"].append({"role": "user", "content": in_message})

    state = graph.invoke(state)
    print("Bot:", state["messages"][-1].content)
```



```
+ 1_simple_graph.ipynb  3_chatbot.ipynb  2_graph_with_condition.ipynb
+ ↑ ↓ ⌂ G ↵
  Safety_ratings : [], TURNTURN_REASON : STOP , model_name : gpt-3.5-turbo , Safety_ratings : [], id='run--aa55e1f2-68c5-45c0-8cdc-3abb3091e044-0' , usage_metadata={'input_tokens': 14, 'output_tokens': 3, 'total_tokens': 17, 'input_token_details': {'cache_read': 0}}]

state = None
while True:
    in_message = input("You: ")
    if in_message.lower() in {"quit", "exit"}:
        break
    if state is None:
        state: State = {
            "messages": [{"role": "user", "content": in_message}]
        }
    else:
        state["messages"].append({"role": "user", "content": in_message})

    state = graph.invoke(state)
    print("Bot:", state["messages"][-1].content)
    10s 130ms
```

Jupyter Input Request

You:  
Who was the first person to walk on the moon? pr

OK Cancel

```

if in_message.lower() in {"quit", "exit"}:
    break
if state is None:
    state = {
        "messages": [{"role": "user", "content": "Hello!"}]
    }
else:
    state["messages"].append({"role": "user", "content": "Hello!"})

```

There is now a context being preserved in the messages list and the LLM can answer using it as a form of memory

```

builder.add_edge(START, "chatbot_node")
builder.add_edge("chatbot_node", END)

graph = builder.compile()

# message = {"role": "user", "content": "Who walked on the moon for the first time? Print only the name"}
message = {"role": "user", "content": "What is the latest price of MSFT stock?"}
response = graph.invoke({"messages": [message]})

response["messages"]

```

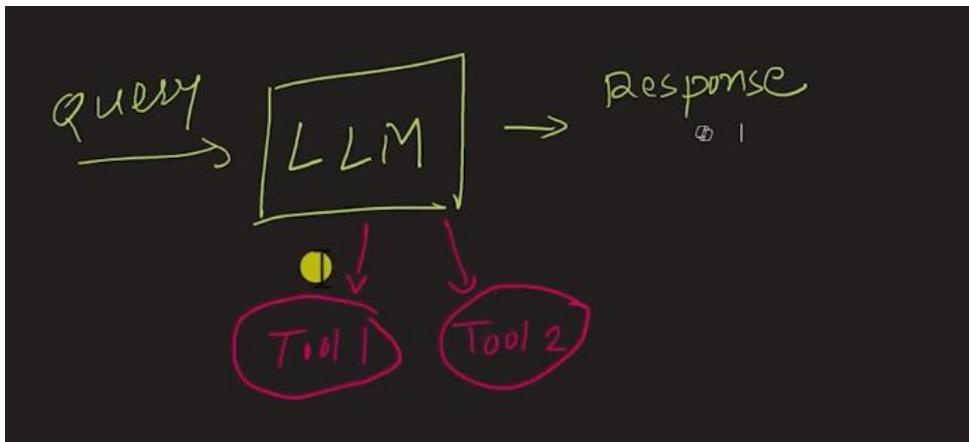
AIResponse(content='I am unable to provide real-time stock prices. The price of MSFT stock fluctuates throughout the day.\nTo get the latest price, I recommend checking one of the following sources:\n\* \*\*Google Finance:\*\* Search "MSFT stock price" on Google.\n\* \*\*Yahoo Finance:\*\* Visit [finance.yahoo.com/quote/MSFT](https://finance.yahoo.com/quote/MSFT)\n\* \*\*Bloomberg:\*\* Visit [bloomberg.com/quote/MSFT-US](https://www.bloomberg.com/quote/MSFT-US)\n\* \*\*Your Brokerage Account:\*\* If you have an account with a brokerage firm (e.g., Fidelity, Schwab, Robinhood), the current price will be displayed there.', 'model\_name': 'gemini-2.0-flash', 'safety\_ratings': [{}], 'temperature': 0.0, 'top\_p': 1.0, 'max\_tokens': 100, 'stop\_sequences': ['\n'], 'response\_metadata': {'prompt\_feedback': {}, 'block\_reason': 0, 'safety\_ratings': []}, 'id': 'run--ee7984ec-d496-46ab-8f3c-123456789012', 'timestamp': 1688451655.0, 'usage': {'prompt\_tokens': 10, 'completion\_tokens': 100, 'total\_tokens': 110}})

```

state = None
while True:
    in_message = input("You: ")
    if in_message.lower() in {"quit", "exit"}:

```

This is because the LLM's cut off is in the past, we need to add tools for the chatbot to use



Project: langgraph-learning master

```

from langchain.chat_models import init_chat_model
from typing import Annotated
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END
from langgraph.message import add_messages
from langchain_core.tools import tool
from langgraph.prebuilt import ToolNode, tools_condition

```

from dotenv import load\_dotenv  
load\_dotenv()  
True

class State(TypedDict):  
 # Messages have the type "list". The `add\_messages` function  
 # in the annotation defines how this state key should be updated  
 # (in this case, it appends messages to the list, rather than overwriting them)  
 messages: Annotated[list, add\_messages]

llm = init\_chat\_model("google\_genai:gemini-2.0-flash")



Project: langgraph-learning master

```

from langgraph.message import add_messages
from langchain_core.tools import tool
from langgraph.prebuilt import ToolNode, tools_condition

```

from dotenv import load\_dotenv  
load\_dotenv()  
True

class State(TypedDict):  
 # Messages have the type "list". The `add\_messages` function  
 # in the annotation defines how this state key should be updated  
 # (in this case, it appends messages to the list, rather than overwriting them)  
 messages: Annotated[list, add\_messages]

llm = init\_chat\_model("google\_genai:gemini-2.0-flash")



But we need to provide a tool that can get stock prices from a python function as a mock

langraph-learning master

1\_simple\_graph.ipynb 3\_chatbot.ipynb 4\_tool\_call.ipynb

+ ↑ ↓ ⌂ ⌂ ⌂

from langchain.chat\_models import init\_chat\_model  
 from typing import Annotated  
 from typing\_extensions import TypedDict  
 from langgraph.graph import StateGraph, START, END  
 from langgraph.graph.message import add\_messages  
 from langchain\_core.tools import tool  
 from langgraph.prebuilt import ToolNode, tools\_condition

✓ [1] 1s 192ms

from dotenv import load\_dotenv  
 load\_dotenv()  
 ✓ [2] 29ms

True

class State(TypedDict):  
 # Messages have the type "list". The `add\_messages` function  
 # in the annotation defines how this state key should be updated  
 # (in this case, it appends messages to the list, rather than overwriting them)  
 messages: Annotated[list, add\_messages]

✓ [3] 12ms

@tool

def get\_stock\_price(symbol: str) -> float:  
 '''Return the current price of a stock given the stock symbol  
 :param symbol: stock symbol  
 :return: current price of the stock  
 '''

✓ [4] 900ms



langraph-learning master

1\_simple\_graph.ipynb 3\_chatbot.ipynb 4\_tool\_call.ipynb

+ ↑ ↓ ⌂ ⌂ ⌂

✓ [2] 29ms

True

...

class State(TypedDict):  
 # Messages have the type "list". The `add\_messages` function  
 # in the annotation defines how this state key should be updated  
 # (in this case, it appends messages to the list, rather than overwriting them)  
 messages: Annotated[list, add\_messages]

✓ [3] 12ms

@tool

def get\_stock\_price(symbol: str) -> float:  
 '''Return the current price of a stock given the stock symbol  
 :param symbol: stock symbol  
 :return: current price of the stock  
 '''

return {  
 "MSFT": 200.3,  
 "AAPL": 100.4,  
 "AMZN": 150.0,  
 "RIL": 87.6  
 }.get(symbol, 0.0)

tools = [get\_stock\_price]

llm = init\_chat\_model("google\_genai:gemini-2.0-flash")  
 llm\_with\_tools = llm.bind\_tools(tools)

✓ [4] 900ms





```
llm = init_chat_model("google_genai:gemini-2.0-flash")
llm_with_tools = llm.bind_tools(tools)
✓ [4] 900ms

def chatbot(state: State):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}

builder = StateGraph(State)

builder.add_node(chatbot)
builder.add_node("tools", ToolNode(tools))

builder.add_edge(START, "chatbot")
builder.add_conditional_edges("chatbot", tools_condition)
builder.add_edge("chatbot", END)

graph = builder.compile()

from IPython.display import Image, display

display(Image(graph.get_graph().draw_mermaid_png()))
```

We are using a **tools** node and a **conditional edge** here because it will only be used if there is need for using the stock price tool



```
builder.add_node(chatbot)
builder.add_node("tools", ToolNode(tools))

builder.add_edge(START, "chatbot")
builder.add_conditional_edges("chatbot", tools_condition)
builder.add_edge("chatbot", END)

graph = builder.compile()

from IPython.display import Image, display

display(Image(graph.get_graph().draw_mermaid_png()))
✓ [5] 162ms
```

graph TD; START(( )) --> chatbot[chatbot]; chatbot --> tools[tools]; tools --> END(( ));

langgraph-learning master

1\_simple\_graph.ipynb 3\_chatbot.ipynb 4\_tool\_call.ipynb

```

return {"messages": [llm_with_tools.invoke(state["messages"])]}

builder = StateGraph(State)

builder.add_node(chatbot)
builder.add_node("tools", ToolNode(tools))

builder.add_edge(START, "chatbot")
builder.add_conditional_edges("chatbot", tools_condition)
builder.add_edge("chatbot", END)

```

graph = builder.compile()

from IPython.display import Image, display

```

graph().draw_mermaid_png()

```

langgraph-learning master

1\_simple\_graph.ipynb 3\_chatbot.ipynb 4\_tool\_call.ipynb tool\_node.py

```

class ToolNode(RunnableCallable):
    def _validate_tool_command(
        ...
    )
        raise ValueError(
            f"Expected to have a matching ToolMessage in Command.update for tool '{call[0]}'"
            "Every tool call (LLM requesting to call a tool) in the message history MUST have"
            f"You can fix it by modifying the tool to return {example_update}.""
        )
    return updated_command

    def tools_condition(
        state: Union[list[AnyMessage], dict[str, Any], BaseModel],
        messages_key: str = "messages",
    ) -> Literal["tools", "__end__"]:
        ...

```

```

return {"messages": [llm_with_tools.invoke(state["messages"])]}

builder = StateGraph(State)

builder.add_node(chatbot)
builder.add_node("tools", ToolNode(tools))

builder.add_edge(START, "chatbot")
builder.add_conditional_edges("chatbot", tools_condition)

graph = builder.compile()

```

from IPython.display import Image, display

```

display(Image(graph.get_graph().draw_mermaid_png()))

```

We can even remove the END line as above

```

from IPython.display import Image, display

```

```

display(Image(graph.get_graph().draw_mermaid_png()))

```

```

state = graph.invoke({"messages": [{"role": "user", "content": "What is the price of AMZN stock right now?"}]})
print(state["messages"][-1].content)    state

```

langgraph-learning master

1\_simple\_graph.ipynb 3\_chatbot.ipynb 4\_tool\_call.ipynb

[2] 29ms  
True

```
class State(TypedDict):
    # Messages have the type "list". The `add_messages` function
    # in the annotation defines how this state key should be updated
    # (in this case, it appends messages to the list, rather than overwriting them)
    messages: Annotated[list, add_messages]
```

[3] 12ms

```
@tool
def get_stock_price(symbol: str) -> float:
    '''Return the current price of a stock given the stock symbol
    :param symbol: stock symbol
    :return: current price of the stock
    '''
    return {
        "MSFT": 200.3,
        "AAPL": 100.4,
        "AMZN": 150.0,
        "RIL": 87.6
    }.get(symbol, 0.0)

tools = [get_stock_price]
```

```
llm = init_chat_model("google_genai:gemini-2.0-flash")
llm_with_tools = llm.bind_tools(tools)
```

[4] 900ms

langgraph-learning master

1\_simple\_graph.ipynb 3\_chatbot.ipynb 4\_tool\_call.ipynb

[6] 73ms

```
from IPython.display import Image, display
```

```
display(Image(graph.get_graph().draw_mermaid_png()))
```

```
state = graph.invoke({"messages": [{"role": "user", "content": "What is the price of AAPL stock right now?"}]})
print(state["messages"][-1].content)
```

[8] 528ms  
100.4

```
: [{"role": "user", "content": "What is the price of AAPL stock right now?"}])
```

content) state

```
state = {AddableValuesDict: 1} {'messages': [HumanMessage(content='What is the price of AAPL stock right now?', additional_kwarg... View
  'messages': [list: 3] [HumanMessage(content='What is the price of AAPL stock right now?', additional_kwarg... View
    > 0 = (HumanMessage) HumanMessage(content='What is the price of AAPL stock right now?', additional_kwarg... View
    > 1 = (AIMessage) AIMessage(content='', additional_kwarg... ('function_call': {'name': 'get_stock_price', 'arguments': {'symbol': ... View
    > 2 = (ToolMessage) ToolMessage(content='100.4', name='get_stock_price', id='2f222dd4-7026-4dec-a36a-12b2be58abc3', tool_call_id='ed3078d9-d234-49d9-a181... View
      > __len__ = (int) 3
    > > Protected Attributes
      > __len__ = (int) 1
    > > Protected Attributes
```

