Langgraph Tutorial | Agentic AI crash course

codebasics
1.31M subscribers

Subscribed | 195 | Share | Thanks | Clip | ...

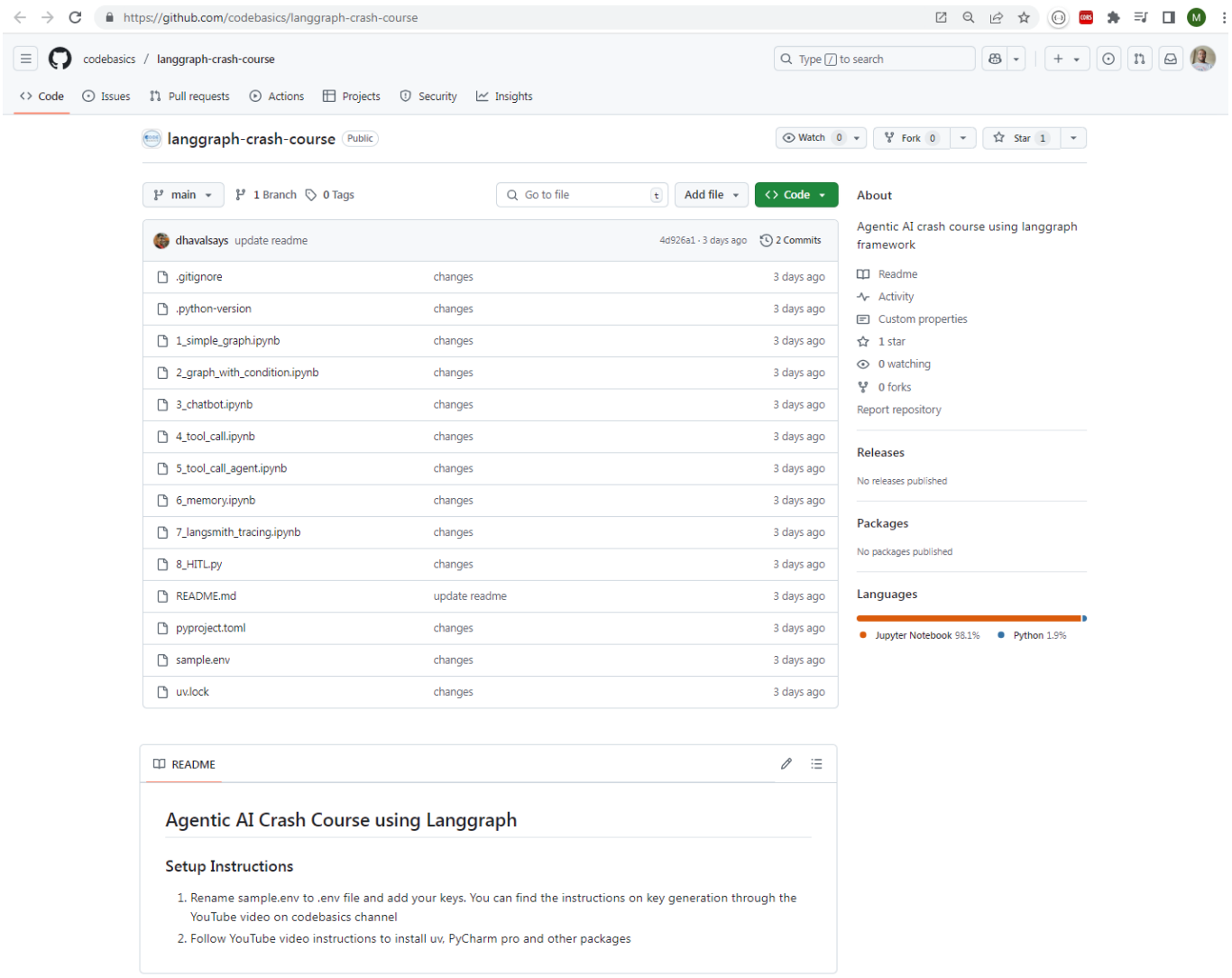
3,591 views Premiered 8 hours ago

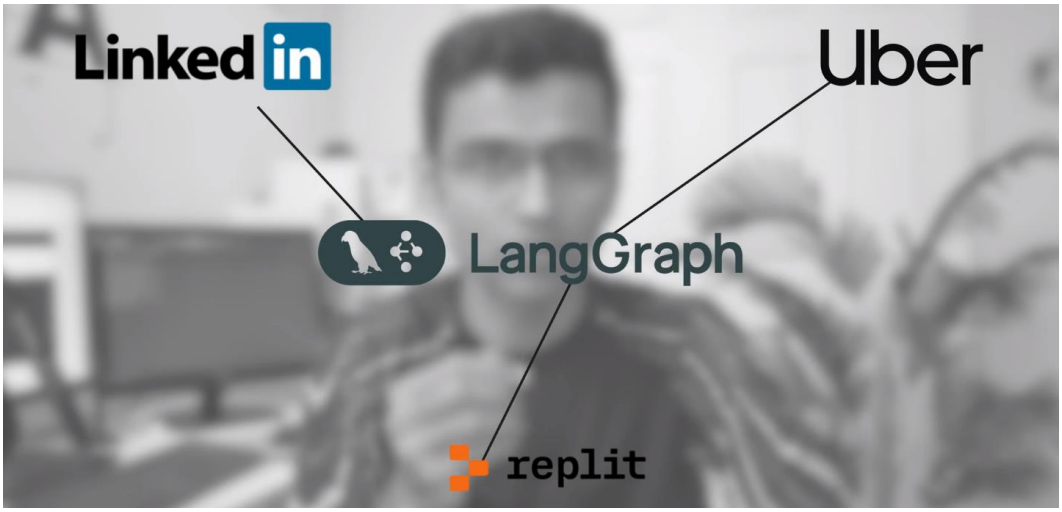
Learn Agentic AI using a popular framework langgraph. In this video, we will start with agentic AI basics and then we will dive deeper into langgraph by covering a wide range of topics.

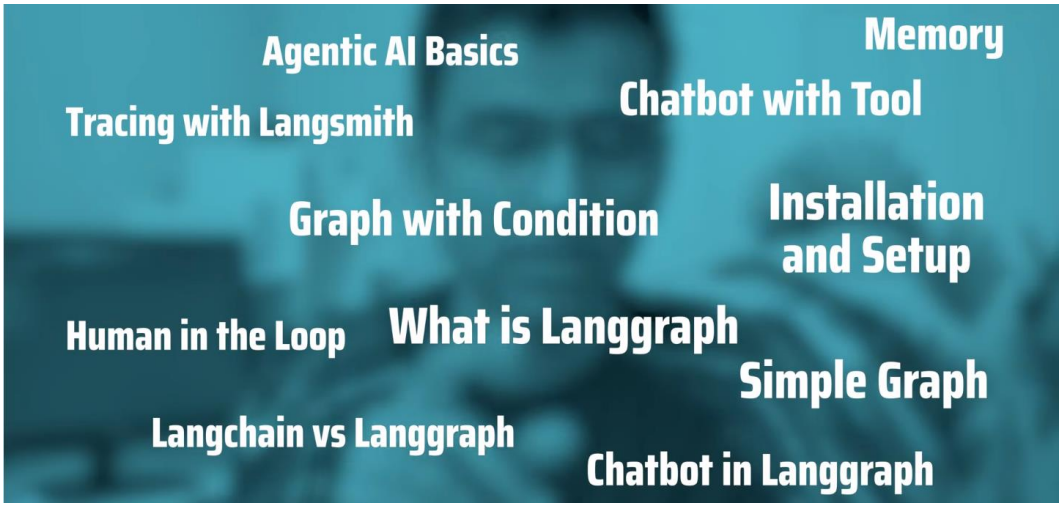
☞ Check out PyCharm, the IDE built for data science and AI/ML professionals: https://jb.gg/try-pycharm-now
☞ Free 3-Month PyCharm Pro subscription with coupon code "PyCharm2codebasics": https://jb.gg/pc_codebasics
Redeem on new purchases before Feb 01, 2026

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

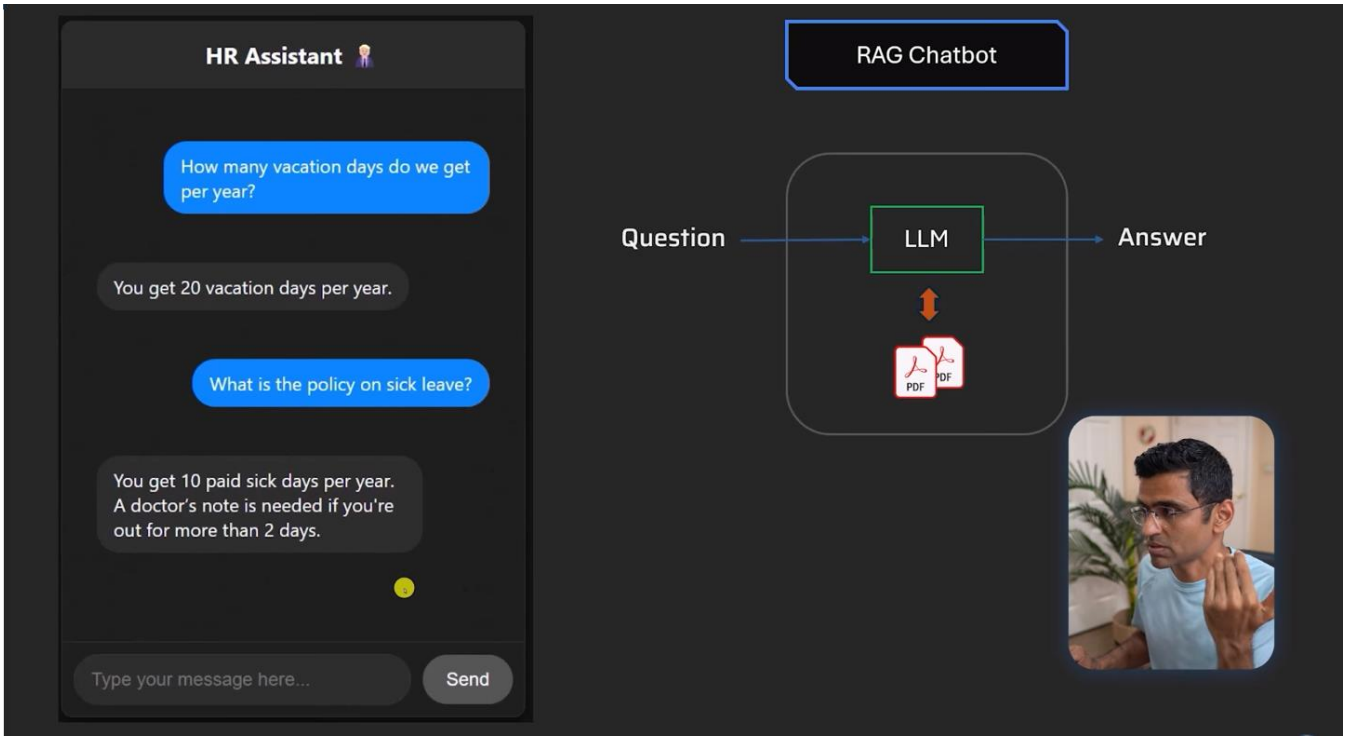
00:00 Introduction
00:40 Agentic AI Basics
00:00 What is Langgraph
07:41 Langchain vs Langgraph
13:08 Installation and Setup
19:00 Simple Graph
31:36 Graph with Condition
36:55 Chatbot in Langgraph
45:38 Chatbot with Tool
56:03 Memory
1:05:05 Tracing with Langsmith
1:10:11 Human in the Loop