```
: [{"role": "user", "content": "What is the price of AAPL stock right now?"}])  
content) state  
  ✓ state = {AddableValuesDict: 1} {'messages': [HumanMessage(content='What is the price of AAPL stock right now?', additional_kwarg... View  
    ✓ 'messages' = (list: 3) [HumanMessage(content='What is the price of AAPL stock right now?', additional_kwarg..., response_meta... View  
      > 0 = (HumanMessage) HumanMessage(content='What is the price of AAPL stock right now?', additional_kwarg..., response_meta...  
      ✓ 1 = (AIMessage) AIMessage(content='', additional_kwarg...={'function_call': {'name': 'get_stock_price', 'arguments': {"symbol": ... View  
        > additional_kwarg... = (dict: 1) {'function_call': {'arguments': {"symbol": "AAPL"}, 'name': 'get_stock_price'}}  
          10 content = (str) "  
          11 example = (bool) False  
          12 id = (str) 'run--a0c8ca5b-8b0e-4123-9fd7-221897c7ac-0'  
          > invalid_tool_calls = (list: 0) []  
          > lc_attributes = (dict: 2) {'invalid_tool_calls': [], 'tool_calls': [(args: {'symbol': 'AAPL'}, id: 'ed3078d9-d234-49d9-a181-e18680d243bf', name: 'get_stock_price')]}  
          > lc_secrets = (dict: 0) {}  
          > model_computed_fields = (dict: 0) {}  
          > model_config = (dict: 1) {'extra': 'allow'}  
          > model_extra = (dict: 0) {}  
          > model_fields = (dict: 10) {'additional_kwarg...': FieldInfo(annotation=dict, required=False, default_factory=dict), 'content': FieldInfo(annotation=...  
          > model_fields_set = (set: 7) {'additional_kwarg...', 'content', 'id', 'invalid_tool_calls', 'response_metadata', 'tool_calls', 'usage_meta...  
            10 name = (NoneType) None  
          > response_metadata = (dict: 4) {'finish_reason': 'STOP', 'model_name': 'gemini-2.0-flash', 'prompt_feedback': {'block_reason': '...  
          > tool_calls = (list: 1) [(args: {'symbol': 'AAPL'}, id: 'ed3078d9-d234-49d9-a181-e18680d243bf', name: 'get_stock_price', ty...  
            10 type = (str) 'ai'  
          > usage_metadata = (dict: 4) {'input_token_details': {'cache_read': 0}, 'input_tokens': 49, 'output_tokens': 8, 'total_tokens': 57}  
  ✓ 2 = {ToolMessage} ToolMessage(content='100.4', name='get_stock_price', id='2f222dd4-7026-4dec-a36a-12b2be58abc3', tool_call_id='...  
    > additional_kwarg... = (dict: 0) {}  
    10 artifact = (NoneType) None  
    10 content = (str) '100.4'  
    10 id = (str) '2f222dd4-7026-4dec-a36a-12b2be58abc3'  
    > lc_attributes = (dict: 0) {}  
    > lc_secrets = (dict: 0) {}  
    > model_computed_fields = (dict: 0) {}  
    > model_config = (dict: 1) {'extra': 'allow'}  
    > model_extra = (dict: 0) {}  
    > model_fields = (dict: 9) {'additional_kwarg...': FieldInfo(annotation=dict, required=False, default_factory=dict, repr=False), 'content': FieldInfo(annotation=...  
    > model_fields_set = (set: 6) {'artifact', 'content', 'id', 'name', 'status', 'tool_call_id'}  
      10 name = (str) 'get_stock_price'  
    > response_metadata = (dict: 0) {}  
    10 status = (str) 'success'  
    10 tool_call_id = (str) 'ed3078d9-d234-49d9-a181-e18680d243bf'  
    10 type = (str) 'tool'  
  ✓ 3 = {HumanMessage} HumanMessage(content='The price of AAPL stock right now is 100.4', additional_kwarg..., response_meta... View  
    > additional_kwarg... = (dict: 1) {'function_call': {'name': 'get_stock_price', 'arguments': {"symbol": "AAPL"}, 'name': 'get_stoc...  
    > content = (str) 'The price of AAPL stock right now is 100.4'  
    > id = (str) 'run--a0c8ca5b-8b0e-4123-9fd7-221897c7ac-0'  
    > invalid_tool_calls = (list: 0) []  
    > lc_attributes = (dict: 2) {'invalid_tool_calls': [], 'tool_calls': []}  
    > lc_secrets = (dict: 0) {}  
    > model_computed_fields = (dict: 0) {}  
    > model_config = (dict: 1) {'extra': 'allow'}  
    > model_extra = (dict: 0) {}  
    > model_fields = (dict: 10) {'additional_kwarg...': FieldInfo(annotation=dict, required=False, default_factory=dict), 'content': FieldInfo(annotation=...  
    > model_fields_set = (set: 7) {'additional_kwarg...', 'content', 'id', 'invalid_tool_calls', 'response_metadata', 'tool_calls', 'usage_meta...  
      10 name = (NoneType) None  
    > response_metadata = (dict: 4) {'finish_reason': 'STOP', 'model_name': 'gemini-2.0-flash', 'prompt_feedback': {'block_reason': '...  
    > tool_calls = (list: 0) []  
    > usage_metadata = (dict: 4) {'input_token_details': {'cache_read': 0}, 'input_tokens': 49, 'output_tokens': 8, 'total_tokens': 57}  
  ✓ 4 = {HumanMessage} HumanMessage(content='The price of AAPL stock right now is 100.4', additional_kwarg..., response_meta... View  
    > additional_kwarg... = (dict: 1) {'function_call': {'name': 'get_stock_price', 'arguments': {"symbol": "AAPL"}, 'name': 'get_stoc...  
    > content = (str) 'The price of AAPL stock right now is 100.4'  
    > id = (str) 'run--a0c8ca5b-8b0e-4123-9fd7-221897c7ac-0'  
    > invalid_tool_calls = (list: 0) []  
    > lc_attributes = (dict: 2) {'invalid_tool_calls': [], 'tool_calls': []}  
    > lc_secrets = (dict: 0) {}  
    > model_computed_fields = (dict: 0) {}  
    > model_config = (dict: 1) {'extra': 'allow'}  
    > model_extra = (dict: 0) {}  
    > model_fields = (dict: 10) {'additional_kwarg...': FieldInfo(annotation=dict, required=False, default_factory=dict), 'content': FieldInfo(annotation=...  
    > model_fields_set = (set: 7) {'additional_kwarg...', 'content', 'id', 'invalid_tool_calls', 'response_metadata', 'tool_calls', 'usage_meta...  
      10 name = (NoneType) None  
    > response_metadata = (dict: 4) {'finish_reason': 'STOP', 'model_name': 'gemini-2.0-flash', 'prompt_feedback': {'block_reason': '...  
    > tool_calls = (list: 0) []  
    > usage_metadata = (dict: 4) {'input_token_details': {'cache_read': 0}, 'input_tokens': 49, 'output_tokens': 8, 'total_tokens': 57}
```



```
: [{"role": "user", "content": "What is the price of AAPL stock right now?"}])  
content) state  
  ✓ state = {AddableValuesDict: 1} {'messages': [HumanMessage(content='What is the price of AAPL stock right now?', additional_kwarg... View  
    ✓ 'messages' = (list: 3) [HumanMessage(content='What is the price of AAPL stock right now?', additional_kwarg..., response_meta... View  
      > 0 = (HumanMessage) HumanMessage(content='What is the price of AAPL stock right now?', additional_kwarg..., response_meta...  
      ✓ 1 = (AIMessage) AIMessage(content='', additional_kwarg...={'function_call': {'name': 'get_stock_price', 'arguments': {"symbol": ... View  
        > additional_kwarg... = (dict: 1) {'function_call': {'arguments': {"symbol": "AAPL"}, 'name': 'get_stock_price'}}  
          10 content = (str) "  
          11 example = (bool) False  
          12 id = (str) 'run--a0c8ca5b-8b0e-4123-9fd7-221897c7ac-0'  
          > invalid_tool_calls = (list: 0) []  
          > lc_attributes = (dict: 2) {'invalid_tool_calls': [], 'tool_calls': [(args: {'symbol': 'AAPL'}, id: 'ed3078d9-d234-49d9-a181-e18680d243bf', name: 'get_stock_price')]}  
          > lc_secrets = (dict: 0) {}  
          > model_computed_fields = (dict: 0) {}  
          > model_config = (dict: 1) {'extra': 'allow'}  
          > model_extra = (dict: 0) {}  
          > model_fields = (dict: 10) {'additional_kwarg...': FieldInfo(annotation=dict, required=False, default_factory=dict), 'content': FieldInfo(annotation=...  
          > model_fields_set = (set: 7) {'additional_kwarg...', 'content', 'id', 'invalid_tool_calls', 'response_metadata', 'tool_calls', 'usage_meta...  
            10 name = (NoneType) None  
          > response_metadata = (dict: 4) {'finish_reason': 'STOP', 'model_name': 'gemini-2.0-flash', 'prompt_feedback': {'block_reason': '...  
          > tool_calls = (list: 1) [(args: {'symbol': 'AAPL'}, id: 'ed3078d9-d234-49d9-a181-e18680d243bf', name: 'get_stock_price', ty...  
            10 type = (str) 'ai'  
          > usage_metadata = (dict: 4) {'input_token_details': {'cache_read': 0}, 'input_tokens': 49, 'output_tokens': 8, 'total_tokens': 57}  
  ✓ 2 = {ToolMessage} ToolMessage(content='100.4', name='get_stock_price', id='2f222dd4-7026-4dec-a36a-12b2be58abc3', tool_call_id='...  
    > additional_kwarg... = (dict: 0) {}  
    10 artifact = (NoneType) None  
    10 content = (str) '100.4'  
    10 id = (str) '2f222dd4-7026-4dec-a36a-12b2be58abc3'  
    > lc_attributes = (dict: 0) {}  
    > lc_secrets = (dict: 0) {}  
    > model_computed_fields = (dict: 0) {}  
    > model_config = (dict: 1) {'extra': 'allow'}  
    > model_extra = (dict: 0) {}  
    > model_fields = (dict: 9) {'additional_kwarg...': FieldInfo(annotation=dict, required=False, default_factory=dict, repr=False), 'content': FieldInfo(annotation=...  
    > model_fields_set = (set: 6) {'artifact', 'content', 'id', 'name', 'status', 'tool_call_id'}  
      10 name = (str) 'get_stock_price'  
    > response_metadata = (dict: 0) {}  
    10 status = (str) 'success'  
    10 tool_call_id = (str) 'ed3078d9-d234-49d9-a181-e18680d243bf'  
    10 type = (str) 'tool'  
  ✓ 3 = {HumanMessage} HumanMessage(content='The price of AAPL stock right now is 100.4', additional_kwarg..., response_meta... View  
    > additional_kwarg... = (dict: 1) {'function_call': {'name': 'get_stock_price', 'arguments': {"symbol": "AAPL"}, 'name': 'get_stoc...  
    > content = (str) 'The price of AAPL stock right now is 100.4'  
    > id = (str) 'run--a0c8ca5b-8b0e-4123-9fd7-221897c7ac-0'  
    > invalid_tool_calls = (list: 0) []  
    > lc_attributes = (dict: 2) {'invalid_tool_calls': [], 'tool_calls': []}  
    > lc_secrets = (dict: 0) {}  
    > model_computed_fields = (dict: 0) {}  
    > model_config = (dict: 1) {'extra': 'allow'}  
    > model_extra = (dict: 0) {}  
    > model_fields = (dict: 10) {'additional_kwarg...': FieldInfo(annotation=dict, required=False, default_factory=dict), 'content': FieldInfo(annotation=...  
    > model_fields_set = (set: 7) {'additional_kwarg...', 'content', 'id', 'invalid_tool_calls', 'response_metadata', 'tool_calls', 'usage_meta...  
      10 name = (NoneType) None  
    > response_metadata = (dict: 4) {'finish_reason': 'STOP', 'model_name': 'gemini-2.0-flash', 'prompt_feedback': {'block_reason': '...  
    > tool_calls = (list: 0) []  
    > usage_metadata = (dict: 4) {'input_token_details': {'cache_read': 0}, 'input_tokens': 49, 'output_tokens': 8, 'total_tokens': 57}
```



langgraph-learning master

1\_simple.graph.ipynb 3\_chatbot.ipynb 4\_tool\_call.ipynb

```

graph TD
    start(( )) --> tools[tools]
    tools --> end_1([end])
    end_1 --> tools

```

```

state = graph.invoke({"messages": [{"role": "user", "content": "What is the price of AAPL stock right now?"}]})
print(state["messages"][-1].content)

```

[8] 528ms  
100.4

state = (AddableValuesDict: 1) {'messages': [HumanMessage(content='What is the price of AAPL stock right now?', additional\_kwarg... View

messages = (list: 3) [HumanMessage(content='What is the price of AAPL stock right now?', additional\_kwarg... response\_me... View

0 = (HumanMessage) HumanMessage(content='What is the price of AAPL stock right now?', additional\_kwarg... response\_meta... View

1 = (AIMessage) AIMessage(content='', additional\_kwarg... {'name': 'get\_stock\_price', 'arguments': {'symbol': ... View

additional\_kwarg... (dict: 1) {'function\_call': {'arguments': {'symbol': 'AAPL', 'name': 'get\_stock\_price'}}}

content = (str) ''

example = (bool) False

id = (str) run-a0c8ca5b-8b0e-4123-9fd7-2218970e27ac-0'

invalid\_tool\_calls = (list: 0) []

lc\_attributes = (dict: 2) {'invalid\_tool\_calls': [], 'tool\_calls': ['args': {'symbol': 'AAPL'}, 'id': 'ed3078d9-d234-49d9-a181-e18680d2... View

lc\_secrets = (dict: 0) {}

model\_computed\_fields = (dict: 0) {}

model\_config = (dict: 1) {'extra': 'allow'}

model\_extra = (dict: 0) {}

model\_fields = (dict: 10) {additional\_kwarg... FieldInfo(annotation=dict, required=False, default\_factory=dict), 'content': Fi... View

model\_fields\_set = (set: 7) {'additional\_kwarg... 'content', 'id', 'invalid\_tool\_calls', 'response\_metadata', 'tool\_calls', 'usage\_metadata'... View

name = (NoneType) None

response\_metadata = (dict: 4) {'finish\_reason': 'STOP', 'model\_name': 'gemini-2.0-flash', 'prompt\_feedback': {'block\_reaso... 0' View

tool\_calls = (list: 1) ['args': {'symbol': 'AAPL'}, 'id': 'ed3078d9-d234-49d9-a181-e18680d243bf', 'name': 'get\_stock\_price', 'type': ... View

type = (str) 'ai'

usage\_metadata = (dict: 4) {'input\_token\_details': {'cache\_read': 0}, 'input\_tokens': 49, 'output\_tokens': 8, 'total\_tokens': 57}

```

state = graph.invoke({"messages": [{"role": "user", "content": "What is the price of AAPL stock right now?"}]})
print(state["messages"][-1].content)

```

[8] 528ms  
100.4

```

state = graph.invoke({"messages": [{"role": "user", "content": "Who invented theory of relativity? print person name only"}]})
print(state["messages"][-1].content)

```

[9] 524ms  
Albert Einstein

```

: [{"role": "user", "content": "Who invented theory of relativity? print person name only"}])
ent) state

```

state = (AddableValuesDict: 1) {'messages': [HumanMessage(content='Who invented theory of relativity? print person name only', a... View

messages = (list: 2) [HumanMessage(content='Who invented theory of relativity? print person name only', additional\_kwarg... View

\_len\_ = (int) 1

> ◊ Protected Attributes