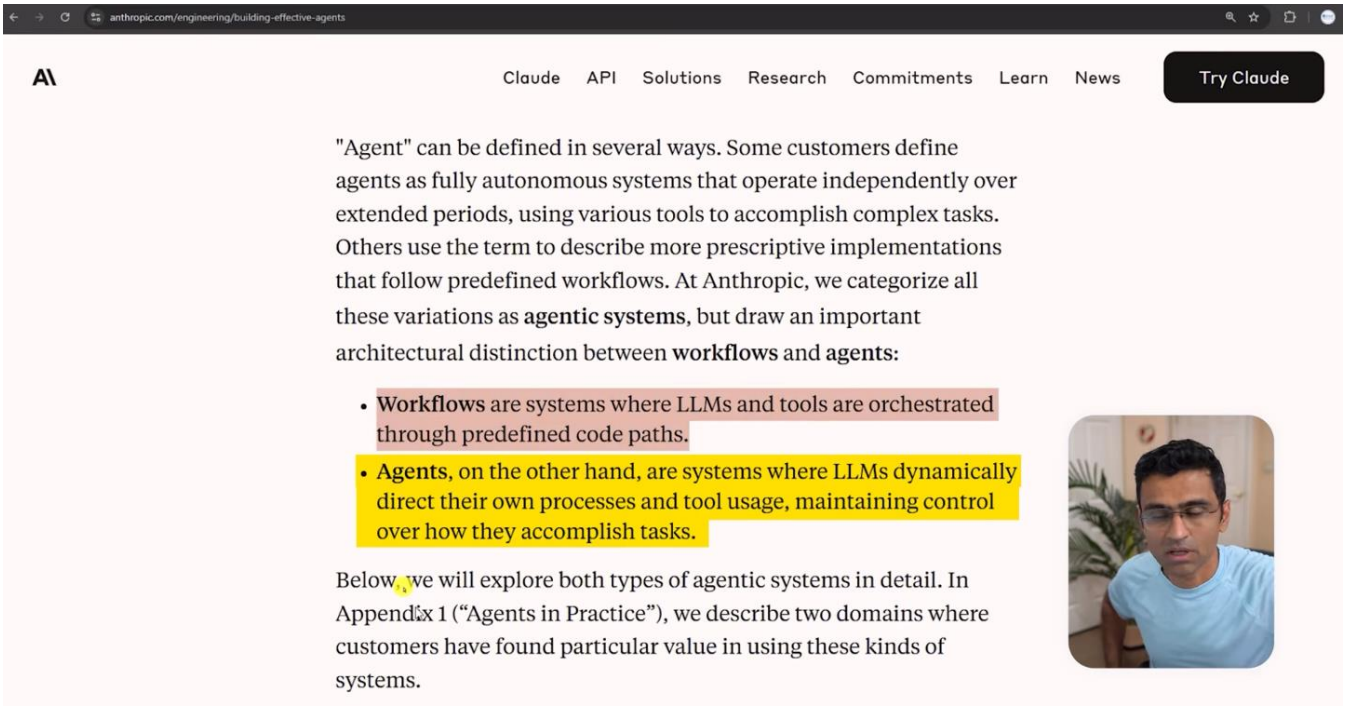


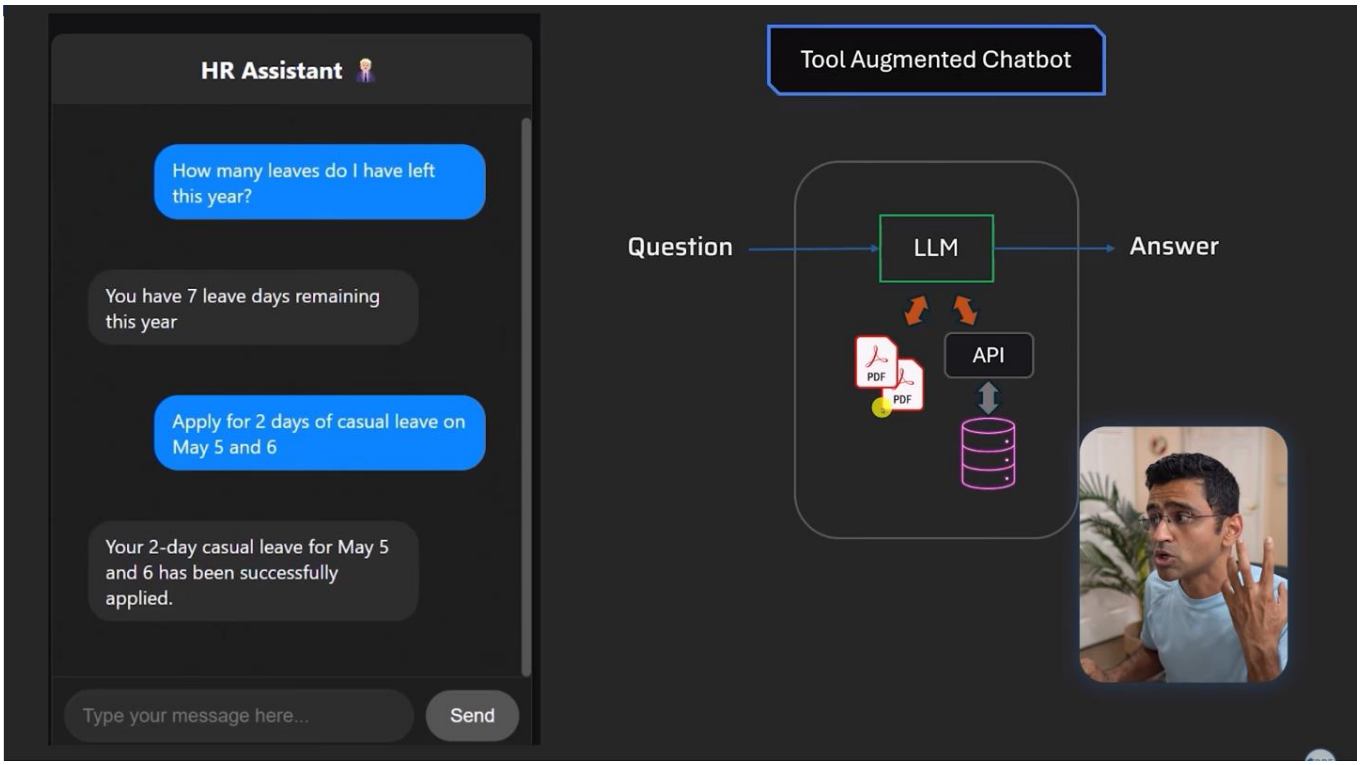


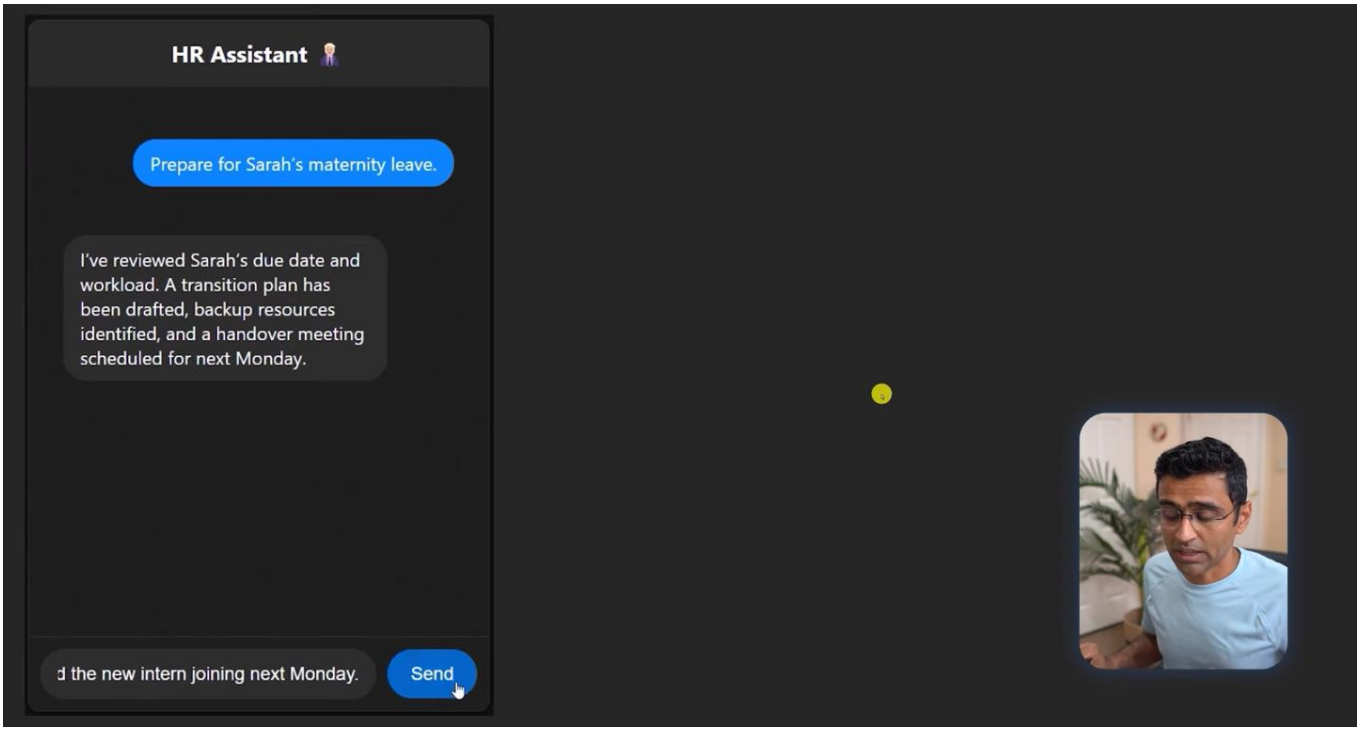


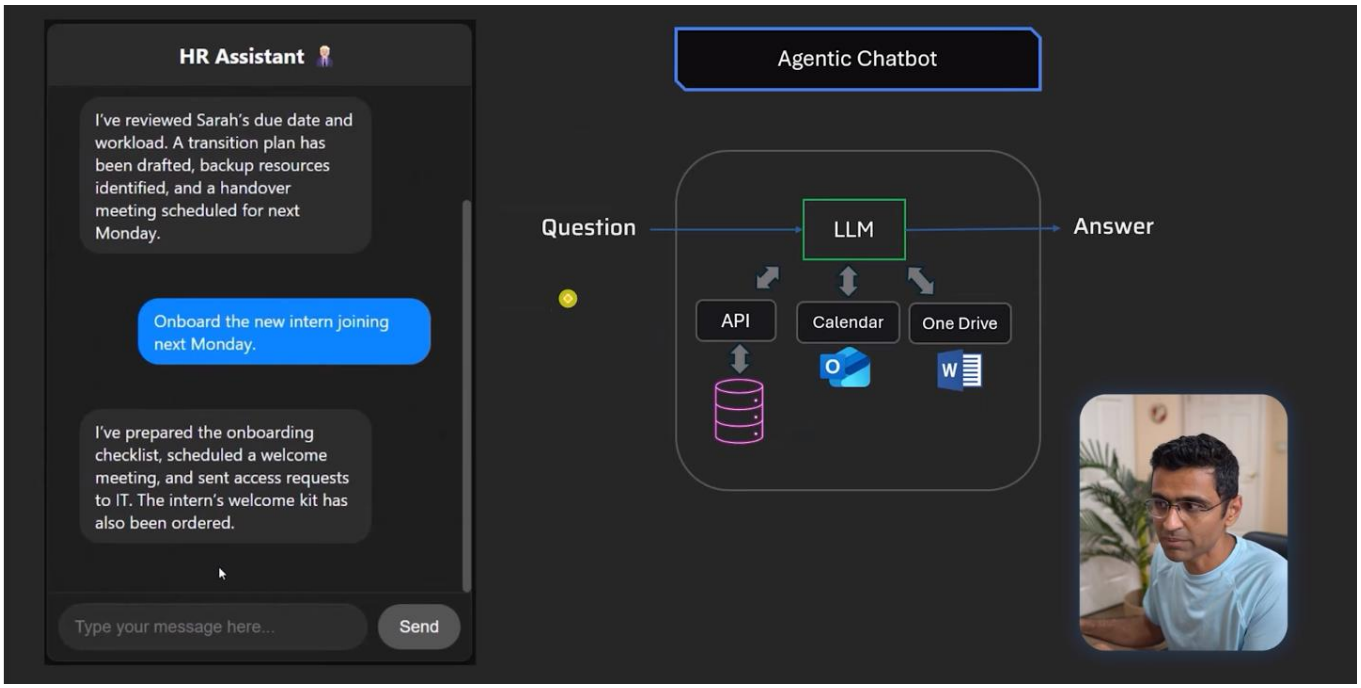
What is an AI Agent

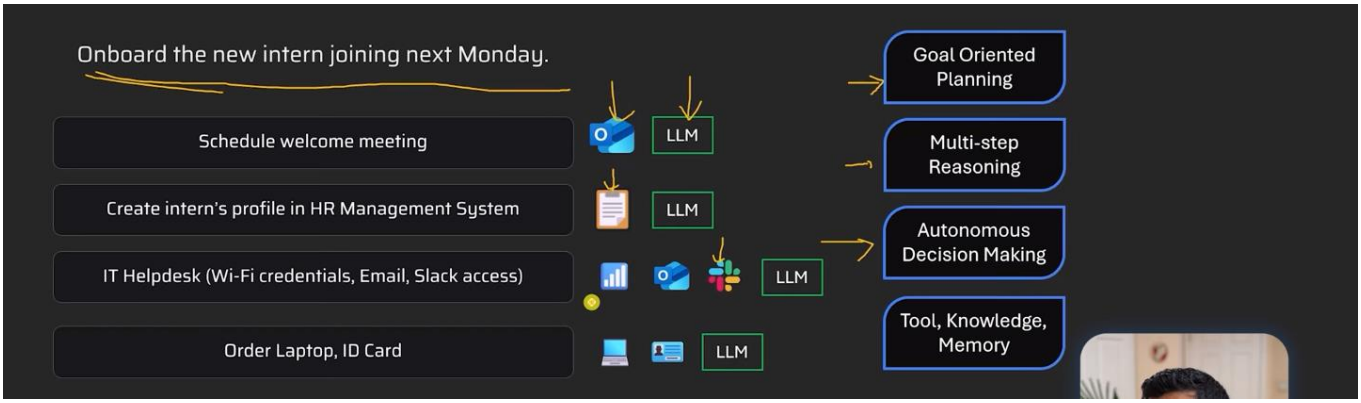

HR Assistant
How many vacation days do we get per year?
You get 20 vacation days per year.
What is the policy on sick leave?
You get 10 paid sick days per year. A doctor's note is needed if you're out for more than 2 days.
Type your message here...
Send
RAG Chatbot
Question
LLM
Answer
PDF
PDF


anthropic.com/engineering/building-effective-agents
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
Question
LLM
Answer
API
PDF
PDF


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.
d the new intern joining next Monday.
Send



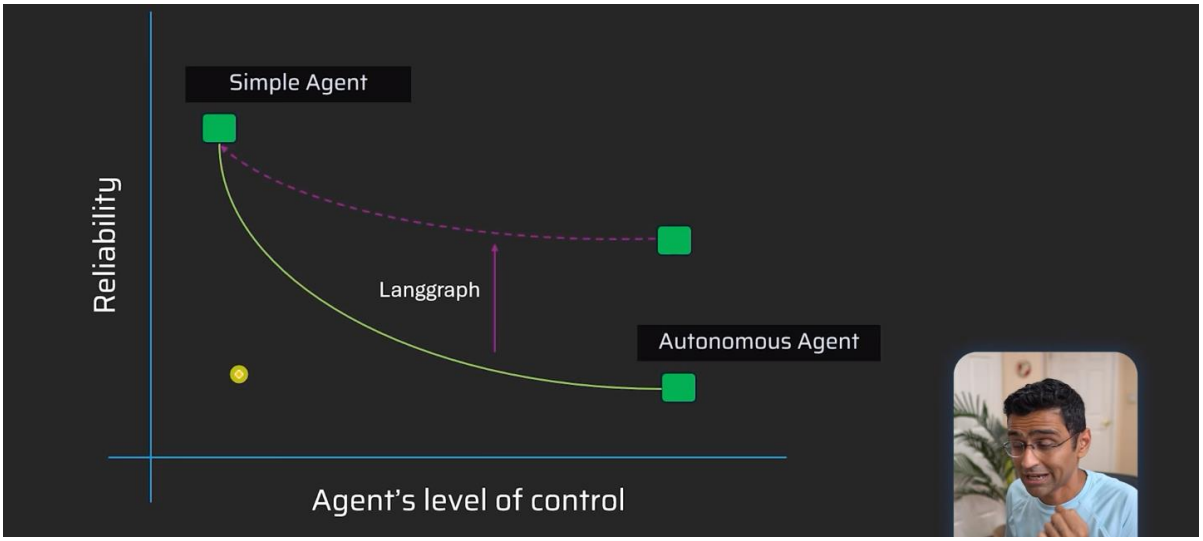


AI Agent can make decisions and take actions on its own to achieve a goal without being told exactly what to do at every step.

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

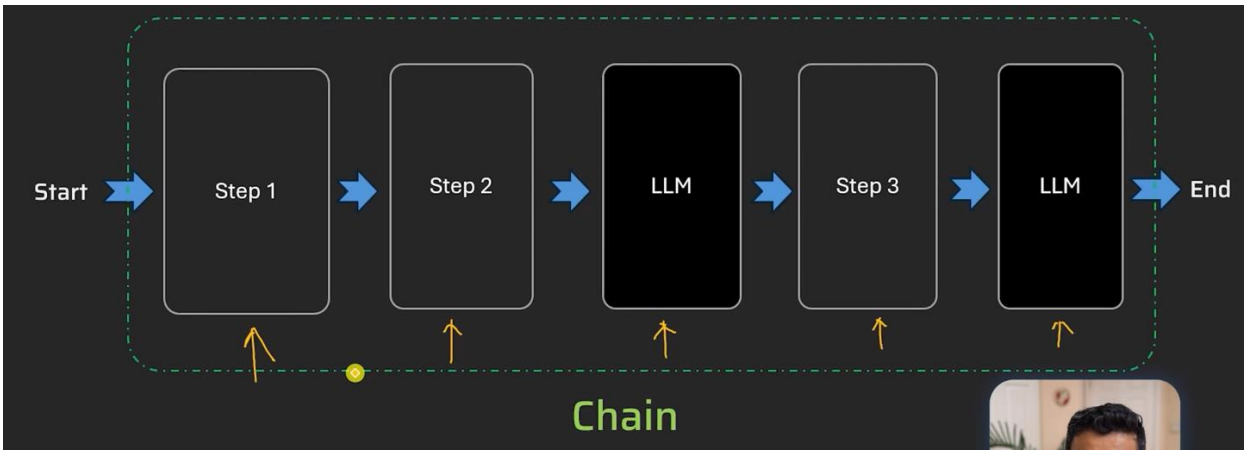
Langgraph excels at graph-based, stateful orchestration – e.g. multi-step, workflows with memory, streaming, human-in-the-loop control

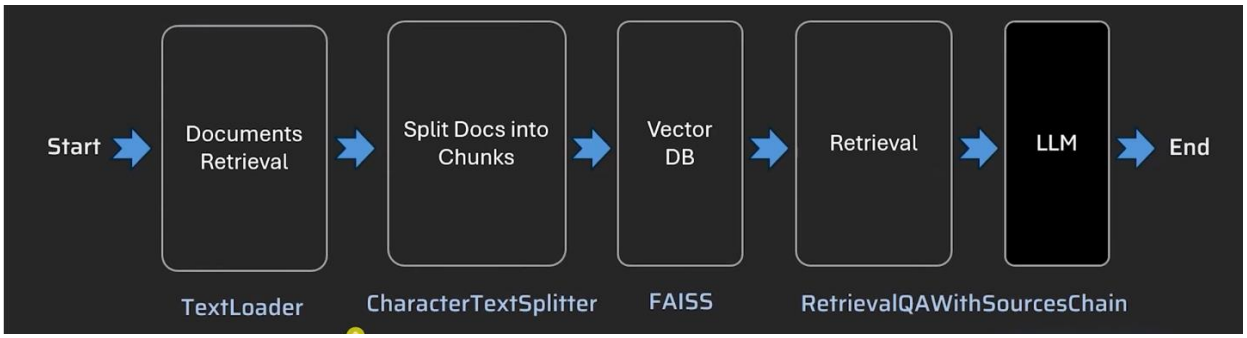
Other Frameworks

Agno Lightweight, fast

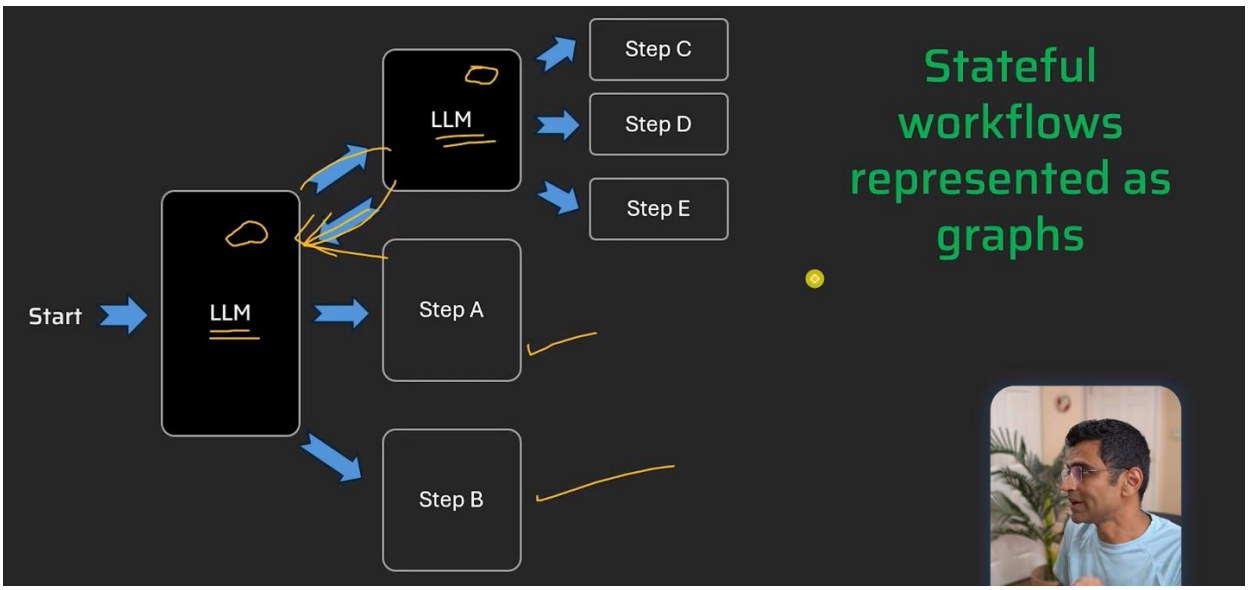
Google ADK Tight GCP, Multi-agent

Langchain vs Langgraph



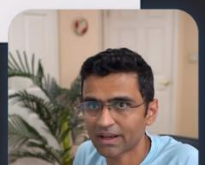


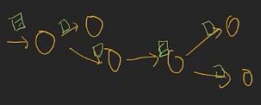
These are some of the classes and tools provided in the Langchain framework as utilities.

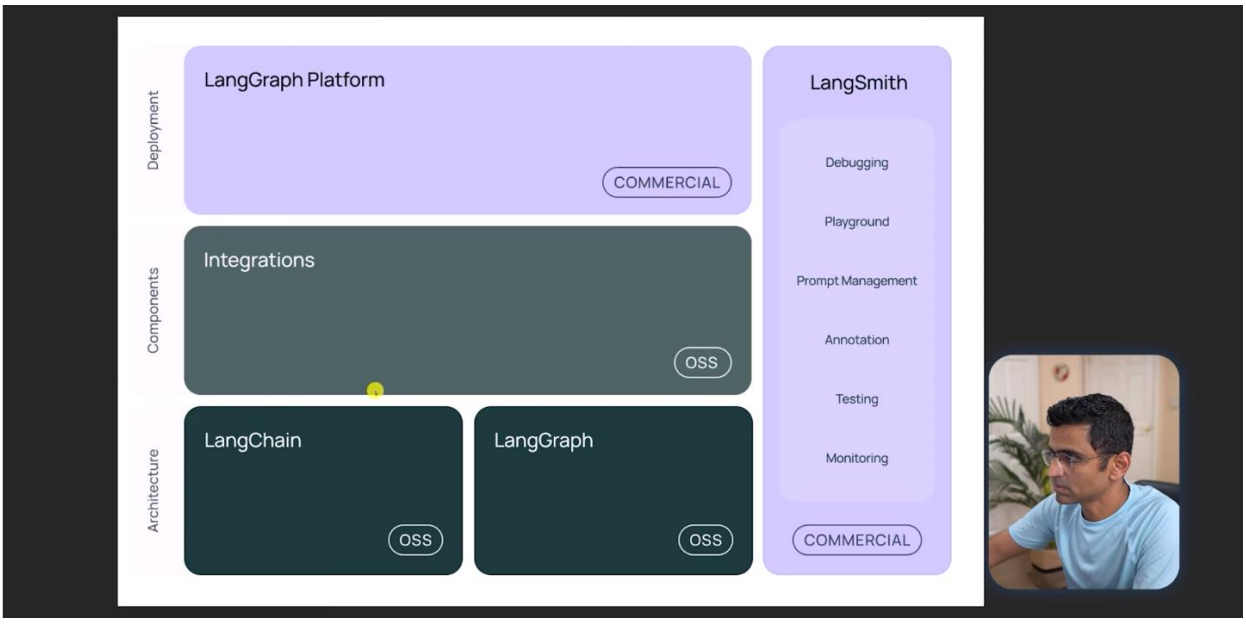


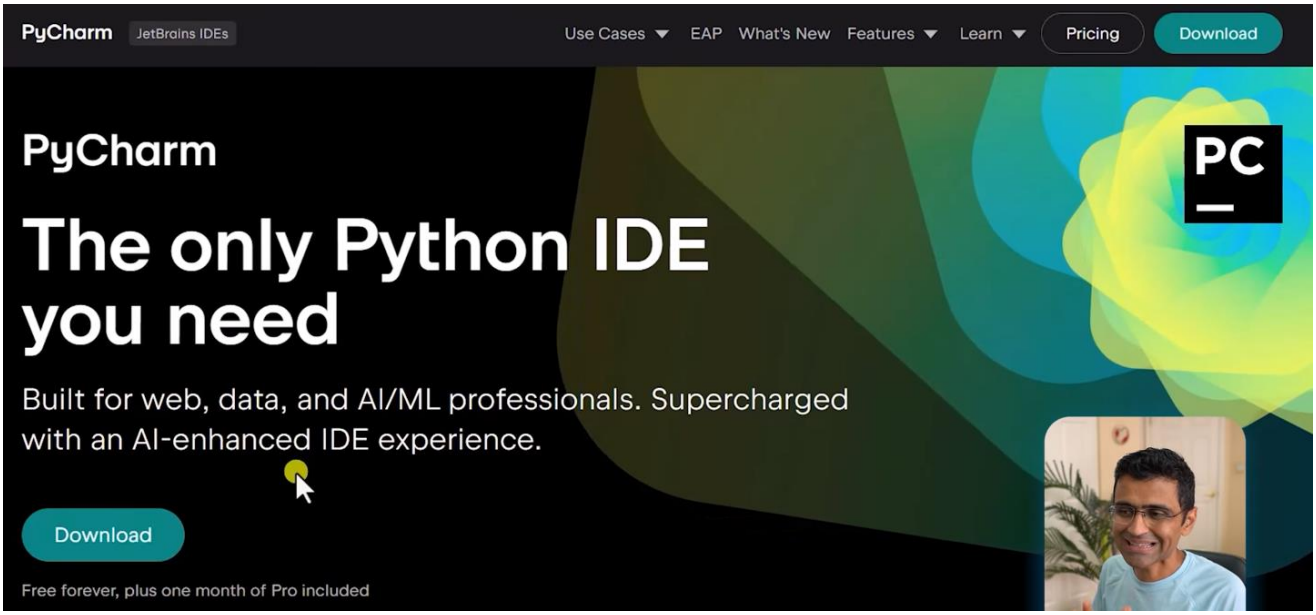
Autonomy is important here and Langgraph framework is best for these graph-like, stateful workflow solutions.

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)









JETBRAINS AI Developer Tools Team Tools Education Solutions Support Store

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

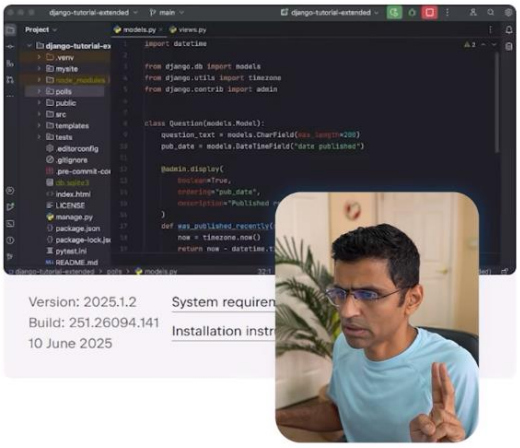
Windows macOS Linux

PyCharm Unified Product

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

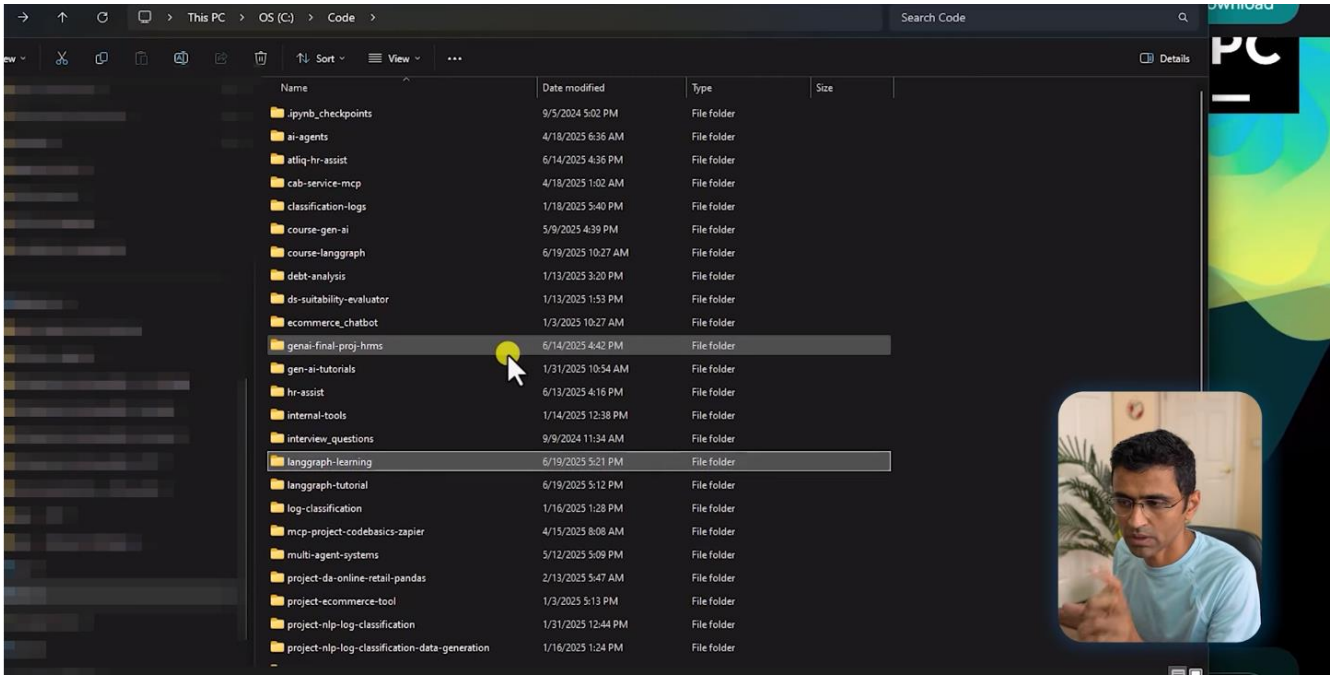
Thank you for downloading PyCharm!

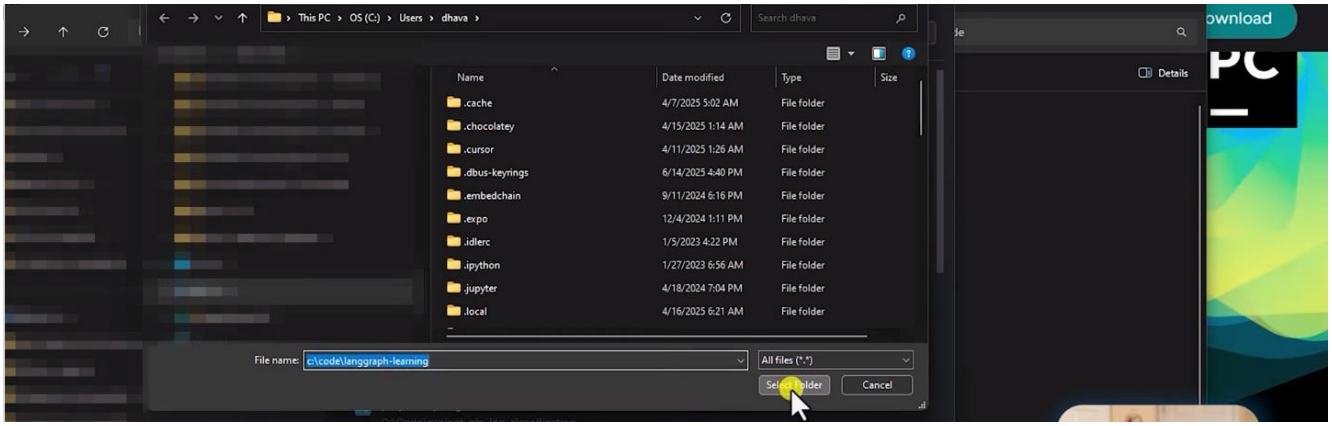
Your download should start shortly. If it doesn't, please use the direct link.