```

: [{"role": "user", "content": "Who invented theory of relativity? print person name only"}])
ent) state

```

state = (AddableValuesDict: 1) {'messages': [HumanMessage(content='Who invented theory of relativity? print person name only', a... View

messages = (list: 2) [HumanMessage(content='Who invented theory of relativity? print person name only', additional\_kwarg... View

0 = (HumanMessage) HumanMessage(content='Who invented theory of relativity? print person name only', additional\_kwarg... View

1 = (AIMessage) AIMessage(content='Albert Einstein', additional\_kwarg... response\_metadata={'prompt\_feedback': {'block... View

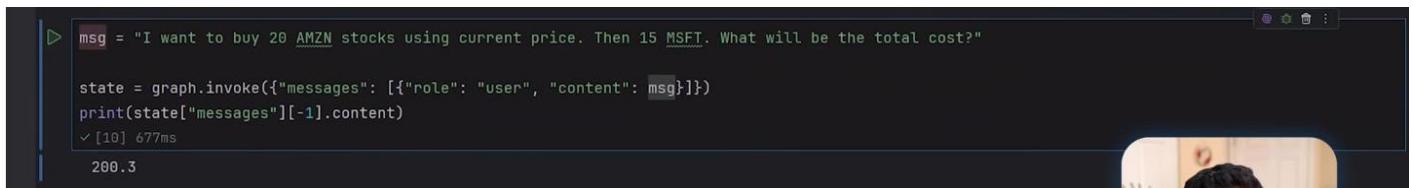
\_len\_ = (int) 2

> ◊ Protected Attributes

\_len\_ = (int) 1

> ◊ Protected Attributes

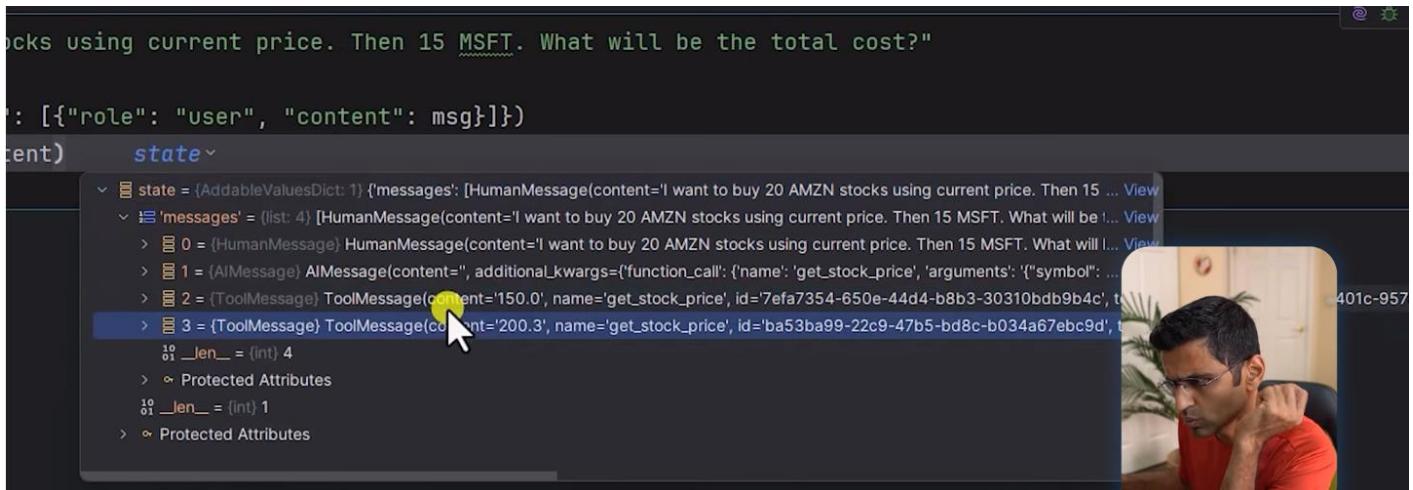
No ToolMessage seen here because the LLM correctly knows it doesn't need a tool for that question.



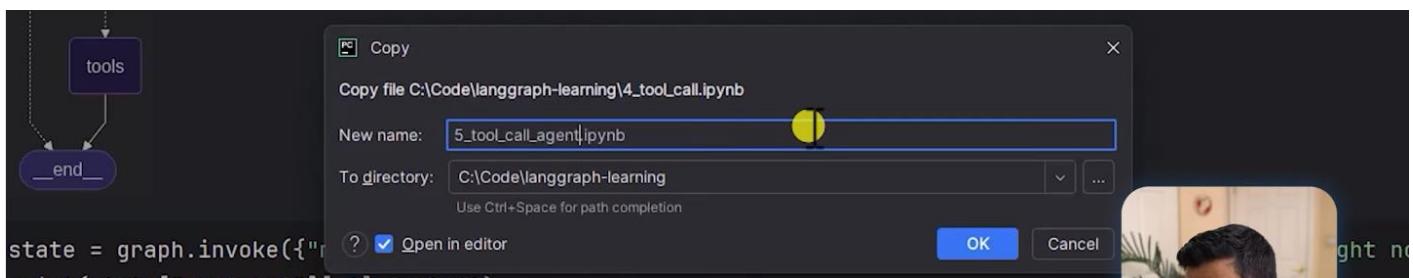
```
msg = "I want to buy 20 AMZN stocks using current price. Then 15 MSFT. What will be the total cost?"  
  
state = graph.invoke({"messages": [{"role": "user", "content": msg}])  
print(state["messages"][-1].content)  
✓ [10] 677ms  
200.3
```

The LLM just returned the MSFT stock price using the tool and did not do the expected Math since it did not return the value back to the LLM to reflect on.

Let us copy the chatbot code and use it to create our agent as below



```
ocks using current price. Then 15 MSFT. What will be the total cost?"  
  
: [{"role": "user", "content": msg}])  
tent) state  
v state = {AddableValuesDict: 1} {'messages': [HumanMessage(content='I want to buy 20 AMZN stocks using current price. Then 15 ... View  
v messages' = ([list: 4] [HumanMessage(content='I want to buy 20 AMZN stocks using current price. Then 15 MSFT. What will be t... View  
> 0 = {HumanMessage} HumanMessage(content='I want to buy 20 AMZN stocks using current price. Then 15 MSFT. What will ... View  
> 1 = {AIMessage} AIMessage(content='', additional_kwargs={'function_call': {'name': 'get_stock_price', 'arguments': ('symbol': ... View  
> 2 = {ToolMessage} ToolMessage(content='150.0', name='get_stock_price', id='7efa7354-650e-44d4-b8b3-30310bdb9b4c', t... View  
> 3 = {ToolMessage} ToolMessage(content='200.3', name='get_stock_price', id='ba53ba99-22c9-47b5-bd8c-b034a67ebc9d', t... View  
10 __len__ = (int) 4  
> ⚠ Protected Attributes  
11 __len__ = (int) 1  
> ⚠ Protected Attributes
```



Project langgraph-learning master

1.simple\_graph.ipynb 3.chatbot.ipynb 4.tool\_call.ipynb 5.tool\_call\_agent.ipynb

.env .gitignore .python-version 1.simple\_graph.ipynb 2.graph\_with\_condition 3.chatbot.ipynb 4.tool\_call.ipynb 5.tool\_call\_agent.ipynb pyproject.toml README.md uv.lock External Libraries Scratches and Consoles

```
from langchain.chat_models import init_chat_model
from typing import Annotated
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END
from langgraph.message import add_messages
from langchain_core.tools import tool
from langchain_prebuilt import ToolNode, tools_condition

from dotenv import load_dotenv
load_dotenv()

True

class State(TypedDict):
    # Messages have the type "list". The `add_messages` function
    # in the annotation defines how this state key should be updated
    # (in this case, it appends messages to the list, rather than overwriting them)
    messages: Annotated[list, add_messages]

@tool
def get_stock_price(symbol: str) -> float:
    '''Return the current price of a stock given the stock symbol
    :param symbol: stock symbol
    :return: current price of the stock
    '''
    return {
        "MSFT": 200.3,
        "AAPL": 100.4,
        "AMZN": 150.0,
        "RIL": 87.6
    }.get(symbol, 0.0)

tools = [get_stock_price]

llm = init_chat_model("google_genai:gemini-2.0-flash")
llm_with_tools = llm.bind_tools(tools)

def chatbot(state: State):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}

builder = StateGraph(State)

builder.add_node(chatbot)
builder.add_node("tools", ToolNode(tools))

builder.add_edge(START, "chatbot")
builder.add_conditional_edges("chatbot", tools_condition)
```



Project langgraph-learning master

1.simple\_graph.ipynb 3.chatbot.ipynb 4.tool\_call.ipynb 5.tool\_call\_agent.ipynb

.env .gitignore .python-version 1.simple\_graph.ipynb 2.graph\_with\_condition 3.chatbot.ipynb 4.tool\_call.ipynb 5.tool\_call\_agent.ipynb pyproject.toml README.md uv.lock External Libraries Scratches and Consoles

```
def get_stock_price(symbol: str) -> float:
    '''Return the current price of a stock given the stock symbol
    :param symbol: stock symbol
    :return: current price of the stock
    '''
    return {
        "MSFT": 200.3,
        "AAPL": 100.4,
        "AMZN": 150.0,
        "RIL": 87.6
    }.get(symbol, 0.0)

tools = [get_stock_price]

llm = init_chat_model("google_genai:gemini-2.0-flash")
llm_with_tools = llm.bind_tools(tools)

def chatbot(state: State):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}

builder = StateGraph(State)

builder.add_node(chatbot)
builder.add_node("tools", ToolNode(tools))

builder.add_edge(START, "chatbot")
builder.add_conditional_edges("chatbot", tools_condition)
```



langgraph-learning

```

builder.add_node(chatbot)
builder.add_node("tools", ToolNode(tools))

builder.add_edge(START, "chatbot")
builder.add_conditional_edges("chatbot", tools_condition)

graph = builder.compile()

from IPython.display import Image, display

display(Image(graph.get_graph().draw_mermaid_png()))

```

```

state = graph.invoke({"messages": [{"role": "user", "content": "What is the price of AAPL stock right now?"}])
print(state["messages"][-1].content)
100.4

state = graph.invoke({"messages": [{"role": "user", "content": "Who invented theory of relativity? print person name only"}])

```

We want the tool response to be returned to the chatbot LLM to reflect on and make the next condition or go to END

langgraph-learning

```

builder.add_edge(START, "chatbot")
builder.add_conditional_edges("chatbot", tools_condition)
builder.add_edge("tools", "chatbot")
graph = builder.compile()

from IPython.display import Image, display

display(Image(graph.get_graph().draw_mermaid_png()))

```

```

state = graph.invoke({"messages": [{"role": "user", "content": "What is the price of AAPL stock right now?"}])
print(state["messages"][-1].content)

```

```

msg = "I want to buy 20 AMZN stocks using current price. Then 15 MSFT. What will be the total cost?"

state = graph.invoke({"messages": [{"role": "user", "content": msg}]})
print(state["messages"][-1].content)

```

```

The current price for AMZN is $150 and for MSFT is $200.3.
The total cost for 20 AMZN stocks is 20 * $150 = $3000.
The total cost for 15 MSFT stocks is 15 * $200.3 = $3004.5.
Therefore, the total cost for both is $3000 + $3004.5 = $6004.5.

```

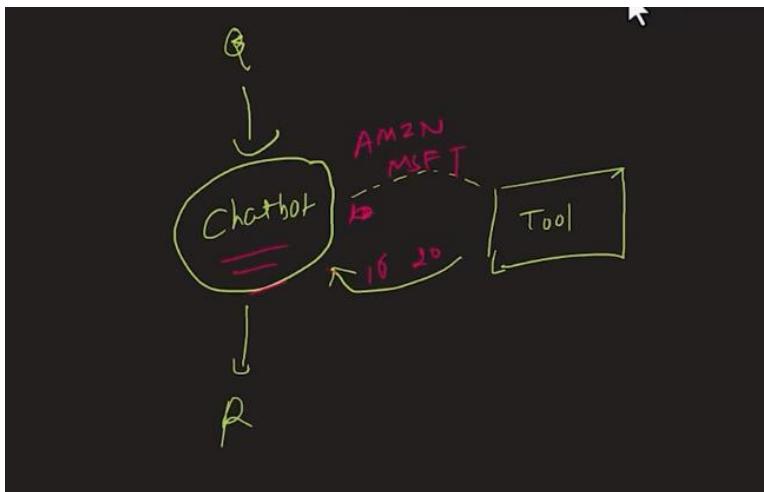
```

: [{"role": "user", "content": msg}])
ent) state
  ↘ state = {AddableValuesDict: 1} {messages: [HumanMessage(content='I want to buy 20 AMZN stocks using current price. Then 15 ... view
  ↘ 'messages' = (list: 5) [HumanMessage(content='I want to buy 20 AMZN stocks using current price. Then 15 MSFT. What will be ... View
  > ↗ 0 = (HumanMessage) HumanMessage(content='I want to buy 20 AMZN stocks using current price. Then 15 MSFT. What will be ... View
  > ↗ 1 = (AIMessage) AIMessage(content='', additional_kwargs={'function_call': {'name': 'get_stock_price', 'arguments': {'symbol': ... View
  > ↗ 2 = (ToolMessage) ToolMessage(content='100', name='get_stock_price', id='7ee19e5a-a66e-4012-809f-7784e188d800', tool_call
  > ↗ 3 = (ToolMessage) ToolMessage(content='20', name='get_stock_price', id='b522aff5-9535-406b-a563-9456fb22066', tool_ca
  > ↗ 4 = (AIMessage) AIMessage(content='The current price for AMZN is $150 and for MSFT is $200.3.\nThe total cost for 20 AM... View
  ↗ _len__ = (int) 5
  > ↗ Protected Attributes
  ↗ _len__ = (int) 1
  > ↗ Protected Attributes

```



It now correctly reasoning on what next best step to make in an agentic manner.



```

MINGW64/c/Code/langgraph-tutorial (main)
new file:  5_tool_call_agent.ipynb

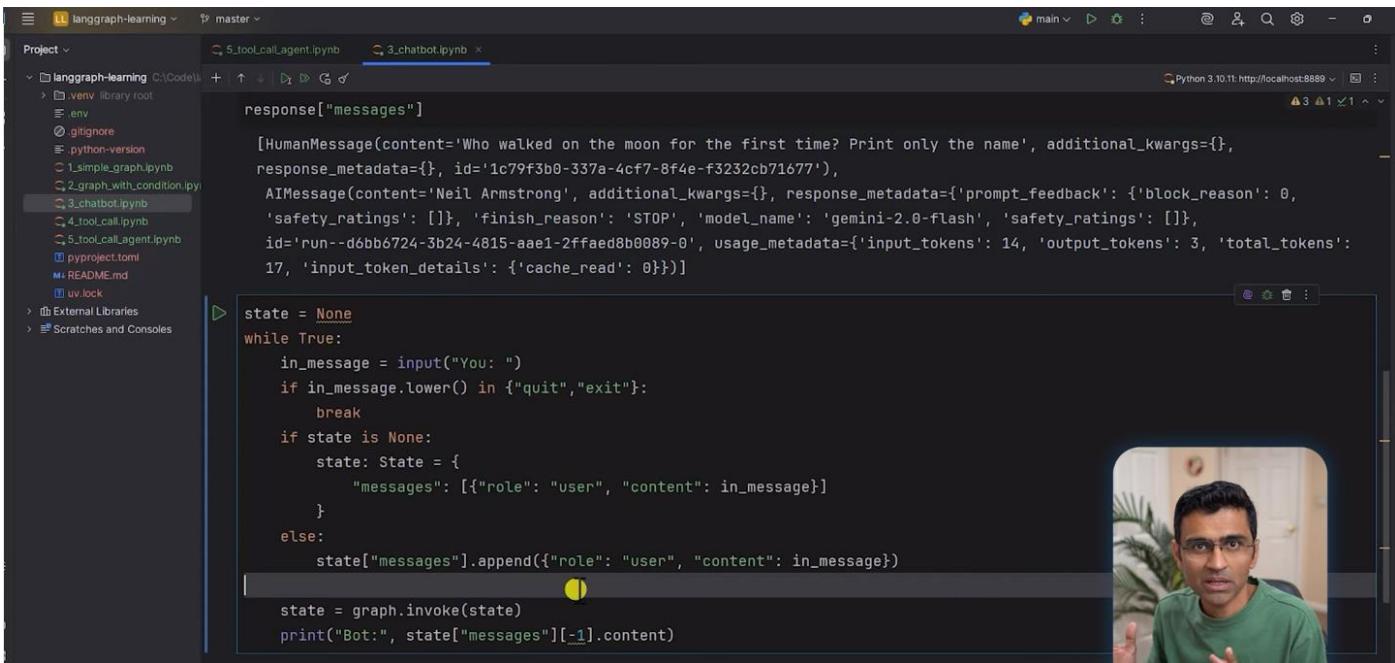
dhava@DP MINGW64 /c/Code/langgraph-tutorial (main)
$ git commit -m 'tool call'
[main 9bbe8f0] tool call
2 files changed, 481 insertions(+)
create mode 100644 4_tool_call.ipynb
create mode 100644 5_tool_call_agent.ipynb

dhava@DP MINGW64 /c/Code/langgraph-tutorial (main)
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 20 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 20.80 KiB | 20.80 MiB/s, done.
Total 4 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/codebasics/langgraph-tutorial.git
 c7c02c9..9bbe8f0 main -> main

dhava@DP MINGW64 /c/Code/langgraph-tutorial (main)
$ |

```





The screenshot shows a Jupyter Notebook interface with a code cell containing Python code for a chatbot. The code uses a list to store messages and prints them out. A video overlay of a man is visible in the bottom right corner.

```

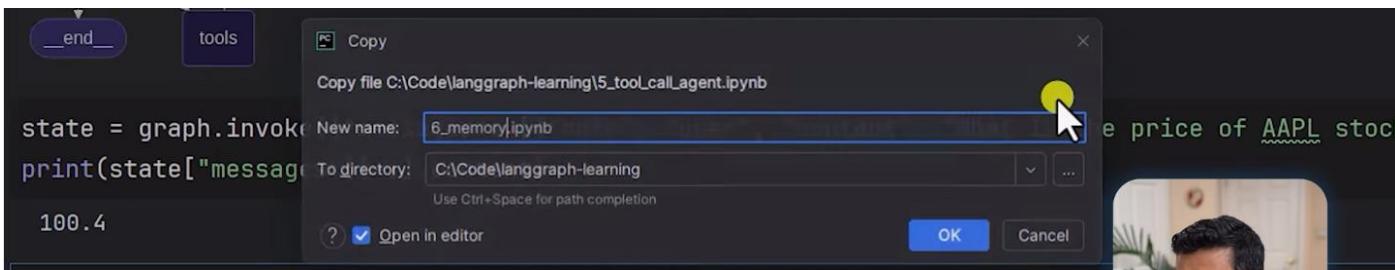
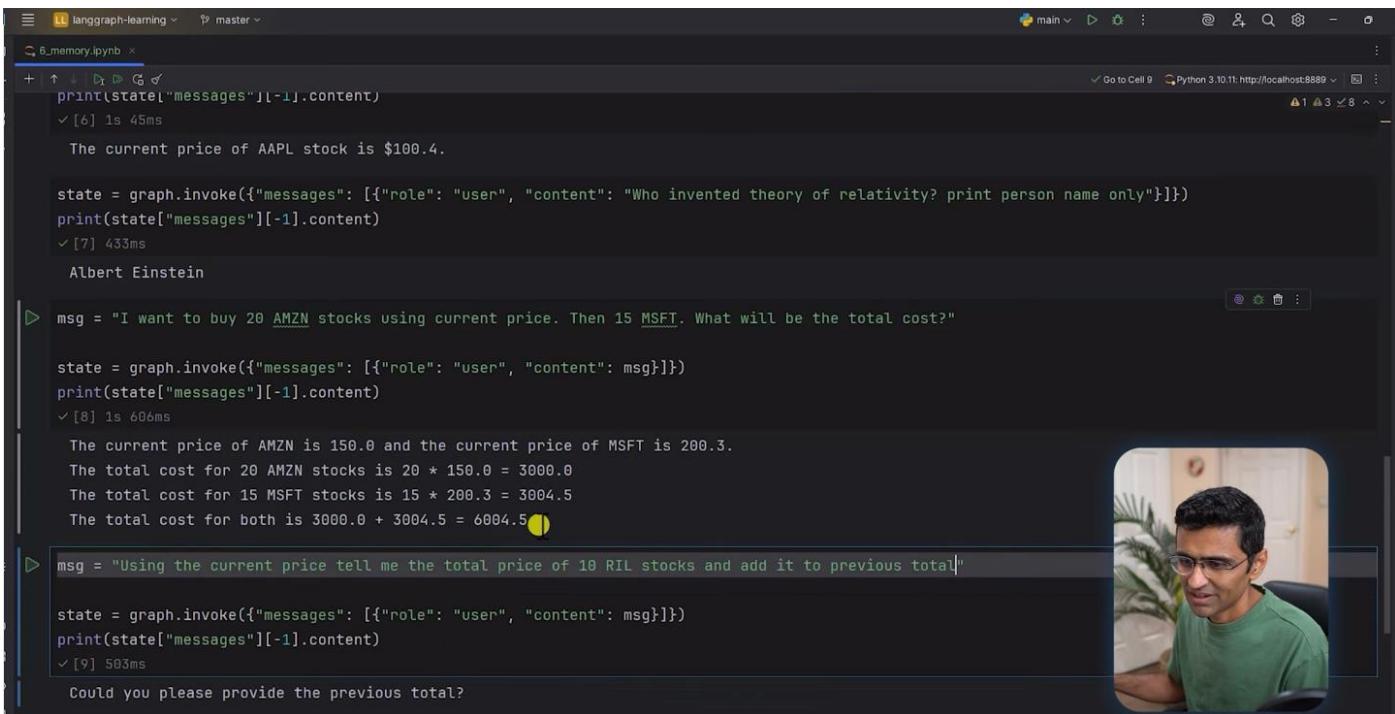
response["messages"]

[HumanMessage(content='Who walked on the moon for the first time? Print only the name', additional_kwargs={}, response_metadata={}, id='1c79f3b0-337a-4cf7-8f4e-f3232cb71677'),
 AIMessage(content='Neil Armstrong', additional_kwargs={'prompt_feedback': {'block_reason': 0, 'safety_ratings': []}, 'finish_reason': 'STOP', 'model_name': 'gemini-2.0-flash', 'safety_ratings': []}, id='run--d6bb6724-3b24-4815-aae1-2fffaed8b0089-0', response_metadata={'input_tokens': 14, 'output_tokens': 3, 'total_tokens': 17, 'input_token_details': {'cache_read': 0}})]

state = None
while True:
    in_message = input("You: ")
    if in_message.lower() in {"quit", "exit"}:
        break
    if state is None:
        state = State([
            {"role": "user", "content": in_message}
        ])
    else:
        state["messages"].append({"role": "user", "content": in_message})
    state = graph.invoke(state)
    print("Bot:", state["messages"][-1].content)

```

Up until now, we have been using a python list with appended messages to keep context. Langgraph has a Memory construct that we can easily use.

The screenshot shows a Jupyter Notebook cell with the following code and output:

```

state = graph.invoke(["messages": [{"role": "user", "content": "Who invented theory of relativity? print person name only"}]])
print(state["messages"][-1].content)

The current price of AAPL stock is $100.4.

state = graph.invoke(["messages": [{"role": "user", "content": "Who invented theory of relativity? print person name only"}]])
print(state["messages"][-1].content)

Albert Einstein

msg = "I want to buy 20 AMZN stocks using current price. Then 15 MSFT. What will be the total cost?"

state = graph.invoke(["messages": [{"role": "user", "content": msg}]])
print(state["messages"][-1].content)

The current price of AMZN is 150.0 and the current price of MSFT is 200.3.
The total cost for 20 AMZN stocks is 20 * 150.0 = 3000.0
The total cost for 15 MSFT stocks is 15 * 200.3 = 3004.5
The total cost for both is 3000.0 + 3004.5 = 6004.5

msg = "Using the current price tell me the total price of 10 RIL stocks and add it to previous total"

state = graph.invoke(["messages": [{"role": "user", "content": msg}]])
print(state["messages"][-1].content)

Could you please provide the previous total?

```

A video overlay of a man is visible in the bottom right corner.

This is what happens when there is no memory being kept as context.

# Add memory

The chatbot can now [use tools](#) to answer user questions, but it does not remember the context of previous interactions. This limits its ability to have coherent, multi-turn conversations.

LangGraph solves this problem through **persistent checkpointing**. If you provide a `checkpoint` when compiling the graph and a `thread_id` when calling your graph, LangGraph automatically saves the state of the graph at each step. When you invoke the graph again using the same `thread_id`, the graph loads its saved state, allowing the chatbot to pick up where it left off.

We will see later that **checkpointing** is *much* more powerful than simple chat memory - it lets you resume complex state at any time for error recovery, human-in-the-loop workflows, time travel inference, and more. But first, let's add checkpointing to enable multi-turn conversations.



## 1. Create a MemorySaver checkpointer

Create a `MemorySaver` checkpointer:

```
f Tm langgraph.checkpoint.memory import MemorySaver  
memory = MemorySaver()
```

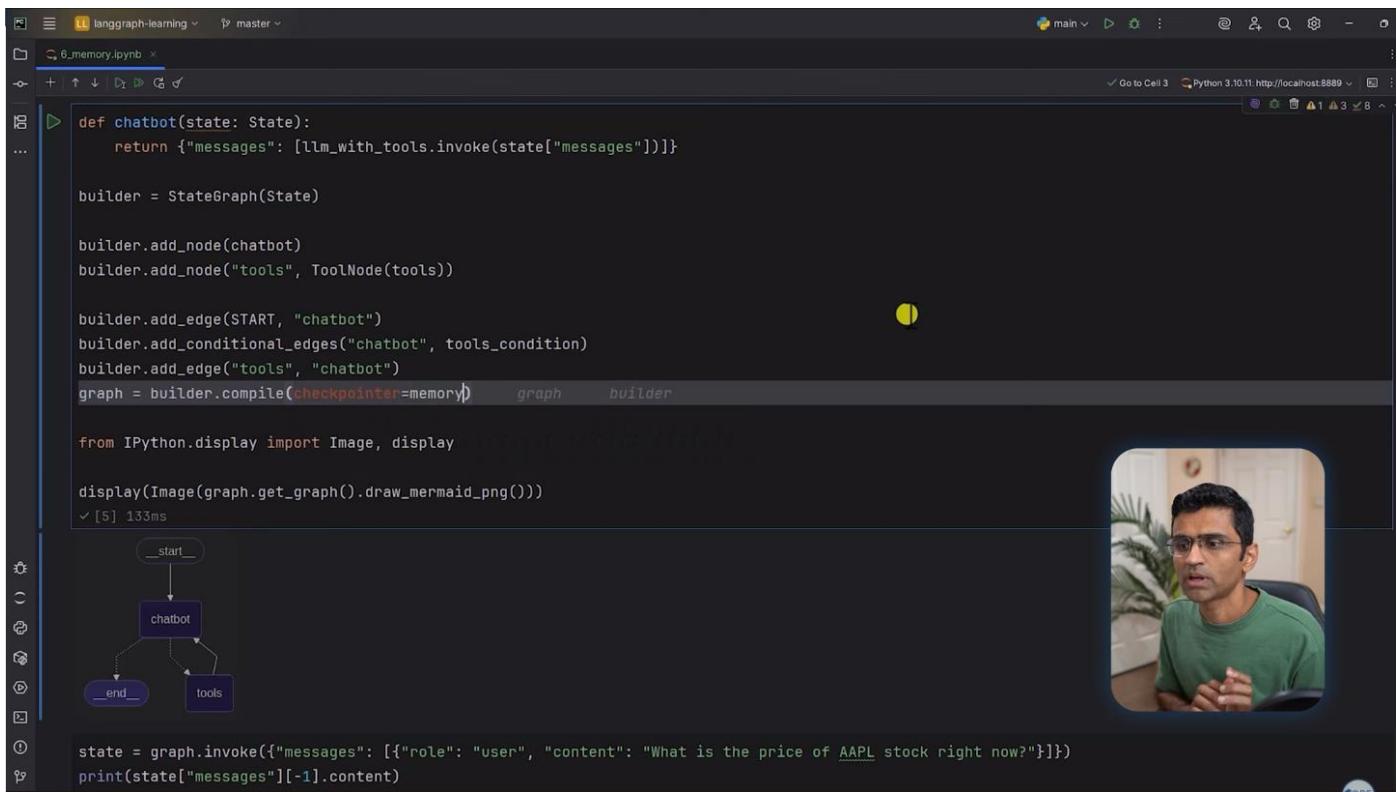
This is in-memory checkpointer, which is convenient for the tutorial. However, in a production app, you would likely change this to use `SqLiteSaver` or `PostgresSaver` and connect a database.



## 2. Compile the graph ¶

Compile the graph with the provided checkpointer, which will checkpoint the `State` as the graph works

```
> import ...  
✓ [1] 639ms  
  
from dotenv import load_dotenv  
load_dotenv()  
✓ [2] 29ms  
True  
  
from langgraph.checkpoint.memory import MemorySaver  
memory = MemorySaver()  
✓ [10] 21ms  
  
class State(TypedDict):  
    # Messages have the type "list". The `add_messages` function  
    # in the annotation defines how this state key should be updated  
    # (in this case, it appends messages to the list, rather than overwriting them)  
    messages: Annotated[list, add_messages]  
✓ [3] 14ms  
  
@tool  
def get_stock_price(symbol: str) -> float:  
    """Return the current price of a stock given the stock symbol  
    :param symbol: stock symbol  
    :return: current price of the stock  
    """  
    return {
```



```

def chatbot(state: State):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}

builder = StateGraph(State)

builder.add_node(chatbot)
builder.add_node("tools", ToolNode(tools))

builder.add_edge(START, "chatbot")
builder.add_conditional_edges("chatbot", tools_condition)
builder.add_edge("tools", "chatbot")
graph = builder.compile(checkpointer=memory) graph builder

```

from IPython.display import Image, display

```

display(Image(graph.get_graph().draw_mermaid_png()))

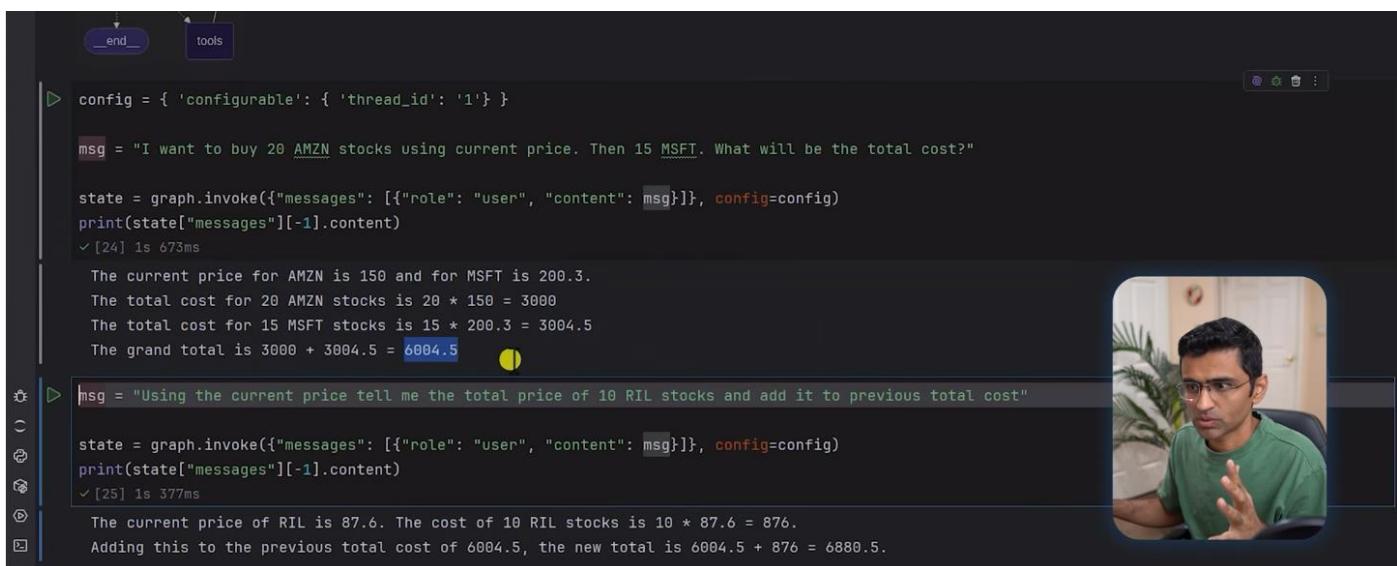
```

state = graph.invoke({"messages": [{"role": "user", "content": "What is the price of AAPL stock right now?"}]})  
print(state["messages"][-1].content)

[5] 133ms

The screenshot shows a Jupyter Notebook cell containing Python code to build a StateGraph. The graph starts at a node labeled 'start', which points to a central node labeled 'chatbot'. From 'chatbot', two edges lead down to nodes labeled 'end' and 'tools'. A conditional edge from 'chatbot' to 'tools' is labeled 'tools\_condition'. Below the code, the generated Mermaid diagram is displayed as a graph. To the right of the code, there is a video frame of a man in a green shirt.

When you compile the graph, add the **memory** object to the **checkpointer** as above. Then add the **config** as below