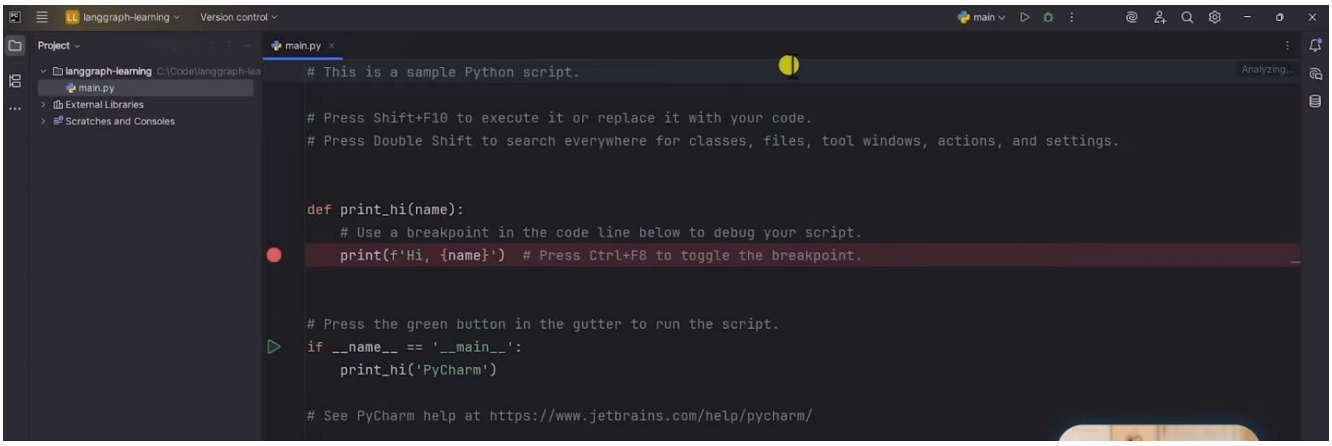
Download and verify the file SHA-256 checksum.

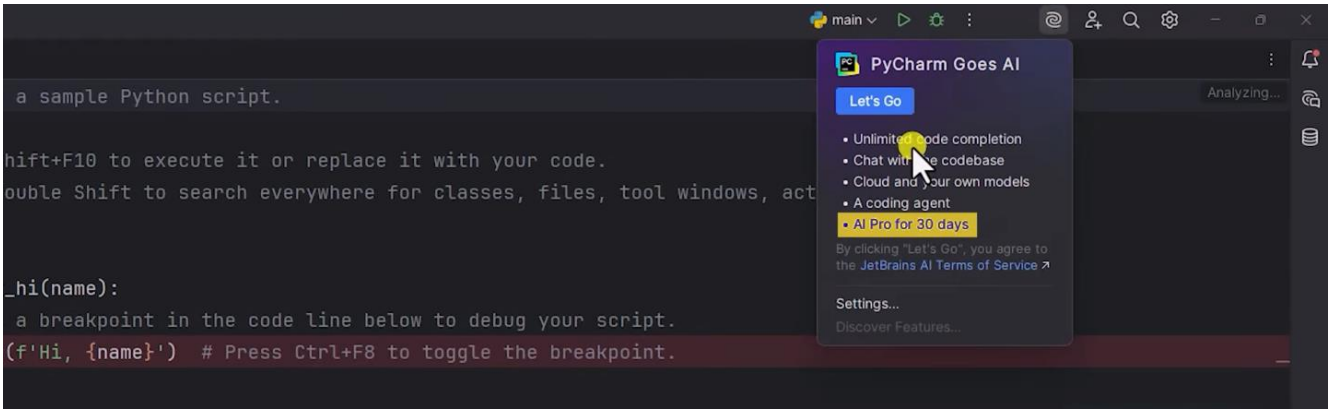
Learn more about the digital signatures of JetBrains binaries.

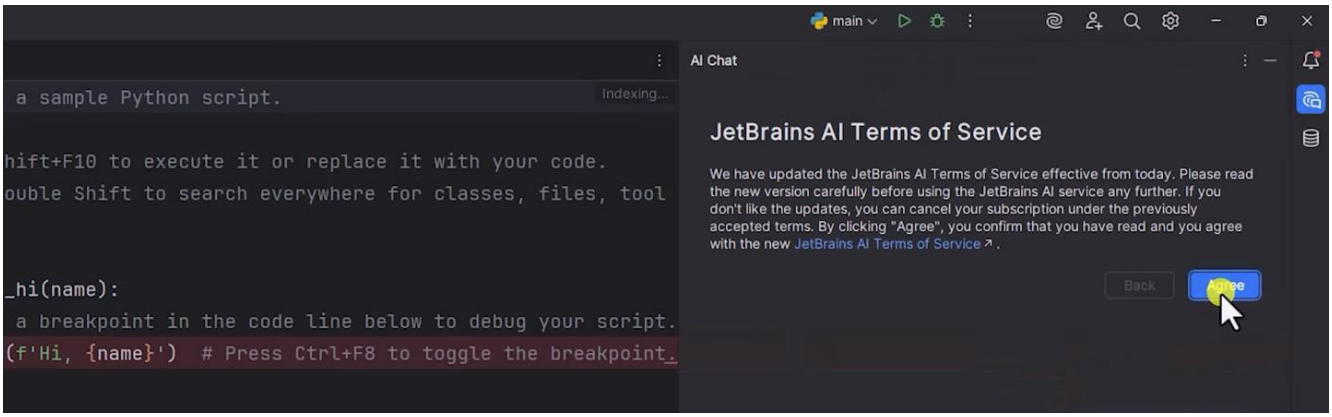
PyCharm goes AI! Less routine, more coding joy. All refined JetBrains AI tools — right in your IDE, for free. Get started

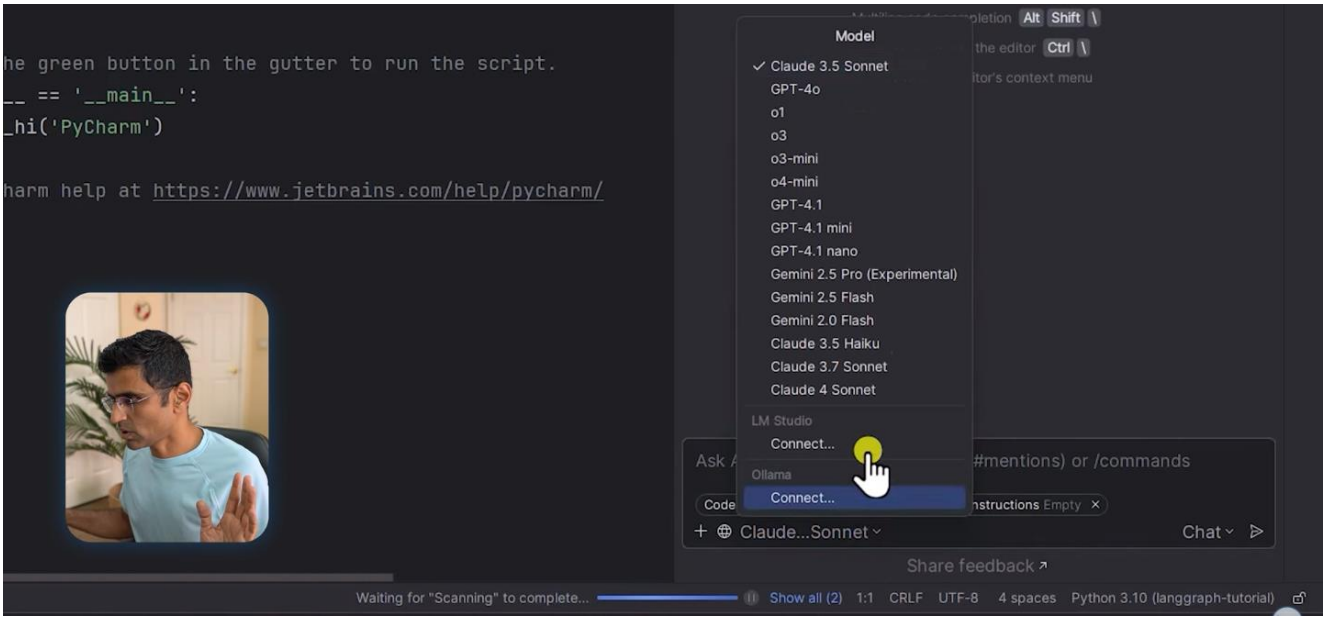


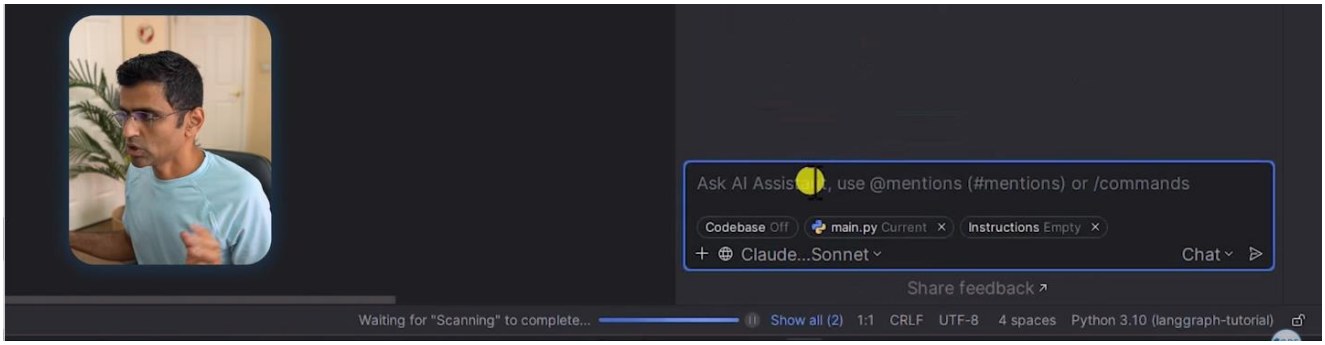

This PC › OS (C:) › Users › dhava ›
Search dhava
Name
Date modified
Type
Size
.cache 4/7/2025 5:02 AM File folder
.chocolatey 4/15/2025 1:14 AM File folder
.cursor 4/11/2025 1:26 AM File folder
.dbus-keyrings 6/14/2025 4:40 PM File folder
.embedchain 9/11/2024 6:16 PM File folder
.expo 12/4/2024 1:11 PM File folder
.idlerc 1/5/2023 4:22 PM File folder
.ipython 1/27/2023 6:56 AM File folder
.jupyter 4/18/2024 7:04 PM File folder
.local 4/16/2025 6:21 AM File folder
File name: c:\code\langgraph-learning
All files (*.*)
Cancel
Details

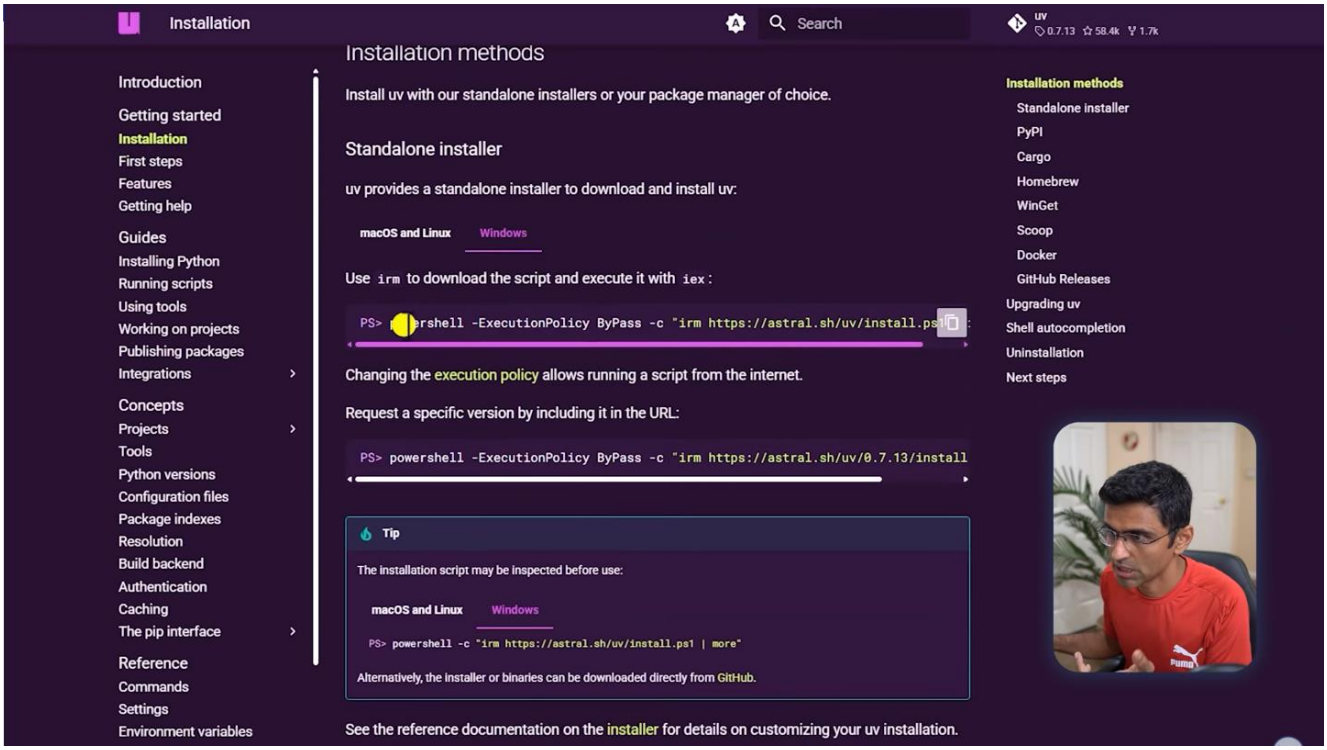

langgraph-learning
Version control
main
Project
main.py
langgraph-learning C:\Code\langgraph-lea
main.py
External Libraries
Scratches and Consoles
Analyzing...
# This is a sample Python script.
# Press Shift+F10 to execute it or replace it with your code.
# Press Double Shift to search everywhere for 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/

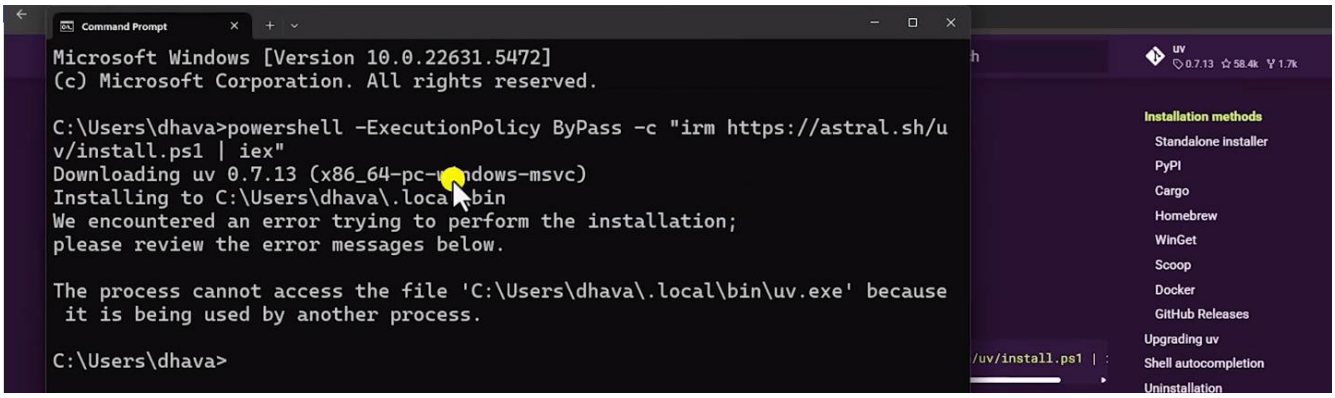

main
PyCharm Goes AI
Let's Go
• Unlimited code completion
• Chat with the codebase
• Cloud and your own models
• A coding agent
• AI Pro for 30 days
By clicking "Let's Go", you agree to the JetBrains AI Terms of Service
Settings...
Discover Features...
Analyzing...
a sample Python script.
hift+F10 to execute it or replace it with your code.
ouble Shift to search everywhere for classes, files, tool windows, act
_hi(name):
a breakpoint in the code line below to debug your script.
(f'Hi, {name}') # Press Ctrl+F8 to toggle the breakpoint.


main
AI Chat
JetBrains AI Terms of Service
We have updated the JetBrains AI Terms of Service effective from today. Please read the new version carefully before using the JetBrains AI service any further. If you don't like the updates, you can cancel your subscription under the previously accepted terms. By clicking "Agree", you confirm that you have read and you agree with the new JetBrains AI Terms of Service.
Back
Agree
Indexing...
a sample Python script.
hift+F10 to execute it or replace it with your code.
ouble Shift to search everywhere for classes, files, tool
_hi(name):
a breakpoint in the code line below to debug your script.
(f'Hi, {name}') # Press Ctrl+F8 to toggle the breakpoint.