```

config = { 'configurable': { 'thread_id': '1' } }

msg = "I want to buy 20 AMZN stocks using current price. Then 15 MSFT. What will be the total cost?"

state = graph.invoke({"messages": [{"role": "user", "content": msg}], config=config)
print(state["messages"][-1].content)

```

The current price for AMZN is 150 and for MSFT is 200.3.  
The total cost for 20 AMZN stocks is  $20 \times 150 = 3000$ .  
The total cost for 15 MSFT stocks is  $15 \times 200.3 = 3004.5$ .  
The grand total is  $3000 + 3004.5 = 6004.5$

```

msg = "Using the current price tell me the total price of 10 RIL stocks and add it to previous total cost"

state = graph.invoke({"messages": [{"role": "user", "content": msg}], config=config)
print(state["messages"][-1].content)

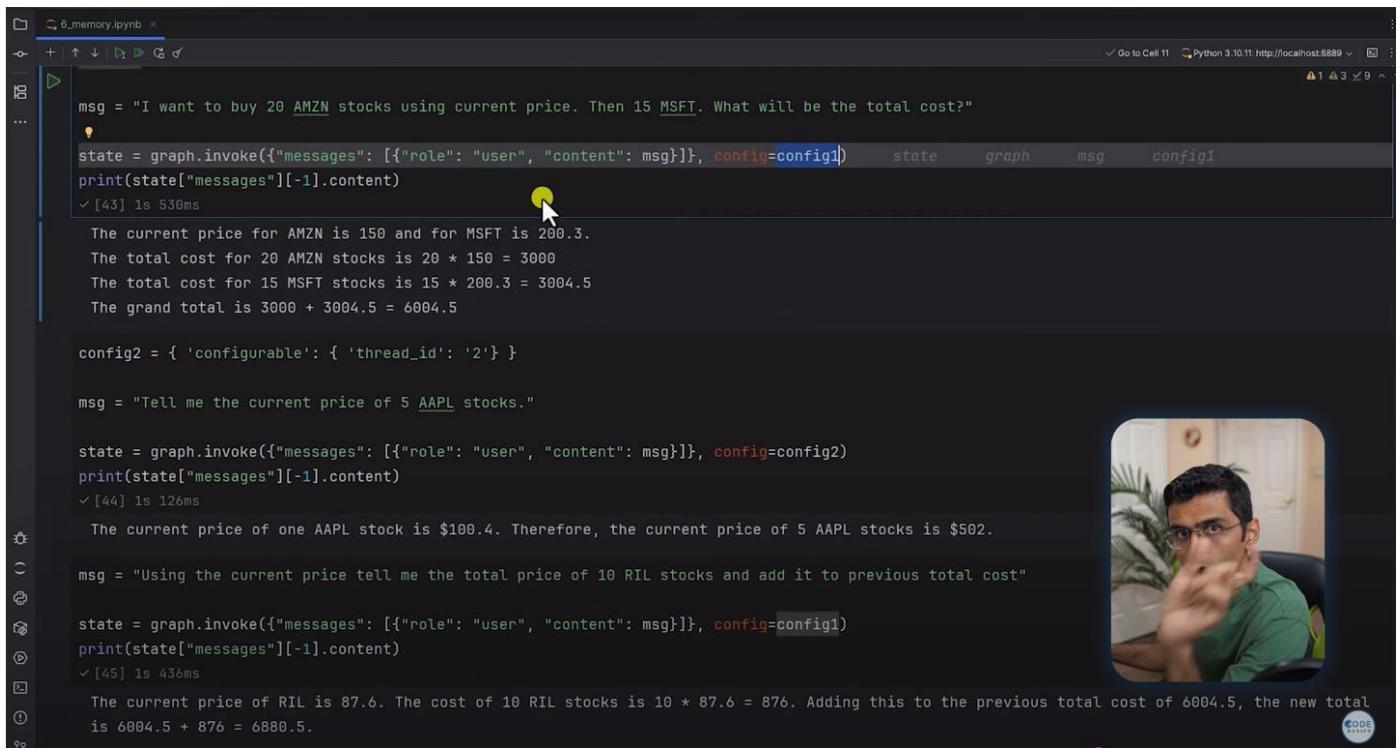
```

The current price of RIL is 87.6. The cost of 10 RIL stocks is  $10 \times 87.6 = 876$ .  
Adding this to the previous total cost of 6004.5, the new total is  $6004.5 + 876 = 6880.5$ .

[24] 1s 673ms  
[25] 1s 377ms

The screenshot shows a Jupyter Notebook cell with two invocations of the graph. The first invocation uses a configuration parameter 'config' with a 'configurable' key. The second invocation adds another message to the total cost calculation. To the right of the code, there is a video frame of a man in a green shirt.

After adding the **config** to the **graph.invoke()**, we can see that the results are correct because the memory is passed



```

msg = "I want to buy 20 AMZN stocks using current price. Then 15 MSFT. What will be the total cost?"
state = graph.invoke({"messages": [{"role": "user", "content": msg}], config=config1)
print(state["messages"][-1].content)
✓ [43] 1s 530ms
The current price for AMZN is 150 and for MSFT is 200.3.
The total cost for 20 AMZN stocks is 20 * 150 = 3000
The total cost for 15 MSFT stocks is 15 * 200.3 = 3004.5
The grand total is 3000 + 3004.5 = 6004.5

config2 = { 'configurable': { 'thread_id': '2' } }

msg = "Tell me the current price of 5 AAPL stocks."

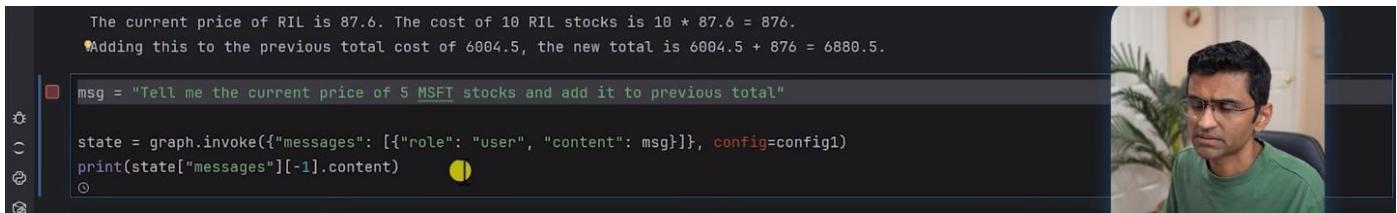
state = graph.invoke({"messages": [{"role": "user", "content": msg}], config=config2)
print(state["messages"][-1].content)
✓ [44] 1s 126ms
The current price of one AAPL stock is $100.4. Therefore, the current price of 5 AAPL stocks is $502.

msg = "Using the current price tell me the total price of 10 RIL stocks and add it to previous total cost"

state = graph.invoke({"messages": [{"role": "user", "content": msg}], config=config1)
print(state["messages"][-1].content)
✓ [45] 1s 436ms
The current price of RIL is 87.6. The cost of 10 RIL stocks is 10 * 87.6 = 876. Adding this to the previous total cost of 6004.5, the new total is 6004.5 + 876 = 6880.5.

```

We can even have **multiple thread\_ids** for maintaining **multiple memory contexts** to be used when needed



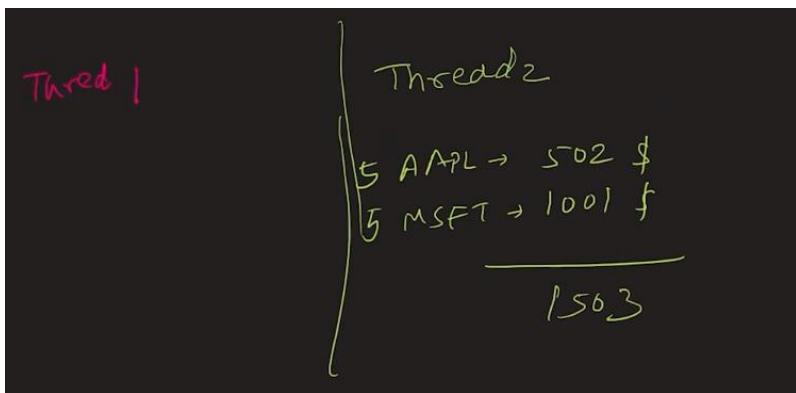
```

The current price of RIL is 87.6. The cost of 10 RIL stocks is 10 * 87.6 = 876.
Adding this to the previous total cost of 6004.5, the new total is 6004.5 + 876 = 6880.5.

msg = "Tell me the current price of 5 MSFT stocks and add it to previous total"

state = graph.invoke({"messages": [{"role": "user", "content": msg}], config=config1)
print(state["messages"][-1].content)

```



Handwritten notes:

Thread 1

Thread 2

$5 \text{ AAPL} \rightarrow \$502$

$5 \text{ MSFT} \rightarrow \$1001.5$

$\hline$

$\$1503$

## Add memory

[Back to top](#)

The chatbot can now **use tools** to answer user questions, but it does not remember the context of previous interactions. This limits its ability to have coherent, multi-turn conversations.

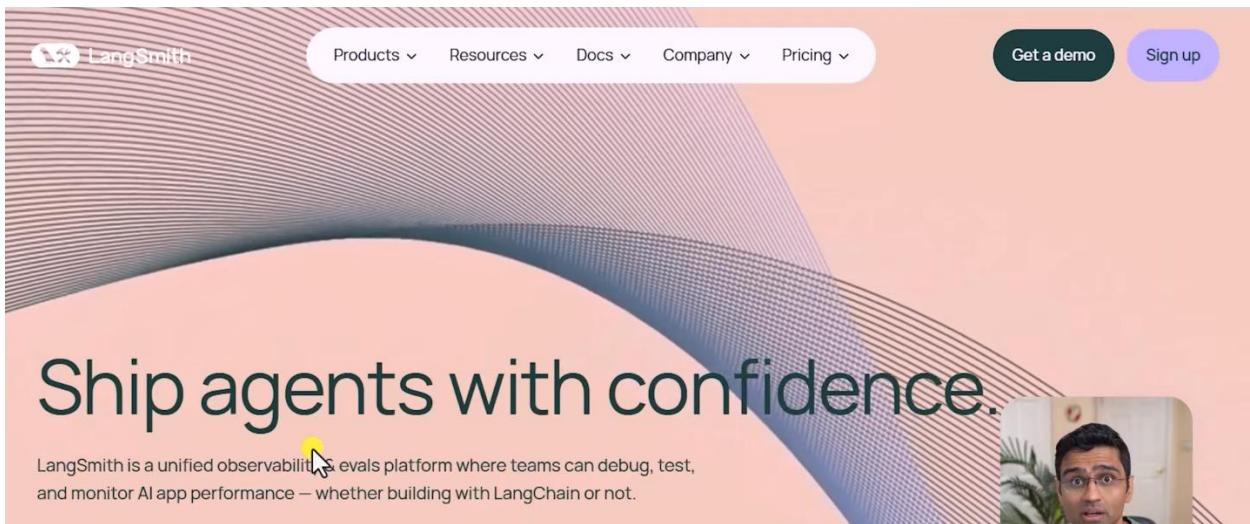
LangGraph solves this problem through **persistent checkpointing**. If you provide a `checkpointer` when compiling the graph and a `thread_id` when calling your graph, LangGraph automatically saves the state after each step. When you invoke the graph again using the same `thread_id`, the graph loads its saved state, allowing the chatbot to pick up where it left off.

We will see later that **checkpointing** is much more powerful than simple chat memory - it lets you save complex state at any time for error recovery, human-in-the-loop workflows, time travel interactivity and more. But first, let's add checkpointing to enable multi-turn conversations.

### Table of contents

- Create a Memory checkpointer
  - Compile the graph
  - Interact with the graph
  - Ask a follow-up question
  - Inspect the state
- Next steps





A screenshot of the LangSmith dashboard titled "Personal". The dashboard includes sections for "Get Started" (with "Set up tracing", "Run an evaluation", and "Try out playground" buttons), "Observability" (showing 2 Tracing Projects and a table with columns: Name, Most Recent Run (7D), Feedback (7D), Run Count (7D), and Error Rate (7D)), and a video player showing a man speaking.

Log in with your Google ID

A screenshot of the LangSmith dashboard with a yellow highlight box around the "Set up tracing" button in the "Get Started" section. This indicates that the user is currently on the login screen after clicking the "Sign in with Google" button.

← Set up tracing

GET STARTED WITH LANGSMITH

## Set up observability

Trace, debug and monitor your application

With LangChain Without LangChain

1. **Generate API Key**

2. Install dependencies

```
1 pip install -U langchain langchain-openai
```

3. Configure environment to connect to LangSmith.

Project Name: pr-unnatural-icicle-52

```
1 LANGSMITH_TRACING=true
2 LANGSMITH_ENDPOINT="https://api.smith.langchain.com"
3 LANGSMITH_API_KEY="<your-api-key>"
```

Python  Copy

langgraph-learning master

Project langgraph-learning C:\Code\ll

6\_memory.ipynb .env

```
GOOGLE_API_KEY=
OPENAI_API_KEY=
LANGSMITH_API_KEY=
LANGSMITH_TRACING=true
LANGSMITH_ENDPOINT="https://api.smith.langchain.com"
LANGSMITH_PROJECT="Langgraph-learning"
```

.env .gitignore .python-version 1\_simple\_graph.ipynb 2\_graph\_with\_condition.ipynb 3\_chatbot.ipynb 4\_tool\_call.ipynb 5\_tool\_call\_agent.ipynb 6\_memory.ipynb pyproject.toml README.md uv.lock

External Libraries Scratches and Consoles

← Set up tracing

2. Install dependencies

Python TypeScript

```
1 pip install -U langchain langchain-openai
```

3. Configure environment to connect to LangSmith.

Project Name: pr-unnatural-icicle-52

```
1 LANGSMITH_TRACING=true
2 LANGSMITH_ENDPOINT="https://api.smith.langchain.com"
3 LANGSMITH_API_KEY="lsv2_pt_111609d267bf4de381aba47a4bc87aa_4180
ceec63"
4 LANGSMITH_PROJECT="pr-unnatural-icicle-52"
5 OPENAI_API_KEY="<your-openai-api-key>"
```

4. Run any LLM, Chat model, or Chain. Its trace will be sent to the set project.