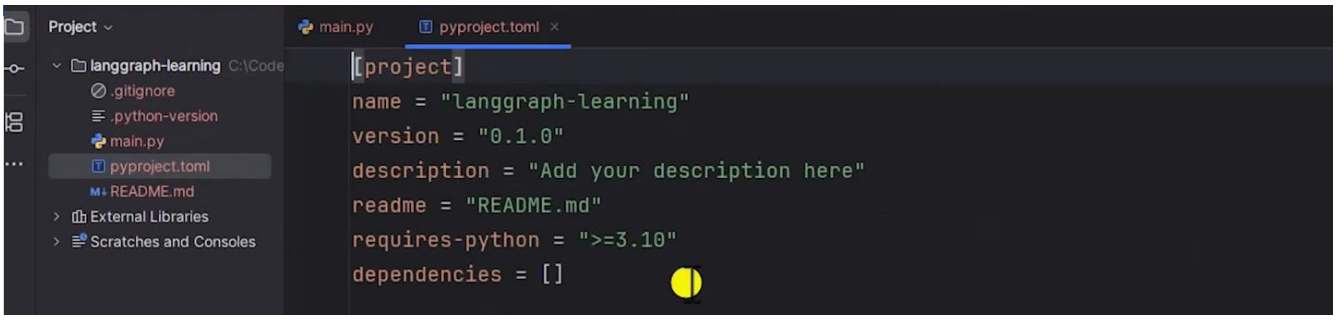

he green button in the gutter to run the script.
__ == '__main__':
_hi('PyCharm')
harm help at https://www.jetbrains.com/help/pycharm/
Model
Claude 3.5 Sonnet
GPT-4o
o1
o3
o3-mini
o4-mini
GPT-4.1
GPT-4.1 mini
GPT-4.1 nano
Gemini 2.5 Pro (Experimental)
Gemini 2.5 Flash
Gemini 2.0 Flash
Claude 3.5 Haiku
Claude 3.7 Sonnet
Claude 4 Sonnet
LM Studio
Connect...
Ollama
Connect...
#mentions) or /commands
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)

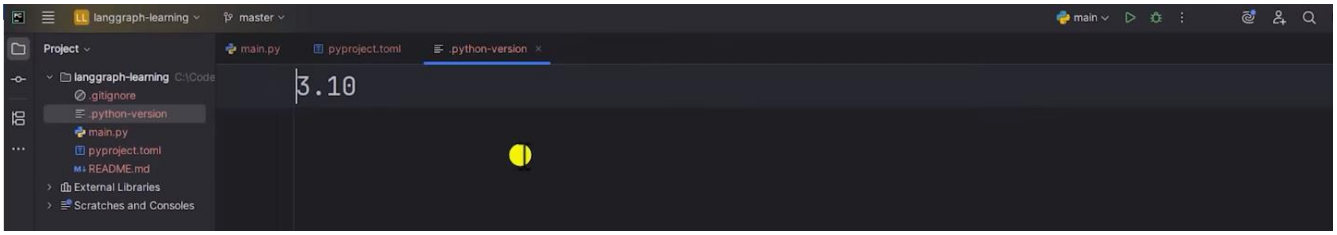

use @mentions (#mentions) or /commands
Codebase Off
main.py Current
Instructions Empty
Claude...Sonnet
Chat
Share feedback
Waiting for "Scanning" to complete...
Show all (2)
1:1
CRLF
UTF-8
4 spaces
Python 3.10 (langgraph-tutorial)

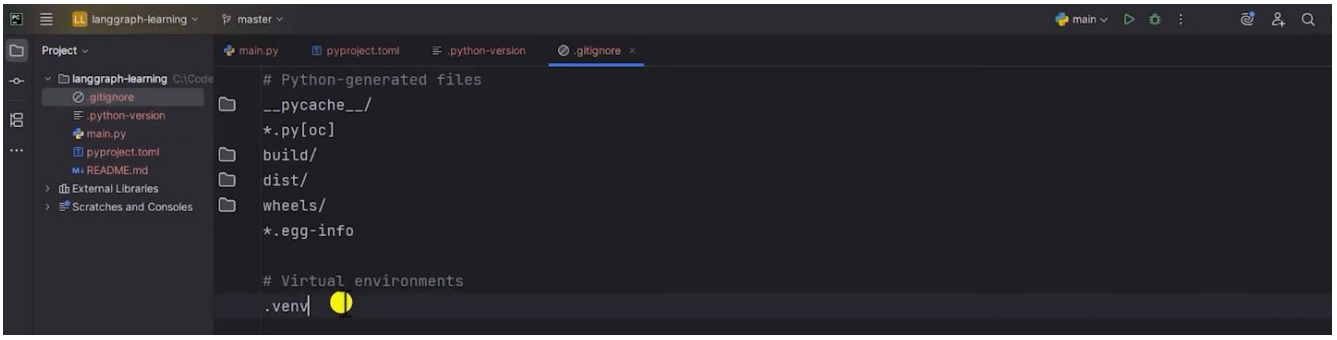

Installation
Search
uv
0.7.13
58.4k
1.7k
Introduction
Getting started
Installation
First steps
Features
Getting help
Guides
Installing Python
Running scripts
Using tools
Working on projects
Publishing packages
Integrations
Concepts
Projects
Tools
Python versions
Configuration files
Package indexes
Resolution
Build backend
Authentication
Caching
The pip interface
Reference
Commands
Settings
Environment variables
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.ps
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
Tip
The installation script may be inspected before use:
macOS and Linux
Windows
PS> powershell -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.
Installation methods
Standalone installer
PyPI
Cargo
Homebrew
WinGet
Scoop
Docker
GitHub Releases
Upgrading uv
Shell autocompletion
Uninstallation
Next steps