```
1 from langchain_openai import ChatOpenAI
2
3 llm = ChatOpenAI()
4 llm.invoke("Hello, world!")
```

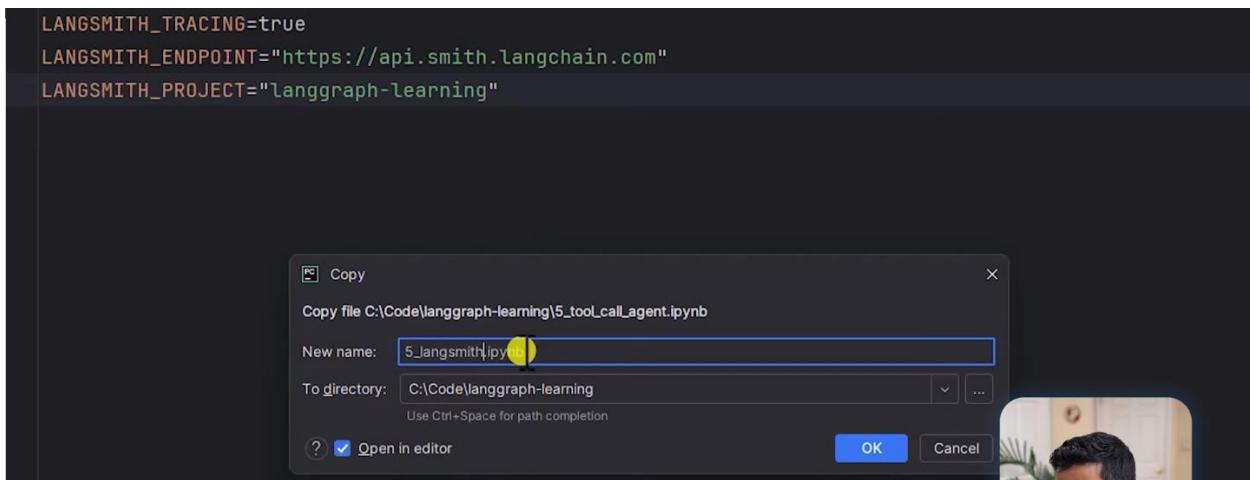
Congratulations, on logging your first trace in LangSmith. Here are some topics to explore next:



```

GOOGLE_API_KEY=AIzaSyAgaW-etR25cCuo6rqDzL2a1rBjMMiYpb0
OPENAI_API_KEY=sk-proj-YPumAKlxacupXcpMaOMeeyOma1C9YH_3xFWE01oDbJWbCV5lUWa
LANGSMITH_API_KEY=lsv2_pt_111609d267bf4de381aabaa47a4bc87aa_4180ceec63
LANGSMITH_TRACING=true
LANGSMITH_ENDPOINT="https://api.smith.langchain.com"
LANGSMITH_PROJECT="langgraph-learning"

```



```

+ rsa==4.9.1
(langgraph-learning) PS C:\Code\langgraph-learning> uv add langsmith
Resolved 137 packages in 537ms
Audited 131 packages in 0.87ms
(langgraph-learning) PS C:\Code\langgraph-learning>

```

```

state = graph.invoke({"messages": [{"role": "user", "content": "What is the price of AAPL stock right now?"}]}
print(state["messages"][-1].content)

```

100.4

We can now add annotations for Langsmith to pick up traces of our calls

Project

```

from langsmith import traceable

@traceable
def call_graph(query: str):
    state = graph.invoke({"messages": [{"role": "user", "content": query}]})
    return state["messages"][-1].content

call_graph("I want to buy 20 AMZN stocks using current price. Then 15 MSFT. What will be the total cost?")

```

The screenshot shows a Jupyter Notebook interface with several cells running. Cell 5, titled '5\_Jangsmith.ipynb', is currently active and displays Python code related to LangSmith libraries. The code includes imports from langchain.chat\_models, typing, langgraph.graph, langchain\_core.tools, and langgraph.prebuilt. It defines a State class using TypedDict and a get\_stock\_price tool function. A video overlay of a man in a green shirt is visible in the top right corner.

```

from langchain.chat_models import init_chat_model
from typing import Annotated
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages
from langchain_core.tools import tool
from langgraph.prebuilt import ToolNode, tools_condition
✓ [1] 703ms

from dotenv import load_dotenv
load_dotenv() ⏪
✓ [2] 28ms
True

class State(TypedDict):
    # Messages have the type "list". The `add_messages` function
    # in the annotation defines how this state key should be updated
    # (in this case, it appends messages to the list, rather than overwriting them)
    messages: Annotated[list, add_messages]
✓ [3] 12ms

@tool
def get_stock_price(symbol: str) -> float:
    '''Return the current price of a stock given the stock symbol
    :param symbol: stock symbol
    :return: current price of the stock
    '''

```

Clicking Run All will add all the Langsmith libraries for us

The screenshot shows a Jupyter Notebook interface with several cells running. Cell 5, titled '5\_Jangsmith.ipynb', is currently active and displays Python code related to LangSmith libraries. The code includes imports from IPython.display and langsmith. It also shows a Mermaid diagram of a state machine with nodes '\_start', 'chatbot', '\_end', and 'tools'. A video overlay of a man in a green shirt is visible in the bottom right corner.

```

from IPython.display import Image, display

display(Image(graph.get_graph().draw_mermaid_png()))
✓ [5] 153ms

graph TD
    _start --> chatbot
    chatbot --> _end
    chatbot --> tools
    tools --> _end

from langsmith import traceable
@traceable
def call_graph(query: str):
    state = graph.invoke({"messages": [{"role": "user", "content": query}]})
    return state["messages"][-1].content

call_graph("I want to buy 20 AMZN stocks using current price. Then 15 MSFT. What will be the total")
✓ [6] 1s 778ms

```

'The current price for AMZN is 150 and for MSFT is 200.3.\n\nThe total cost for 20 AMZN stocks cost for 15 MSFT stocks is 15 \* 200.3 = 3004.5\n\nThe grand total is 3000 + 3004.5 = 6004.5'

[LangSmith](#) [LangSmith](#) + smith.langchain.com/o/2bfff779-a6b1-4c29-91f4-7262e9a696b5

← Set up tracing

1 pip install -U langchain langchain-openai ⌂ Copy

3. Configure environment to connect to LangSmith.

Project Name  
pr-unnatural-icicle-52

```
1 LANGSMITH_TRACING=true
2 LANGSMITH_ENDPOINT="https://api.smith.langchain.com"
3 LANGSMITH_API_KEY="lsv2_pt_111609d267bf4de381aaba47a4bc87aa_4180
ceec63"
4 LANGSMITH_PROJECT="pr-unnatural-icicle-52"
5 OPENAI_API_KEY=<your-openai-api-key>" ⌂ Copy
```

4. Run any LLM, Chat model, or Chain. Its trace will be sent to the set project.

```
1 from langchain_openai import ChatOpenAI ⌂ Copy
2 llm = ChatOpenAI()
3 llm.invoke("Hello, world!")
```

🎉 Congratulations, on logging your first trace in LangSmith. Here are some topics to explore next:

- Read the conceptual guide to learn about runs, traces, projects and more.
- Set up tracing with LangChain, LangGraph, or the other supported integration options.
- If you prefer video tutorials, check out Tracing Basics from LangSmith Academy.

Personal

Setup res...

Personal ID

Get Started

Set up tracing Run an evaluation Try out playground

Observability

Tracing Projects 3 Custom Dashboards 0

Name	Most Recent Run (7D)	Feedback (7D)	Run Count (7D)	Error Rate (7D)	P99 Latency (7D)	P99 Latency (7D)
default	6/22/2025, 6:29:20 PM		1	0%	7.31s	7.31s
langgraph-learning	6/22/2025, 3:37:56 PM		1	0%	1.76s	1.76s
codebasics-langgraph-tutorial	6/22/2025, 3:28:09 PM		2	0%	1.50s	1.50s

Showing 5 most active projects.

Personal > Tracing Projects > langgraph-learning Add res...

langgraph-learning

ID Data Retention 14d Dashboard

Runs Threads Alerts Setup

filter Last 7 days Traces LLM Calls All Runs Columns

Name	Input	Output	Error	Start Time	Latency	Dataset	Annotation Q
call_graph	I want to buy 20 AMZ...	The current price...		6/22/2025, 3:37:56 ...	1.76s		

Stats

Run Count 1 Total Tokens 284 / \$0.00 Median Token 284 Error Rate

Personal

lang

call\_graph

Run Feedback Metadata

Input

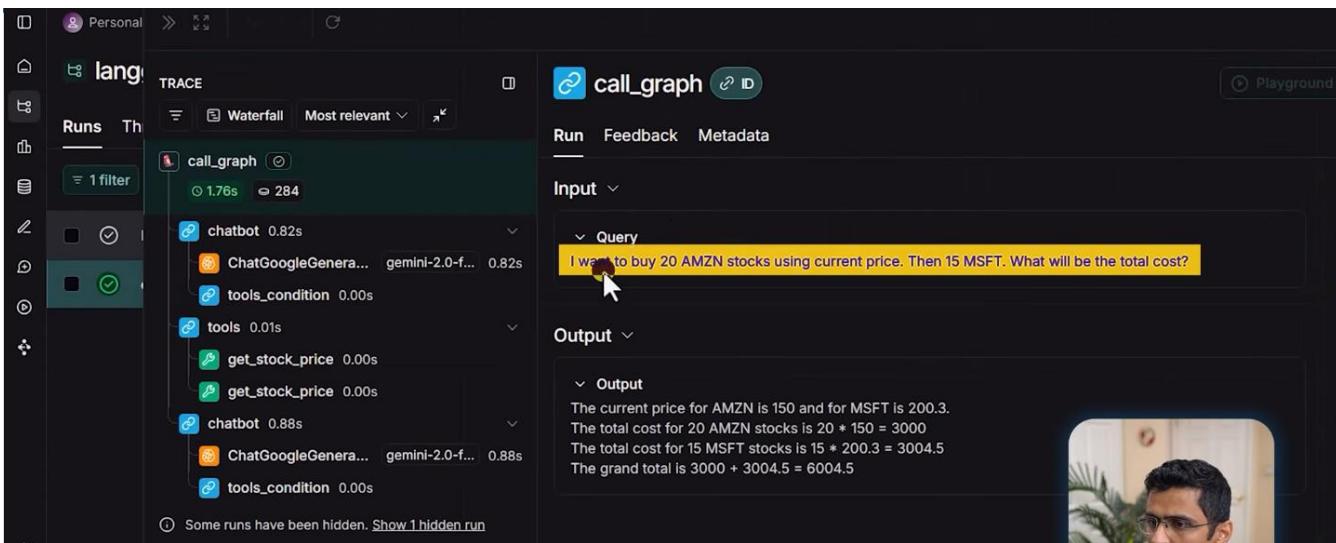
Query

I want to buy 20 AMZN stocks using current price. Then 15 MSFT. What will be the total cost?

Output

Output

The current price for AMZN is 150 and for MSFT is 200.3.  
The total cost for 20 AMZN stocks is  $20 * 150 = 3000$   
The total cost for 15 MSFT stocks is  $15 * 200.3 = 3004.5$   
The grand total is  $3000 + 3004.5 = 6004.5$



Waterfall Most relevant

call\_graph

Run Feedback Metadata

Input

Query

I want to buy 20 AMZN stocks using current price. Then 15 MSFT. What will be the total cost?

Output

Output

The current price for AMZN is 150 and for MSFT is 200.3.  
The total cost for 20 AMZN stocks is  $20 * 150 = 3000$   
The total cost for 15 MSFT stocks is  $15 * 200.3 = 3004.5$   
The grand total is  $3000 + 3004.5 = 6004.5$

START TIME  
06/22/2025, 03:37:56 PM

END TIME  
06/22/2025, 03:37:58 PM

TIME TO FIRST TOKEN

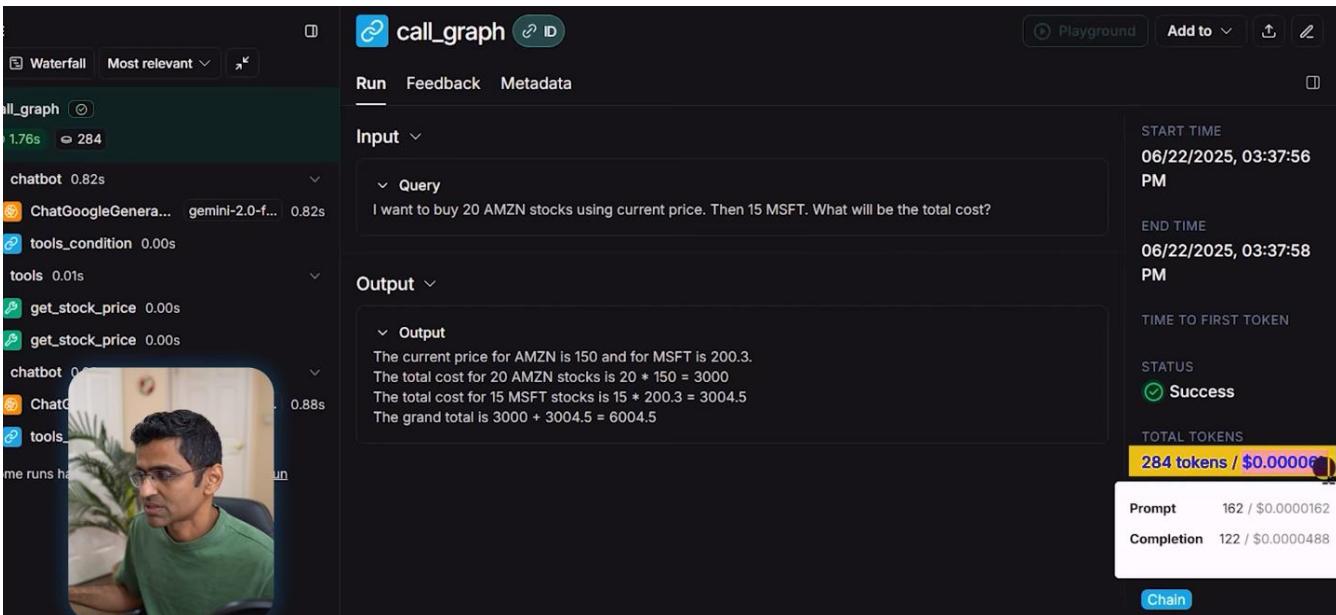
STATUS  
Success

TOTAL TOKENS  
284 tokens / \$0.00006

Prompt 162 / \$0.0000162

Completion 122 / \$0.0000488

Chain



Personal

lang

get\_stock\_price

Run Feedback Metadata

Input

Input

{'symbol': 'AMZN'}

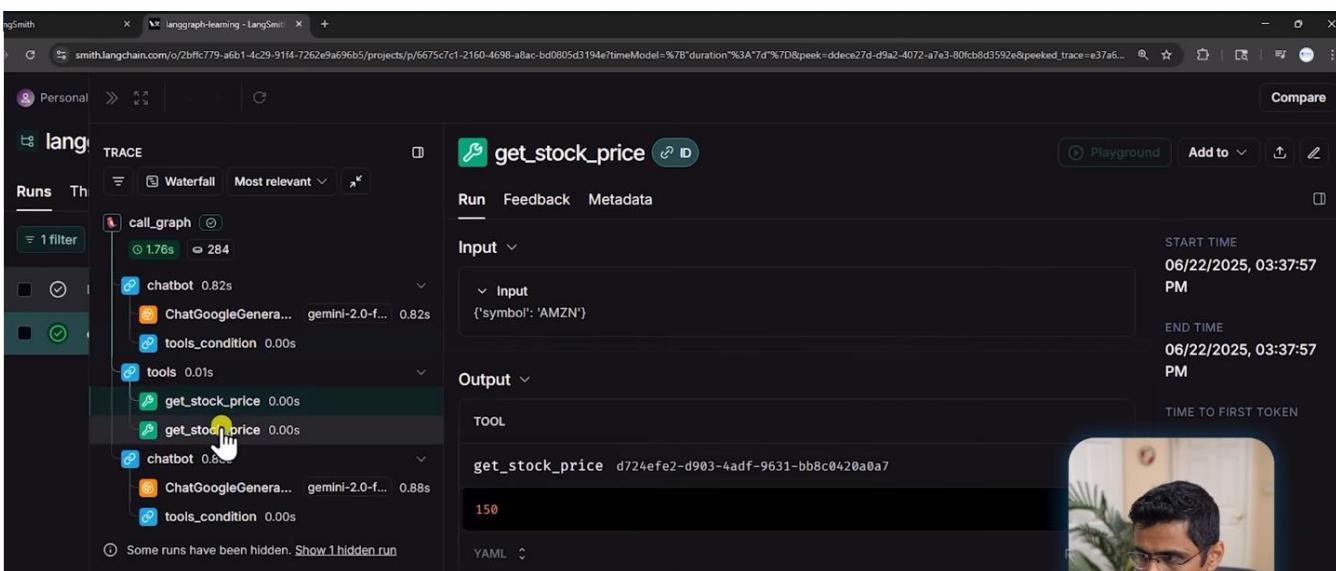
Output

TOOL

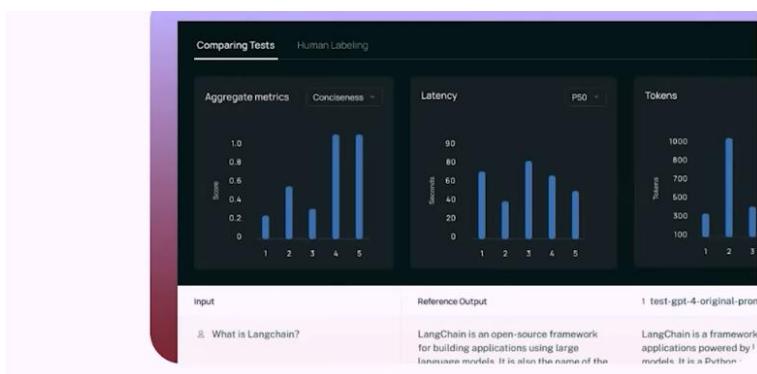
get\_stock\_price d724efef2-d903-4adf-9631-bb8c0420a0a7

150

YAML



We can now monitor cost and latency, output checks can also be performed easily



Evaluate your agent's performance.

Evaluate your app by saving production traces to datasets – then score performance with LLM-as-Judge evaluators. Gather human feedback from subject-matter experts to assess response relevance, correctness, safety, and other criteria.

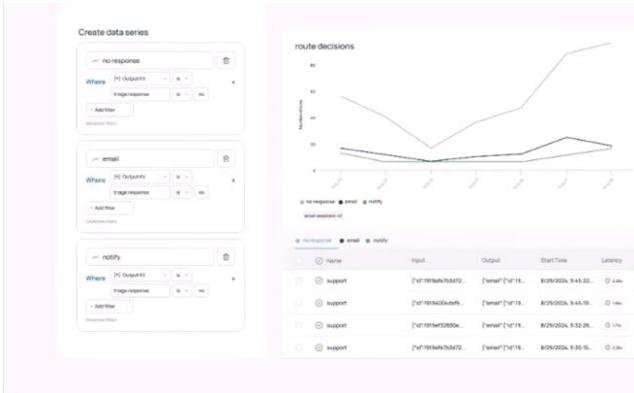
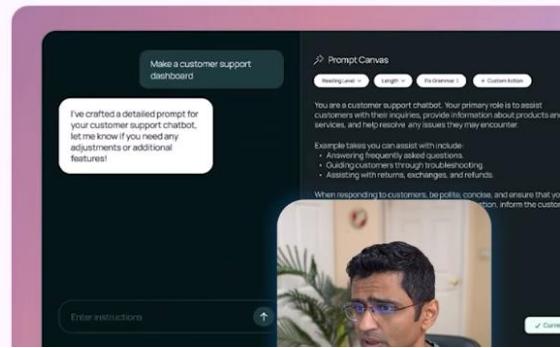
[Learn how to run an eval](#)



## Iterate and collaborate on prompts.

Experiment with models and prompts in the Playground, and compare outputs across different prompt versions. Any teammate can use the Prompt Canvas UI to directly recommend and improve prompts.

Create and test a prompt ↗



## Monitor what matters to the business.

Track business-critical metrics like costs, latency, and response quality with live dashboards – then get alerted when problems arise and drill into root cause.

See how to create a custom dashboard



# Human in the loop - HITL

You might want human approvals within some agentic workflows like below

A screenshot of a code editor showing a Python file named "8\_HITL.py". The code defines a class "State" with a "messages" attribute and a method "get\_stock\_price". A video player in the bottom right corner shows a man with glasses and a black shirt, looking at the camera. The code editor interface includes a sidebar with project files and a status bar showing "master".

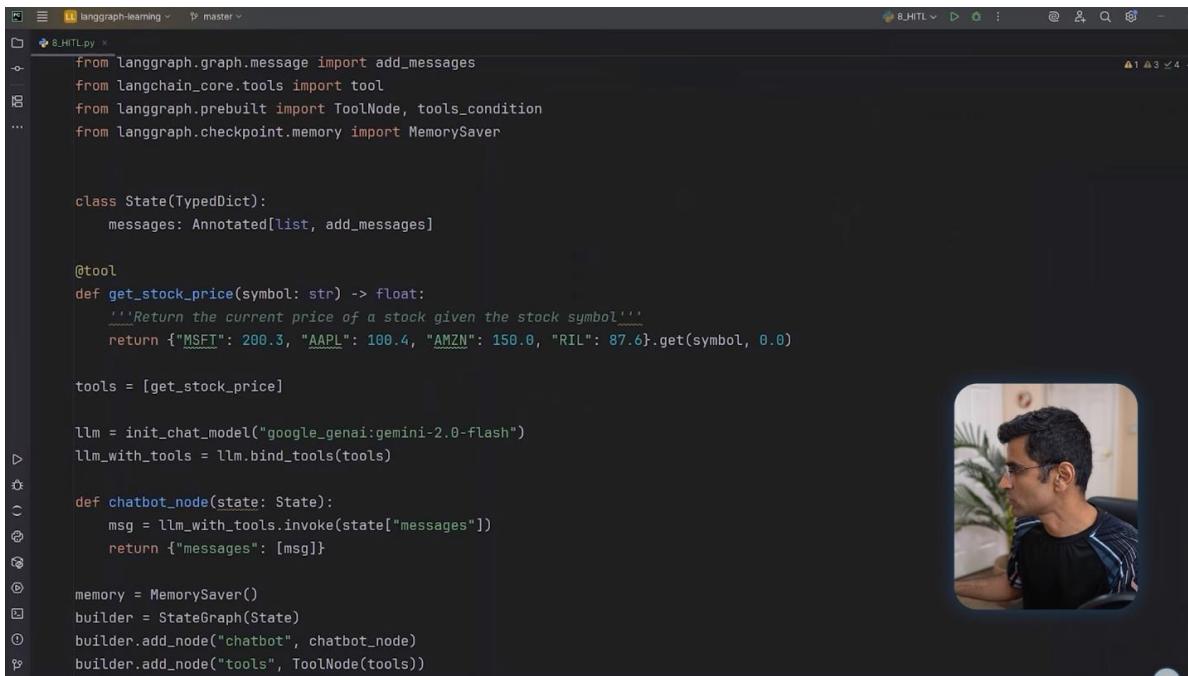
```
from langgraph.graph.message import add_messages
from langchain_core.tools import tool
from langgraph.prebuilt import ToolNode, tools_condition
from langgraph.checkpoint.memory import MemorySaver