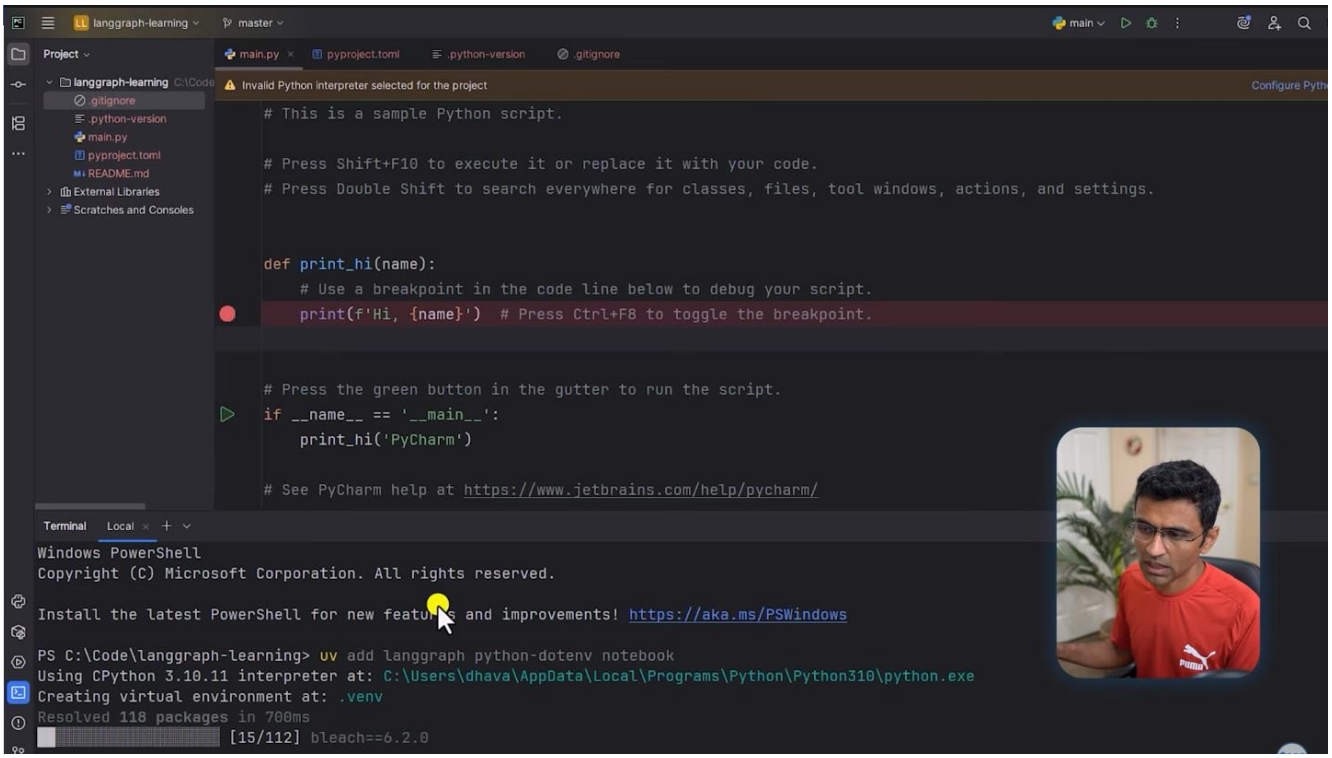


We will use the installed **uv** to install other python packages

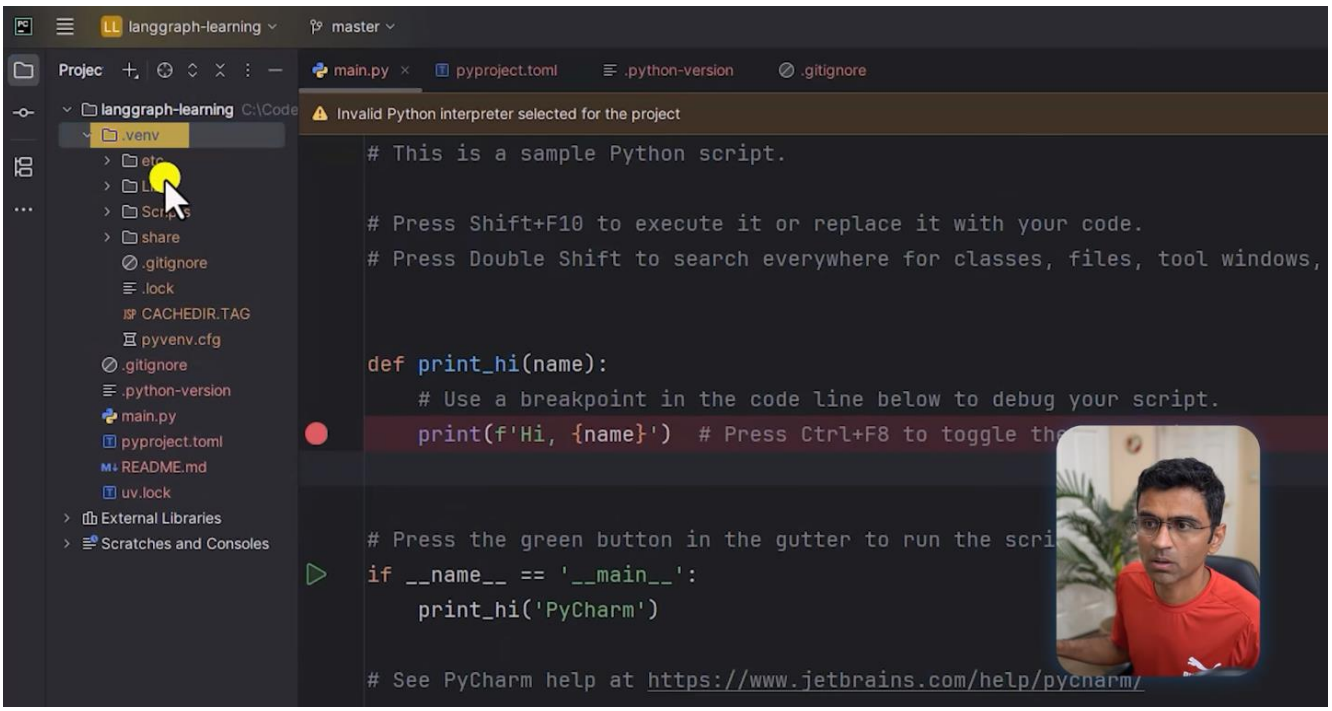


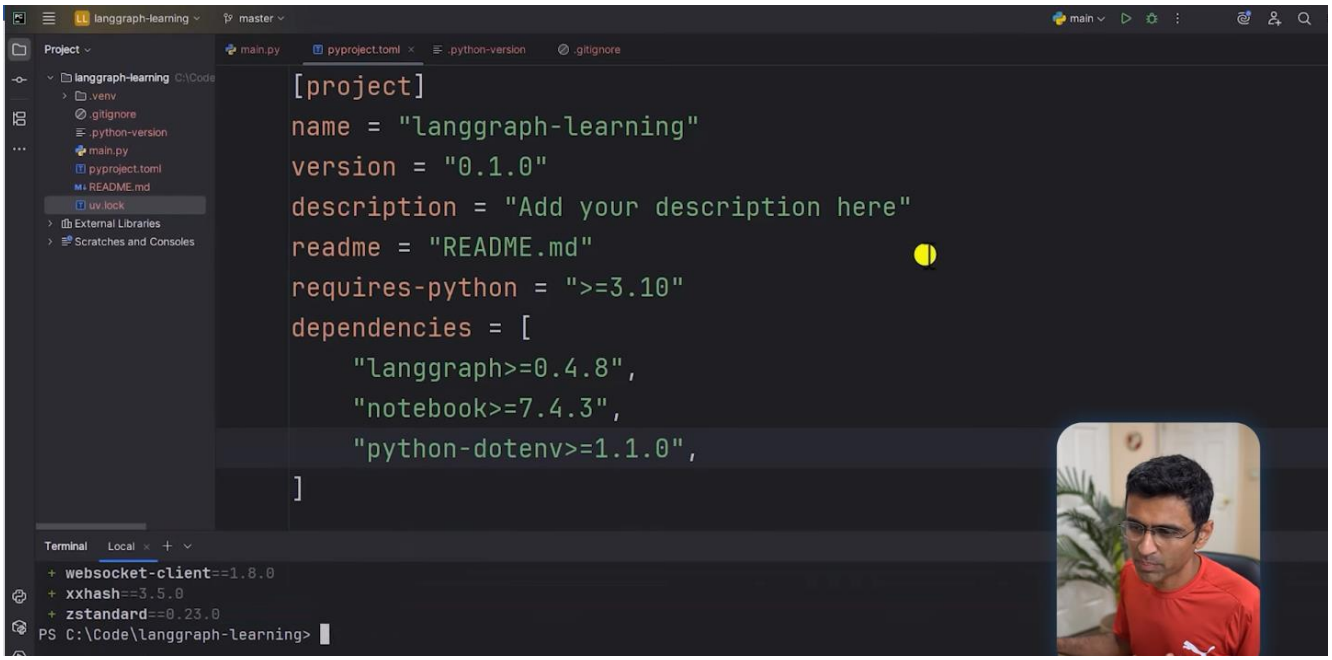


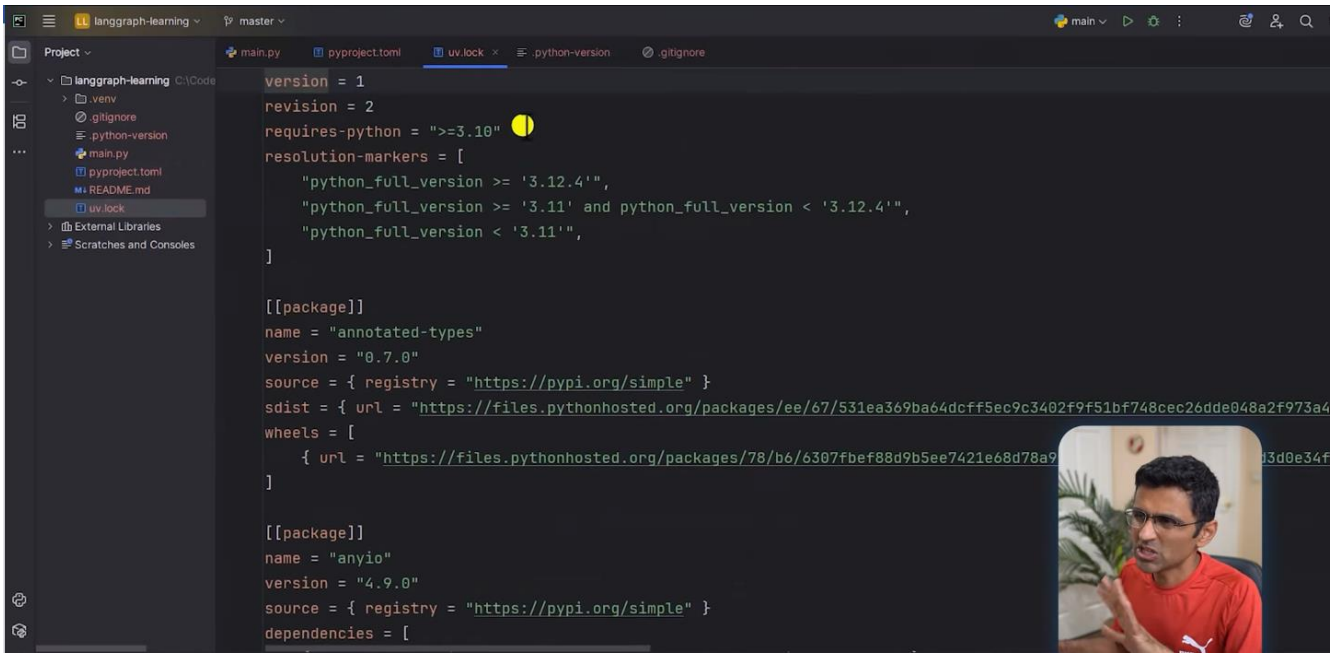


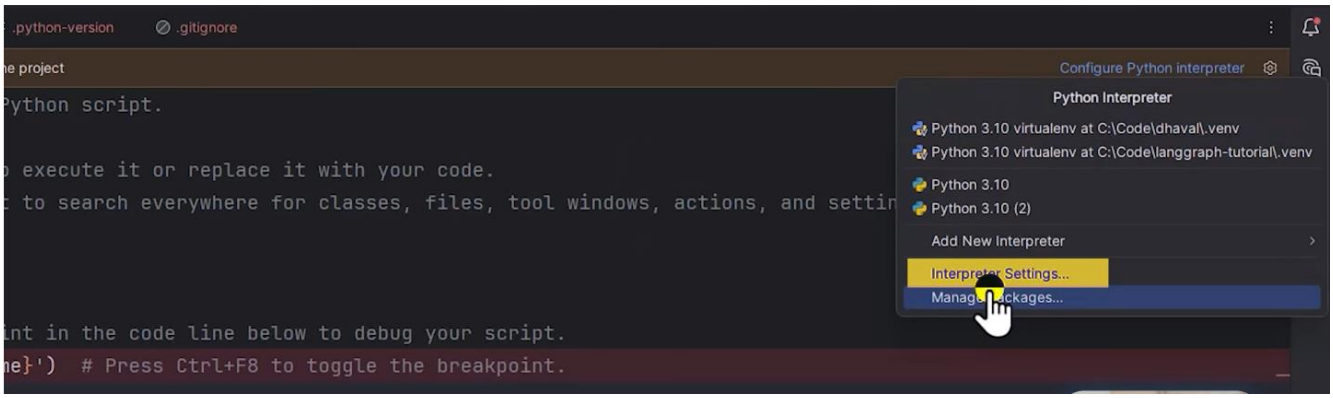


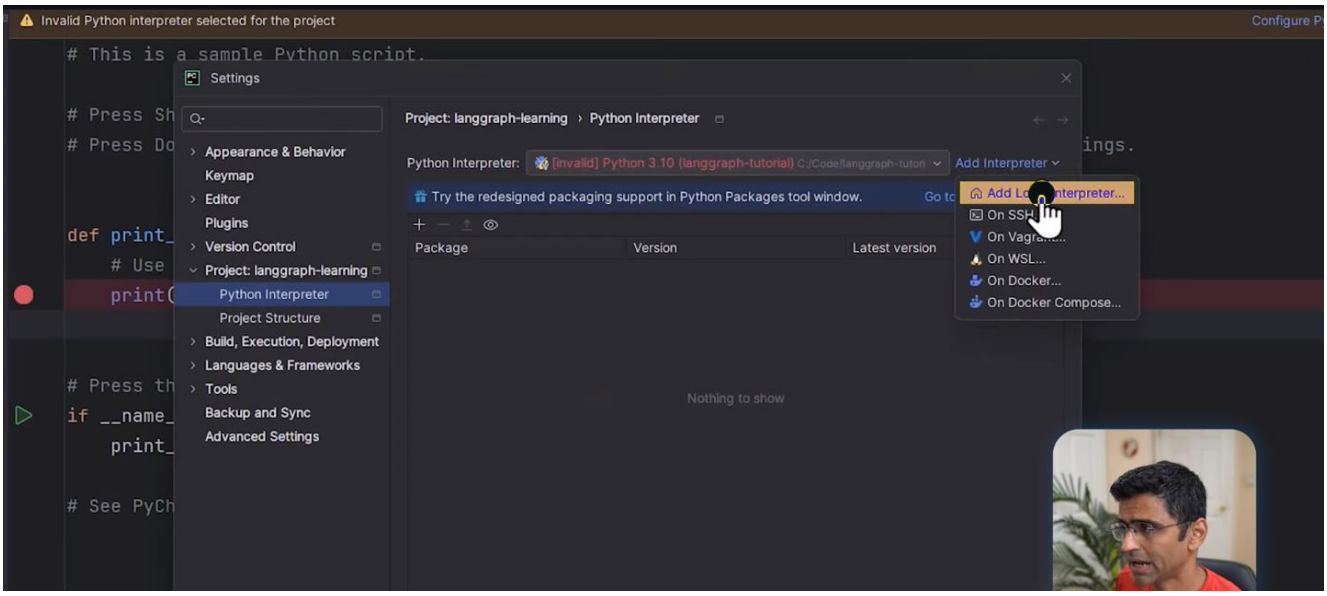
Next, we need to create a virtual environment

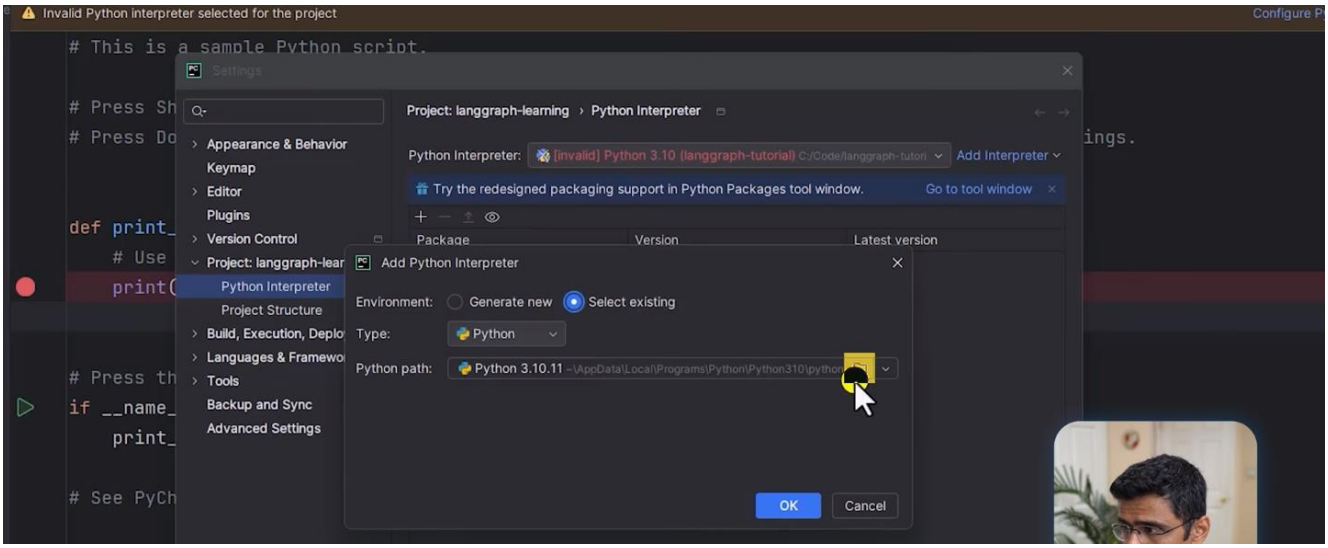


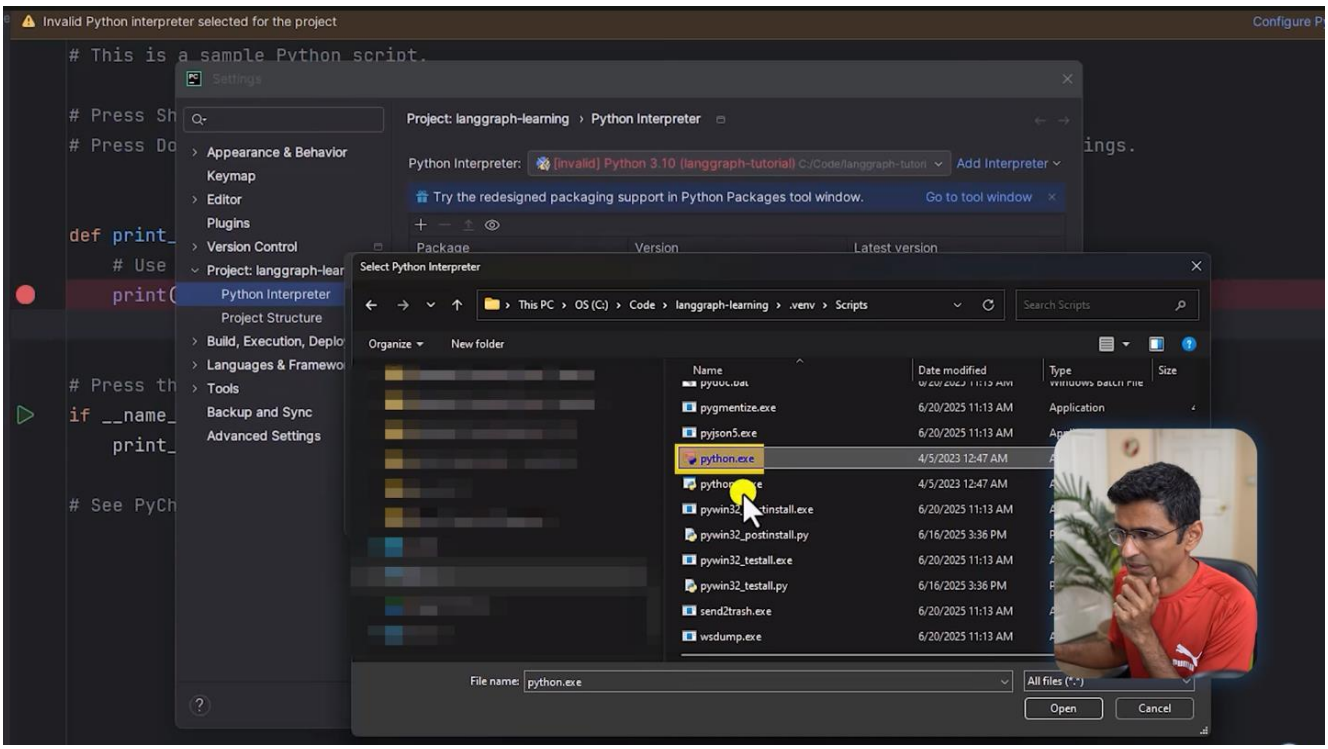

langgraph-learning
master
main
Project
main.py
pyproject.toml
.python-version
.gitignore
langgraph-learning C:\Code
.venv
.gitignore
.python-version
main.py
pyproject.toml
README.md
uv.lock
External Libraries
Scratches and Consoles
[project]
name = "langgraph-learning"
version = "0.1.0"
description = "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",
]
Terminal
Local
+ websocket-client==1.8.0
+ xxhash==3.5.0
+ zstandard==0.23.0
PS C:\Code\langgraph-learning>

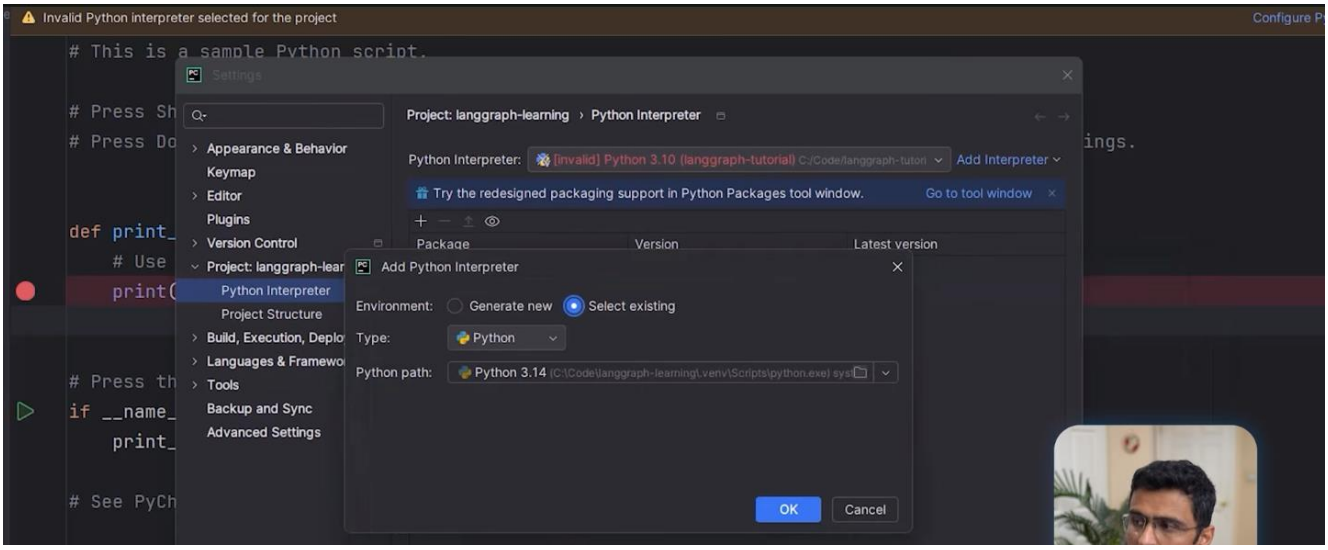

langgraph-learning
master
main
Project
main.py
pyproject.toml
uv.lock
.python-version
.gitignore
langgraph-learning C:\Code
.venv
.gitignore
.python-version
main.py
pyproject.toml
README.md
uv.lock
External Libraries
Scratches and Consoles
version = 1
revision = 2
requires-python = ">=3.10"
resolution-markers = [
"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/67/531ea369ba64dcff5ec9c3402f9f51bf748cec26dde048a2f973a4
wheels = [
{ url = "https://files.pythonhosted.org/packages/78/b6/6307fbef88d9b5ee7421e68d78a9
]
[[package]]
name = "anyio"
version = "4.9.0"
source = { registry = "https://pypi.org/simple" }
dependencies = [

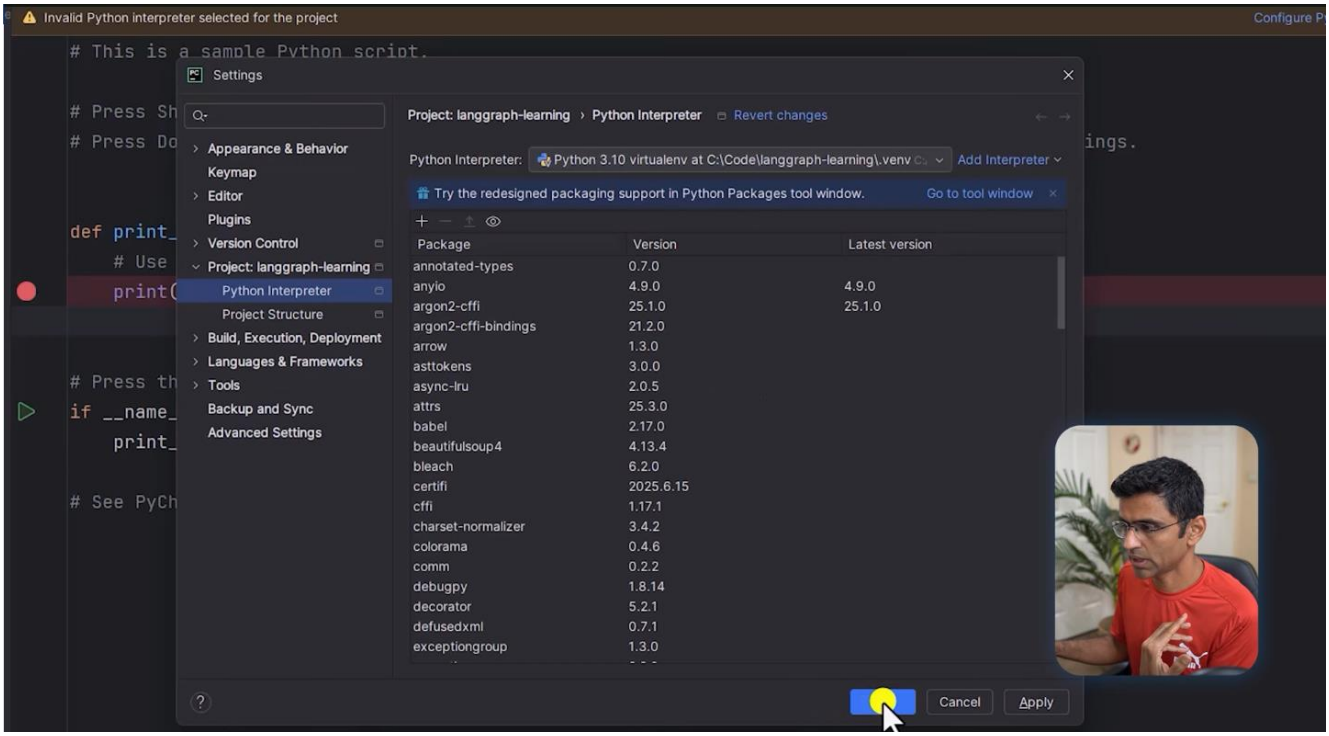

.python-version
.gitignore
e project
Configure Python interpreter
ython script.
execute it or replace it with your code.
to search everywhere for classes, files, tool windows, actions, and settin
int in the code line below to debug your script.
ne}') # Press Ctrl+F8 to toggle the breakpoint.
Python Interpreter
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...
Manage ackages...

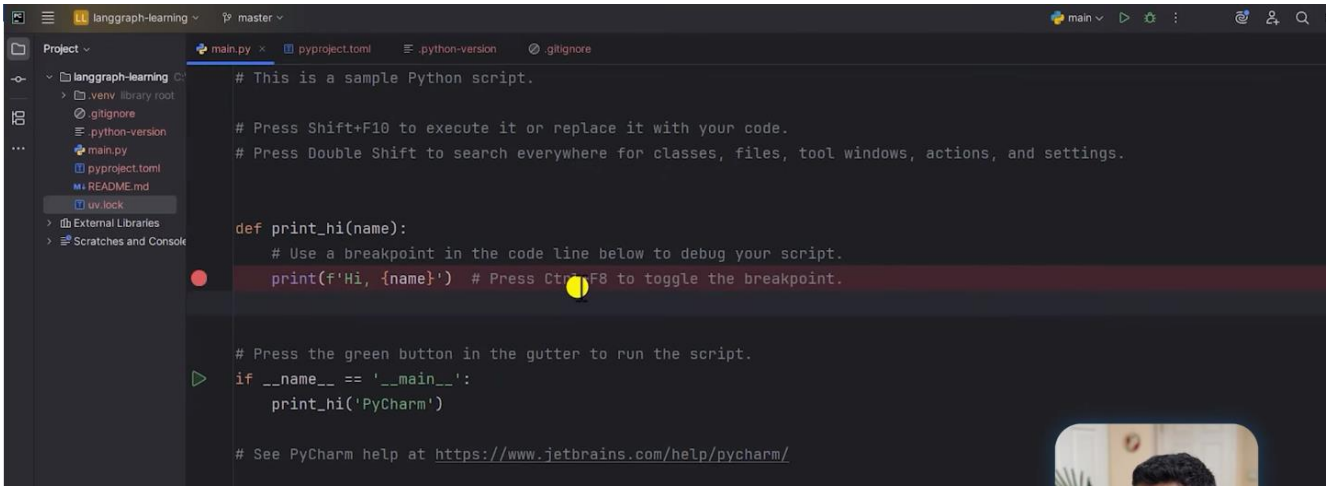

Invalid Python interpreter selected for the project
Settings
Project: langgraph-learning › Python Interpreter
Python Interpreter: [invalid] Python 3.10 (langgraph-tutorial)
Add Interpreter
Try the redesigned packaging support in Python Packages tool window.
Package
Version
Latest version
Nothing to show
On SSH...
On WSL...
On Docker...
On Docker Compose...


Invalid Python interpreter selected for the project
Add Python Interpreter
Environment: Generate new Select existing
Type: Python
Python path: Python 3.10.11
OK
Cancel

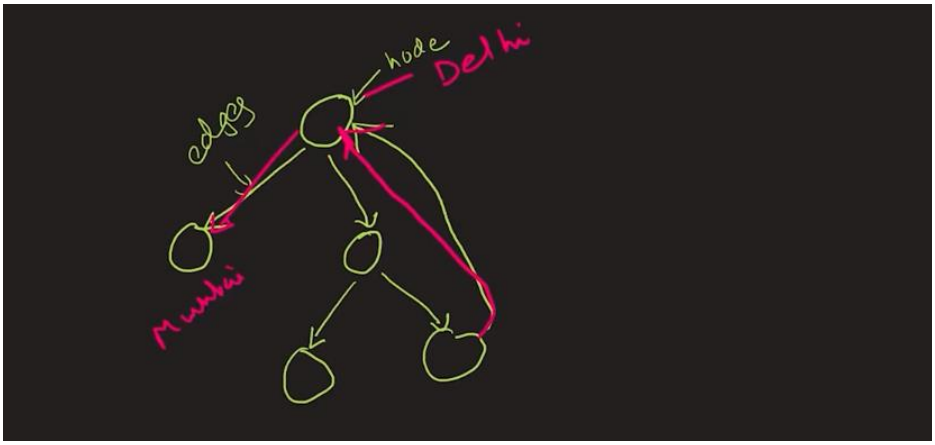

Invalid Python interpreter selected for the project
Select Python Interpreter
Organize
New folder
Name
Date modified
Type
Size
python.exe
File name: python.exe
Open
Cancel

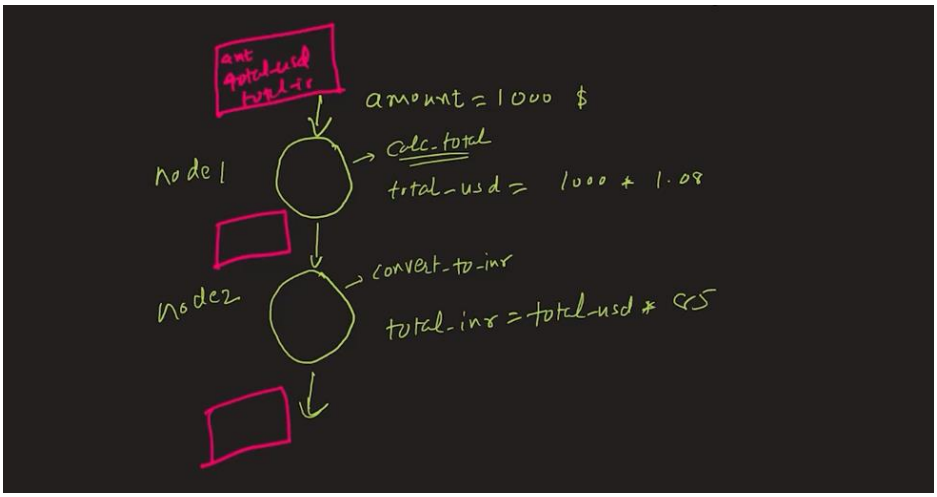


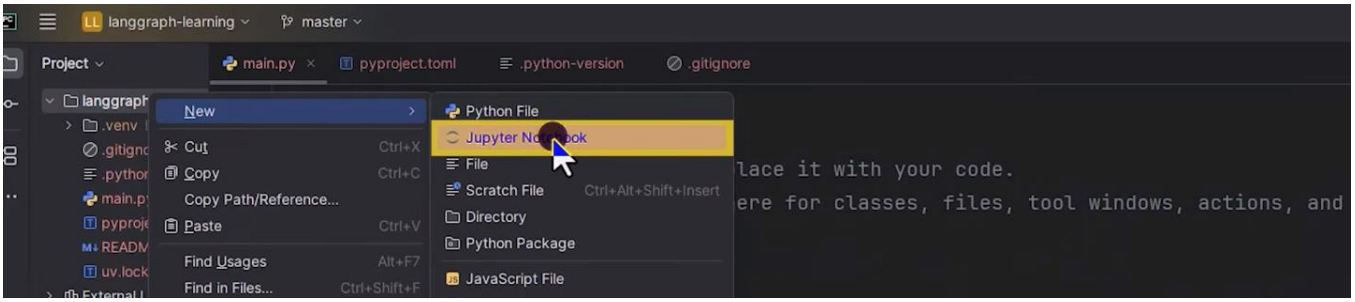


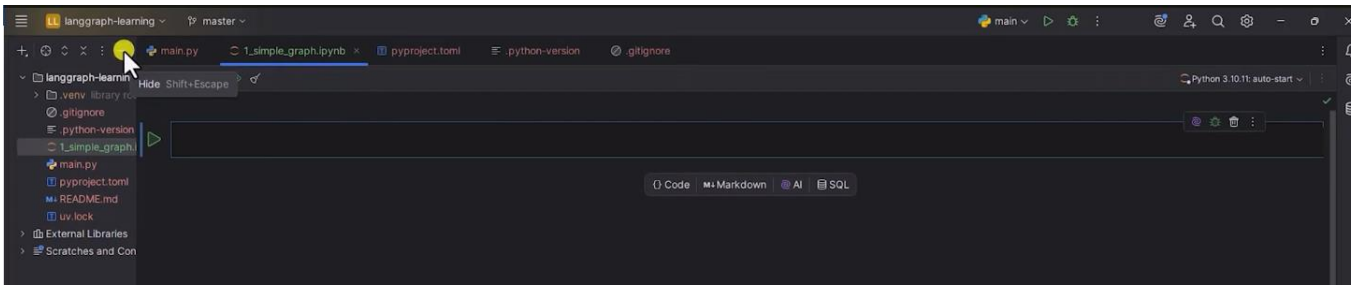


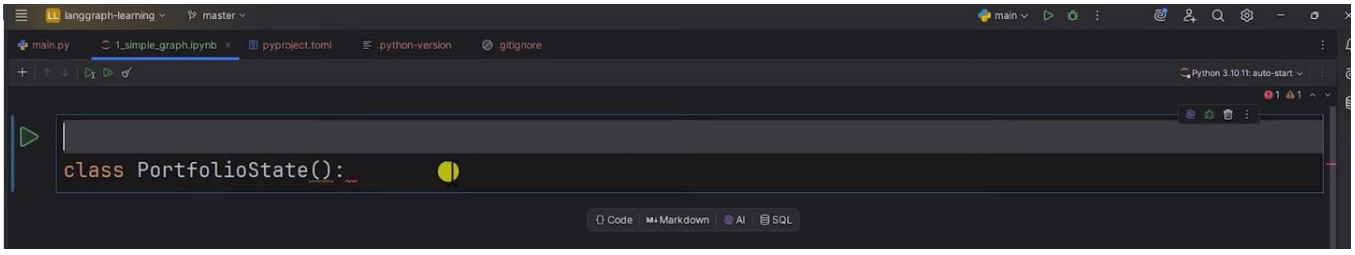
Now your environment is all set.