class State(TypedDict):
    messages: Annotated[list, add_messages]

@tool
def get_stock_price(symbol: str) -> float:
    '''Return the current price of a stock given the symbol'''
    return {"MSFT": 200.3, "AAPL": 100.4, "AMZN": 1500.6}.get(symbol)

tools = [get_stock_price]

llm = init_chat_model("google_genai:gemini-2.0-flash")
```



```
from langgraph.graph.message import add_messages
from langchain_core.tools import tool
from langgraph.prebuilt import ToolNode, tools_condition
from langgraph.checkpoint.memory import MemorySaver

class State(TypedDict):
    messages: Annotated[list, add_messages]

@tool
def get_stock_price(symbol: str) -> float:
    """Return the current price of a stock given the stock symbol"""
    return {"MSFT": 200.3, "AAPL": 100.4, "AMZN": 150.0, "RIL": 87.6}.get(symbol, 0.0)

tools = [get_stock_price]

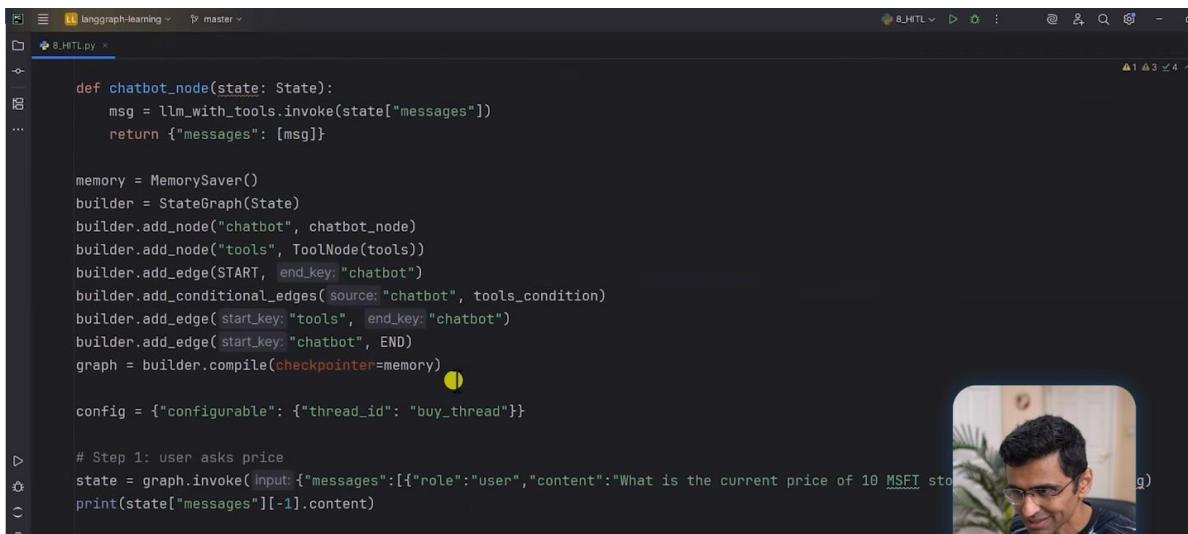
llm = init_chat_model("google_genai:gemini-2.0-flash")
llm_with_tools = llm.bind_tools(tools)

def chatbot_node(state: State):
    msg = llm_with_tools.invoke(state["messages"])
    return {"messages": [msg]}

memory = MemorySaver()
builder = StateGraph(State)
builder.add_node("chatbot", chatbot_node)
builder.add_node("tools", ToolNode(tools))
builder.add_edge(START, end_key="chatbot")
builder.add_conditional_edges(source="chatbot", tools_condition)
builder.add_edge(start_key="tools", end_key="chatbot")
builder.add_edge(start_key="chatbot", END)
graph = builder.compile(checkpointer=memory)

config = {"configurable": {"thread_id": "buy_thread"}}

# Step 1: user asks price
state = graph.invoke(input={"messages": [{"role": "user", "content": "What is the current price of 10 MSFT stocks?"}], config=config)
print(state["messages"][-1].content)
```

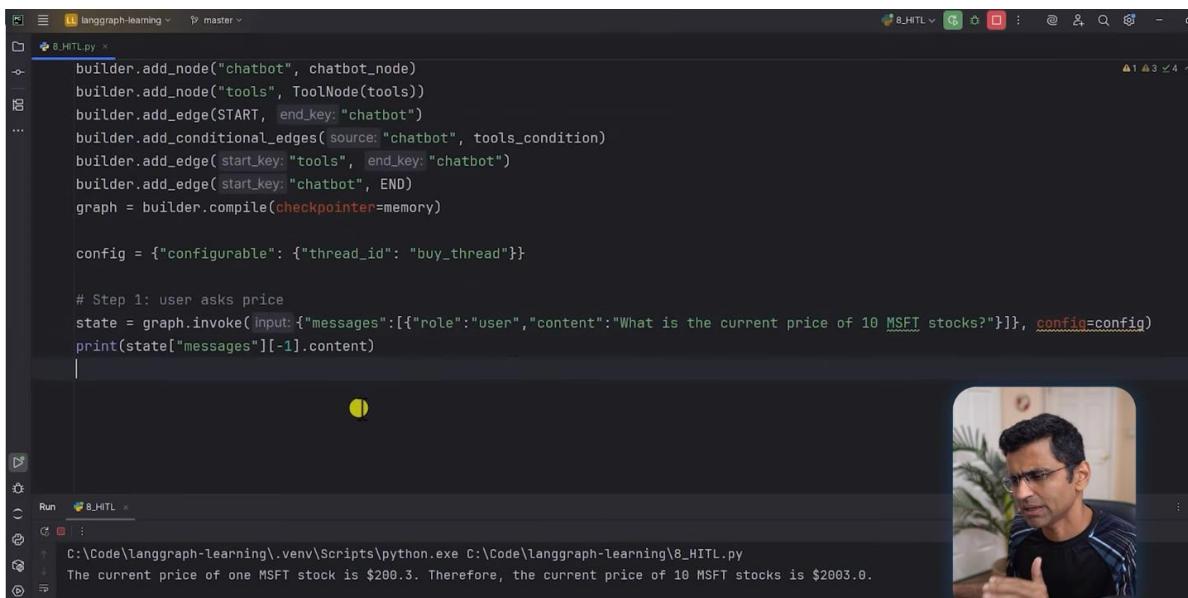


```
def chatbot_node(state: State):
    msg = llm_with_tools.invoke(state["messages"])
    return {"messages": [msg]}

memory = MemorySaver()
builder = StateGraph(State)
builder.add_node("chatbot", chatbot_node)
builder.add_node("tools", ToolNode(tools))
builder.add_edge(START, end_key="chatbot")
builder.add_conditional_edges(source="chatbot", tools_condition)
builder.add_edge(start_key="tools", end_key="chatbot")
builder.add_edge(start_key="chatbot", END)
graph = builder.compile(checkpointer=memory)

config = {"configurable": {"thread_id": "buy_thread"}}

# Step 1: user asks price
state = graph.invoke(input={"messages": [{"role": "user", "content": "What is the current price of 10 MSFT stocks?"}], config=config)
print(state["messages"][-1].content)
```



```
builder.add_node("chatbot", chatbot_node)
builder.add_node("tools", ToolNode(tools))
builder.add_edge(START, end_key="chatbot")
builder.add_conditional_edges(source="chatbot", tools_condition)
builder.add_edge(start_key="tools", end_key="chatbot")
builder.add_edge(start_key="chatbot", END)
graph = builder.compile(checkpointer=memory)

config = {"configurable": {"thread_id": "buy_thread"}}

# Step 1: user asks price
state = graph.invoke(input={"messages": [{"role": "user", "content": "What is the current price of 10 MSFT stocks?"}], config=config)
print(state["messages"][-1].content)
```



We can get a stock price using a tool successfully. Next, let us add a new tool that can buy a stock as below



```

class State(TypedDict):
    messages: Annotated[list, add_messages]

@tool
def get_stock_price(symbol: str) -> float:
    '''Return the current price of a stock given the stock symbol'''
    return {"MSFT": 200.3, "AAPL": 100.4, "AMZN": 150.0, "RIL": 87.6}.get(symbol, 0.0)

@tool
def buy_stocks(symbol: str, quantity: int, total_price: float) -> str:
    '''Buy stocks given the stock symbol and quantity'''
    return f"You bought {quantity} shares of {symbol} for a total price of {total_price}"

tools = [get_stock_price, buy_stocks]

llm = init_chat_model("google_genai:gemini-2.0-flash")
llm_with_tools = llm.bind_tools(tools)

def chatbot_node(state: State):
    msg = llm_with_tools.invoke(state["messages"])
    return {"messages": [msg]}

memory = MemorySaver()
builder = StateGraph(State)
builder.add_node("chatbot", chatbot_node)

```



```

graph = builder.compile(checkpointer=memory)

config = {"configurable": {"thread_id": "buy_thread"}}

# Step 1: user asks price
state = graph.invoke(input={"messages": [{"role": "user", "content": "What is the current price of 10 MSFT stocks?"}], config=config)
print(state["messages"][-1].content)

# Step 2: user asks to buy
state = graph.invoke(input={"messages": [{"role": "user", "content": "Buy 10 MSFT stocks at current price."}], config=config)
print(state["messages"][-1].content)

```

We can now buy stocks successfully. Next, let us add a human-in-the-loop approval using an `interrupt()` function before the BUY gets done as below

## Add human-in-the-loop controls

Agents can be unreliable and may need human input to successfully accomplish tasks. Similarly, for some actions, you may want to require human approval before running to ensure that everything is running as intended.

LangGraph's `persistence` layer supports **human-in-the-loop** workflows, allowing execution to pause and resume based on user feedback. The primary interface to this functionality is the `interrupt` function. Calling `interrupt` inside a node will pause execution. Execution can be resumed, together with new input from a human-in-the-loop `Command`. `interrupt` is ergonomically similar to Python's built-in `input()`, with some caveats.



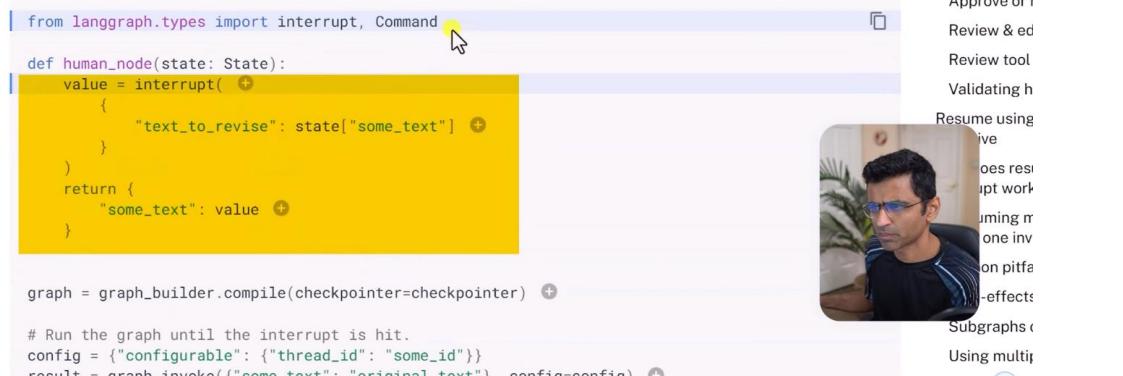
### 1. Add the `human_assistance` tool

Table of contents
1. Add the human assistance tool
2. Compile the graph
3. Visualize the graph
4. Prompt the user for input
5. Resume execution
Next steps

like approvals, edits, or gathering additional context.

The graph is resumed using a `Command` object that provides the human's response.

API Reference: [interrupt](#) | [Command](#)



```
from langgraph.types import interrupt, Command

def human_node(state: State):
    value = interrupt(
        {
            "text_to_revise": state["some_text"]
        }
    )
    return {
        "some_text": value
    }

graph = graph_builder.compile(checkpointer=checkpointer)

# Run the graph until the interrupt is hit.
config = {"configurable": {"thread_id": "some_id"}}
result = graph.invoke({"some_text": "original text"}, config=config)
```

Table of contents

interrupt

Requirements

Design patterns

Approve or edit

Review & edit

Review tool

Validating humans

Resume using previous

does resume

interrupt work

suming multiple

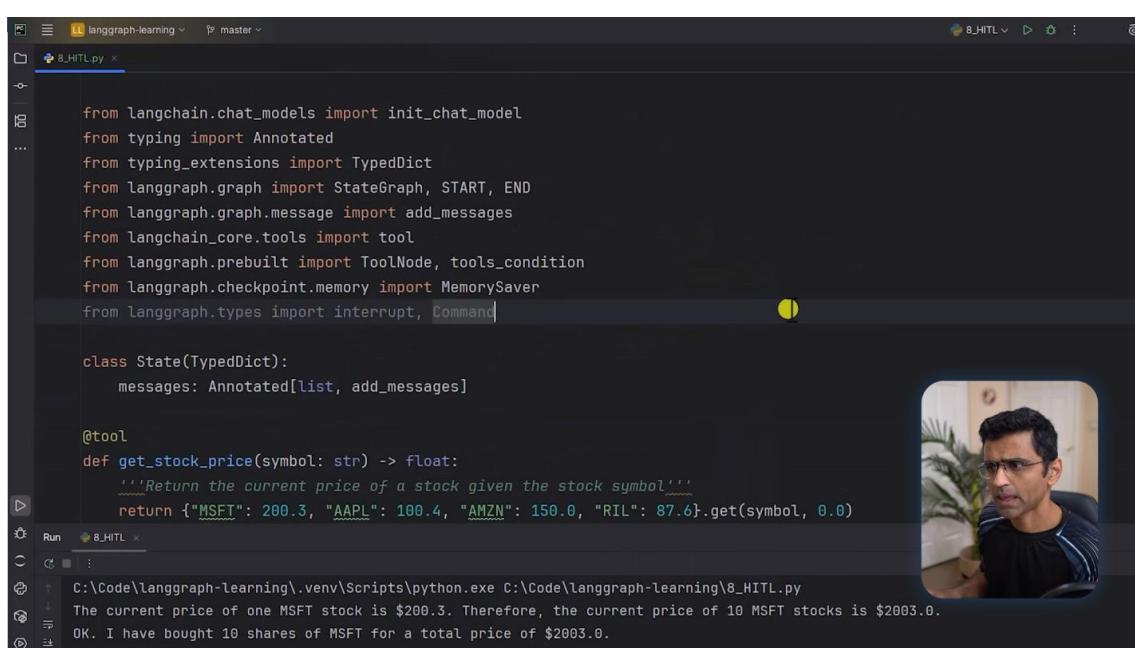
one invocation

on pitfalls

-effects

Subgraphs

Using multip

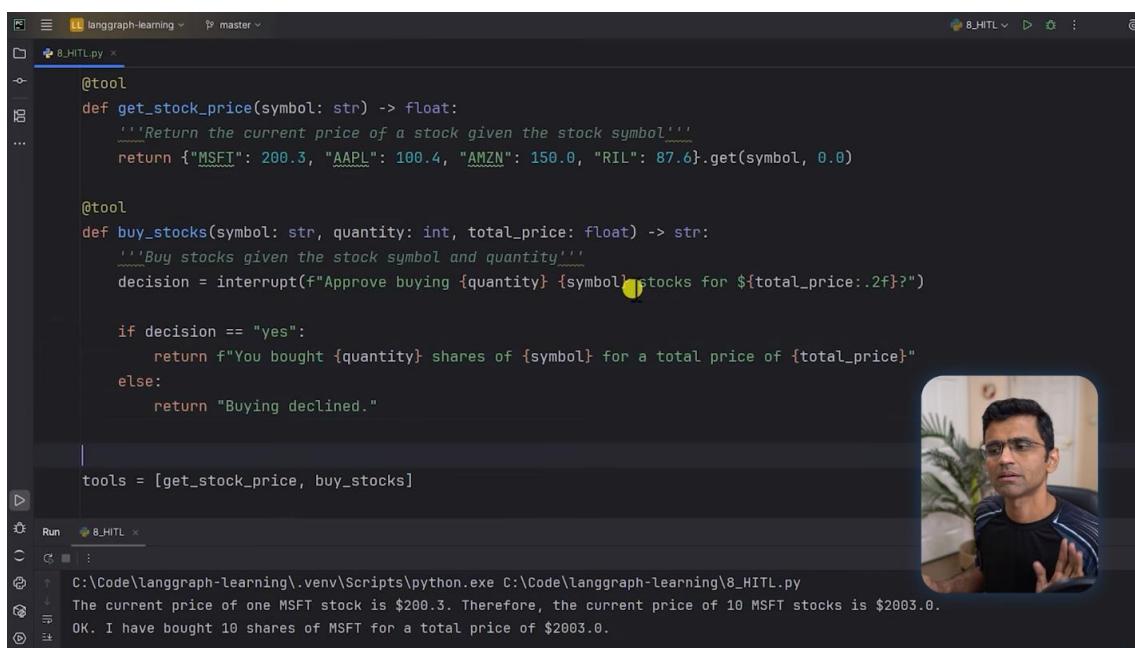


```
from langchain.chat_models import init_chat_model
from typing import Annotated
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages
from langchain_core.tools import tool
from langgraph.prebuilt import ToolNode, tools_condition
from langgraph.checkpoint.memory import MemorySaver
from langgraph.types import interrupt, Command

class State(TypedDict):
    messages: Annotated[list, add_messages]

@tool
def get_stock_price(symbol: str) -> float:
    '''Return the current price of a stock given the stock symbol'''
    return {"MSFT": 200.3, "AAPL": 100.4, "AMZN": 150.0, "RIL": 87.6}.get(symbol, 0.0)

Run 8_HITL x
C:\Code\langgraph-learning\.venv\Scripts\python.exe C:\Code\langgraph-learning\8_HITL.py
The current price of one MSFT stock is $200.3. Therefore, the current price of 10 MSFT stocks is $2003.0.
OK. I have bought 10 shares of MSFT for a total price of $2003.0.
```



```
@tool
def get_stock_price(symbol: str) -> float:
    '''Return the current price of a stock given the stock symbol'''
    return {"MSFT": 200.3, "AAPL": 100.4, "AMZN": 150.0, "RIL": 87.6}.get(symbol, 0.0)

@tool
def buy_stocks(symbol: str, quantity: int, total_price: float) -> str:
    '''Buy stocks given the stock symbol and quantity'''
    decision = interrupt(f"Approve buying {quantity} {symbol} stocks for ${total_price}?")


    if decision == "yes":
        return f"You bought {quantity} shares of {symbol} for a total price of {total_price}"
    else:
        return "Buying declined."


tools = [get_stock_price, buy_stocks]

Run 8_HITL x
C:\Code\langgraph-learning\.venv\Scripts\python.exe C:\Code\langgraph-learning\8_HITL.py
The current price of one MSFT stock is $200.3. Therefore, the current price of 10 MSFT stocks is $2003.0.
OK. I have bought 10 shares of MSFT for a total price of $2003.0.
```



```

config = {"configurable": {"thread_id": "buy_thread"}}

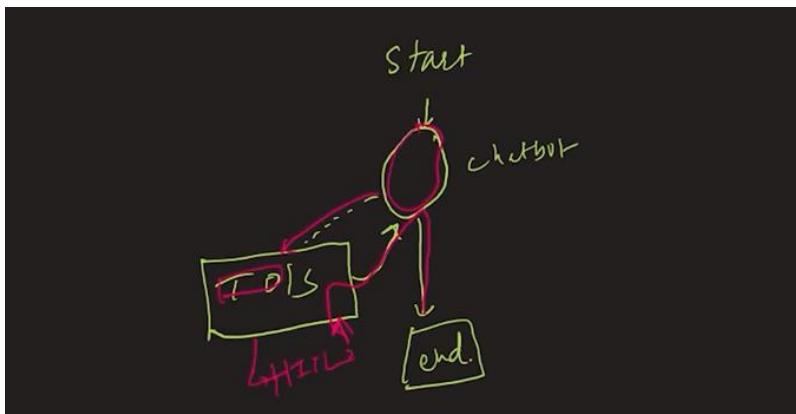
# Step 1: user asks price
state = graph.invoke(input: {"messages": [{"role": "user", "content": "What is the current price of 10 MSFT stocks?"}]}, config=config)
print(state["messages"][-1].content)

# Step 2: user asks to buy
state = graph.invoke(input: {"messages": [{"role": "user", "content": "Buy 10 MSFT stocks at current price."}]}, config=config)
print(state.get("__interrupt__"))

```

This tells us that the graph execution is correctly interrupted as intended

## Human in the loop - HITL



```

builder.add_edge(START, end_key="chatbot")
builder.add_conditional_edges(source="chatbot", tools_condition)
builder.add_edge(start_key="tools", end_key="chatbot")
builder.add_edge(start_key="chatbot", END)
graph = builder.compile(checkpointer=memory)

config = {"configurable": {"thread_id": "buy_thread"}}

# Step 1: user asks price
state = graph.invoke(input: {"messages": [{"role": "user", "content": "What is the current price of 10 MSFT stocks?"}]}, config=config)
print(state["messages"][-1].content)

# Step 2: user asks to buy
state = graph.invoke(input: {"messages": [{"role": "user", "content": "Buy 10 MSFT stocks at current price."}]}, config=config)
print(state.get("__interrupt__"))

decision = input("Approve (yes/no): ")
graph.invoke(Command(resume=decision), config=config)

```

Command will take the HIL reply and feed it back as the decision output value in interrupt as below

```
langgraph-learning master 8_HITL.py

@tool
def get_stock_price(symbol: str) -> float:
    '''Return the current price of a stock given the stock symbol'''
    return {"MSFT": 200.3, "AAPL": 100.4, "AMZN": 150.0, "RIL": 87.6}.get(symbol, 0.0)

@tool
def buy_stocks(symbol: str, quantity: int, total_price: float) -> str:
    '''Buy stocks given the stock symbol and quantity'''
    decision = interrupt(f"Approve buying {quantity} {symbol} stocks for ${total_price}?")


    if decision == "yes":
        return f"You bought {quantity} shares of {symbol} for a total price of {total_price}"
    else:
        return "Buying declined."


tools = [get_stock_price, buy_stocks]

llm = init_chat_model("google_genai:gemini-2.0-flash")
llm_with_tools = llm.bind_tools(tools)

def chatbot_node(state: State):
    msg = llm_with_tools.invoke(state["messages"])
```



```
# Step 2: user asks to buy
state = graph.invoke(input={"messages":[{"role":"user","content":"Buy 10 MSFT stocks at current price."}]}, config=config)
print(state.get("__interrupt__"))

decision = input("Approve (yes/no): ")
state = graph.invoke(Command(resume=decision), config=config)
print(state["messages"][-1].content)
```



```
Run 8_HITL x
G : 

C: C:\Code\langgraph-learning\.venv\Scripts\python.exe C:\Code\langgraph-learning\8_HITL.py
The current price of one MSFT stock is $200.3. Therefore, the current price of 10 MSFT stocks is $2003.0
[Interrupt(value='Approve buying 10 MSFT stocks for $2003.00?', resumable=True, ns=['tools:def9b365-26c4-432f-9e0d-1432f1a432'])]
Approve (yes/no): yes
OK. You bought 10 shares of MSFT for a total price of $2003.0.
Process finished with exit code 0
```

```
Run 8_HITL x
G : 

C: C:\Code\langgraph-learning\.venv\Scripts\python.exe C:\Code\langgraph-learning\8_HITL.py
The current price of one MSFT stock is $200.3. Therefore, the current price of 10 MSFT stocks is $2003.0
[Interrupt(value='Approve buying 10 MSFT stocks for $2003.00?', resumable=True, ns=['tools:e5342d26-6e2c-4b57-9a0d-1b57'])]
Approve (yes/no): no
I was not able to buy the stocks. Buying declined.
```