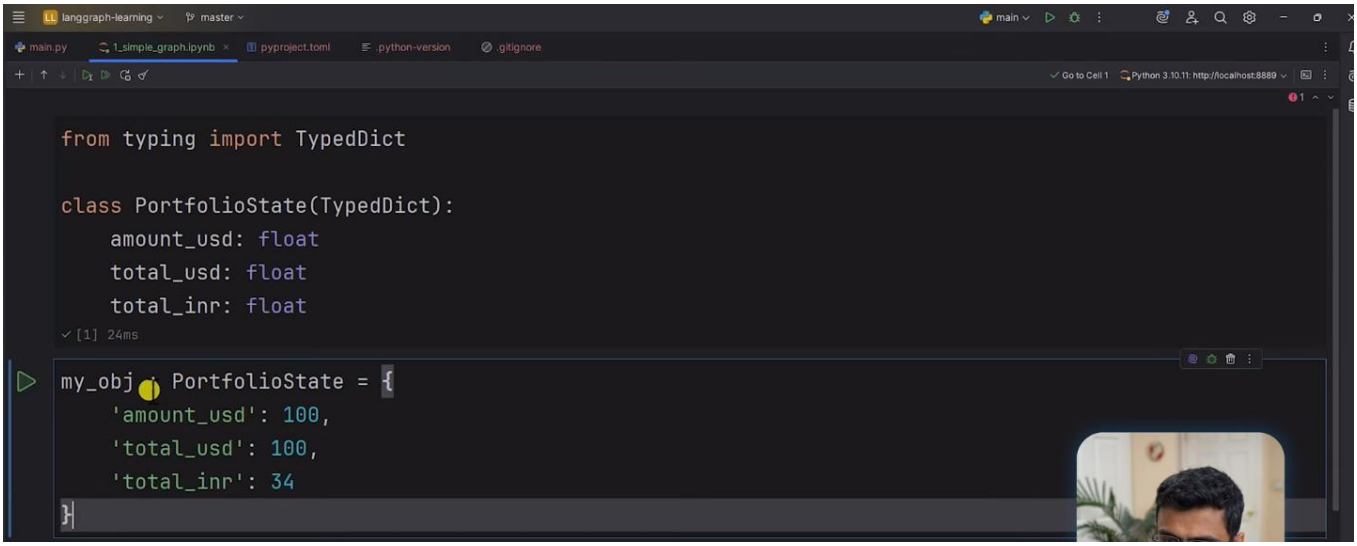

node
Delhi
edges
Mumbai

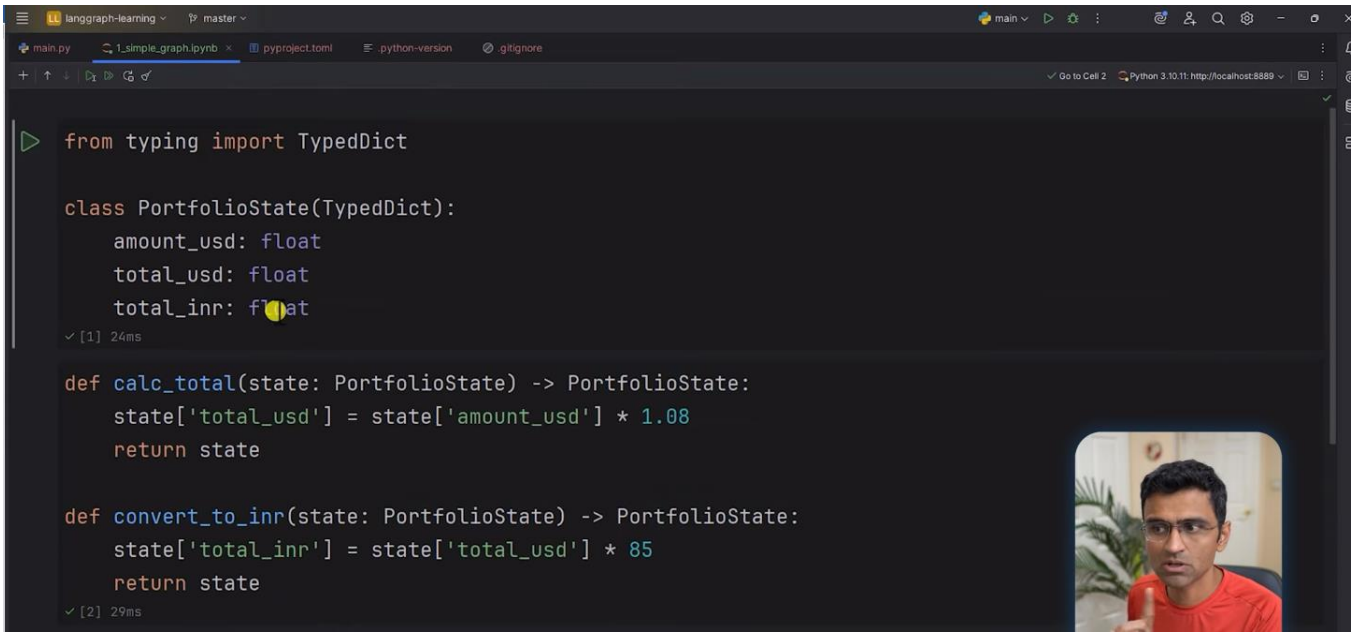

amount = 1000 $
node1
calc_total
total_usd = 1000 * 1.08
node2
convert_to_inr
total_inr = total_usd * 85


langgraph-learning
master
Project
main.py
pyproject.toml
.python-version
.gitignore
New
Cut Ctrl+X
Copy Ctrl+C
Copy Path/Reference...
Paste Ctrl+V
Find Usages Alt+F7
Find in Files... Ctrl+Shift+F
Python File
Jupyter Notebook
File
Scratch File Ctrl+Alt+Shift+Insert
Directory
Python Package
JavaScript File
lace it with your code.
ere for classes, files, tool windows, actions, and

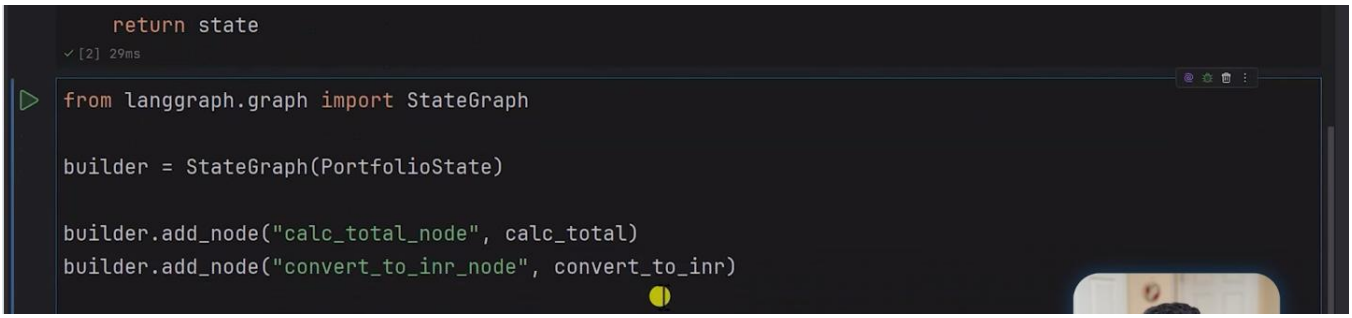

langgraph-learning
master
main
main.py
1_simple_graph.ipynb
pyproject.toml
.python-version
.gitignore
Hide Shift+Escape
Python 3.10.11: auto-start
.venv
.gitignore
.python-version
1_simple_graph.i
main.py
pyproject.toml
README.md
uv.lock
External Libraries
Scratches and Con
Code
Markdown
AI
SQL

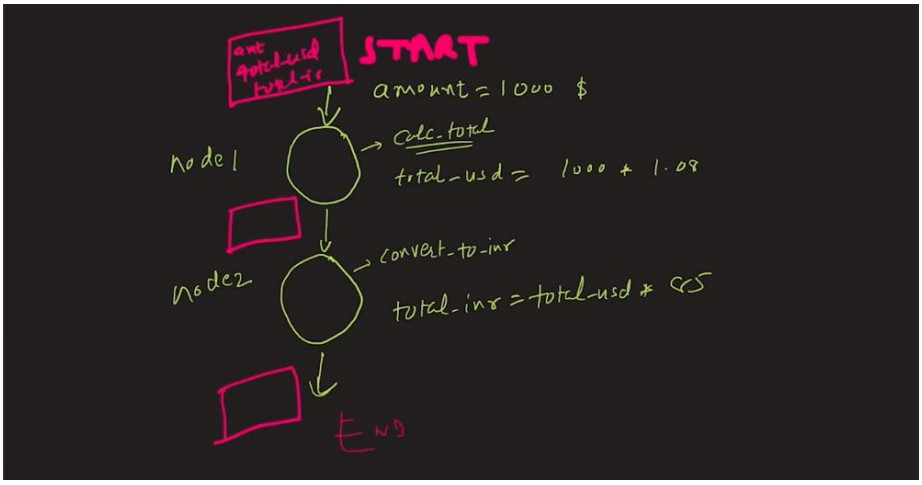

langgraph-learning
master
main
main.py
1_simple_graph.ipynb
pyproject.toml
.python-version
.gitignore
Python 3.10.11: auto-start
class PortfolioState():
Code
Markdown
AI
SQL



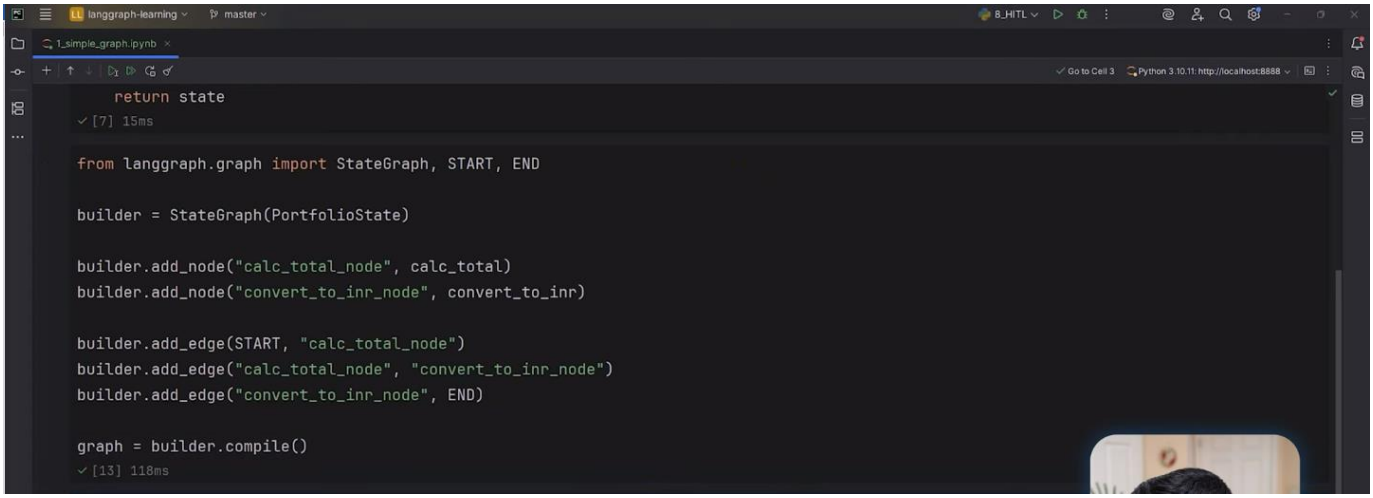


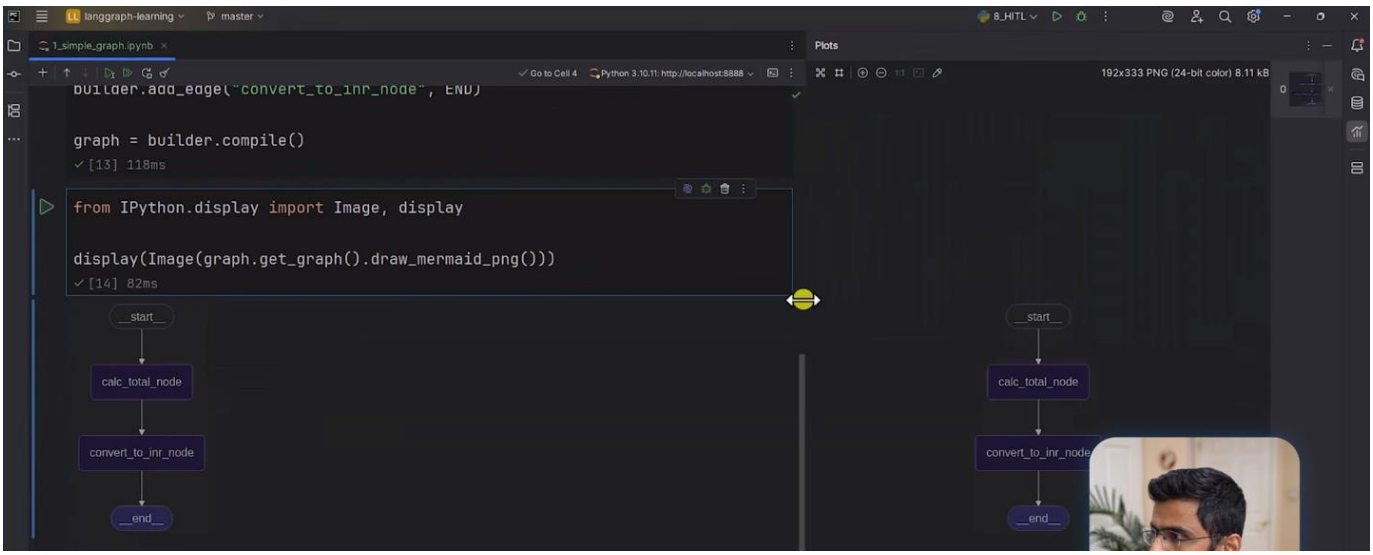
Next, we can define our graph as below



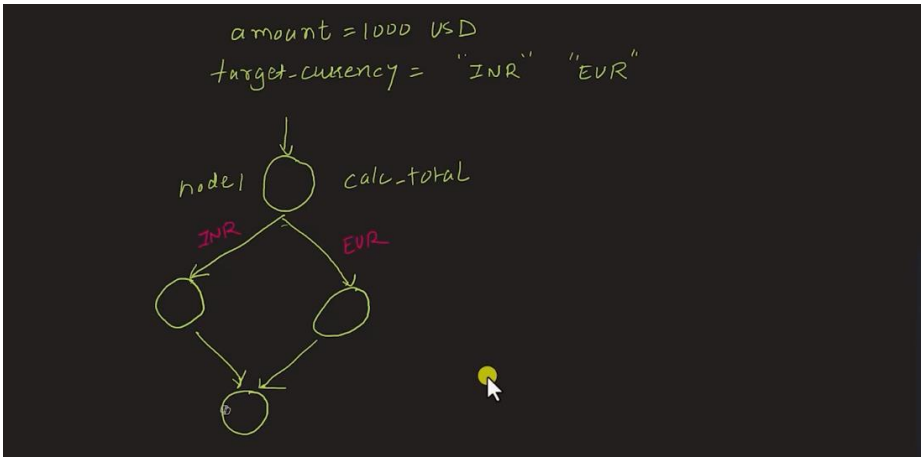


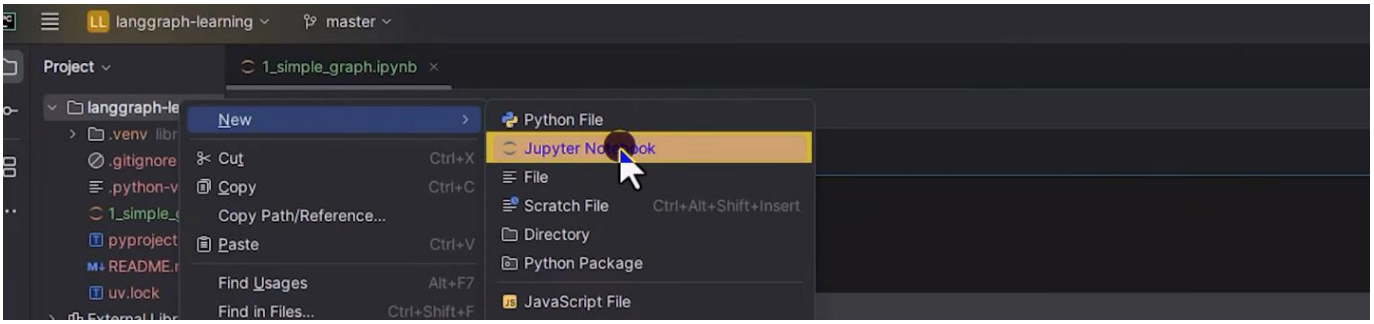
Next, we need to add the edges that define the relationships between the START and END edges.

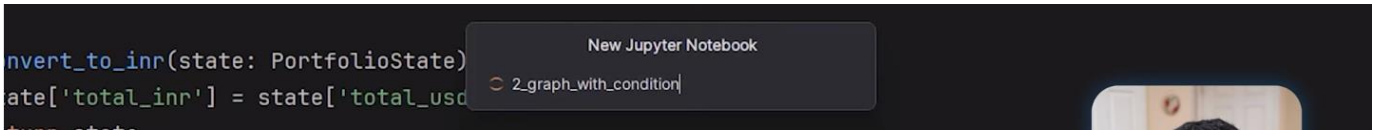


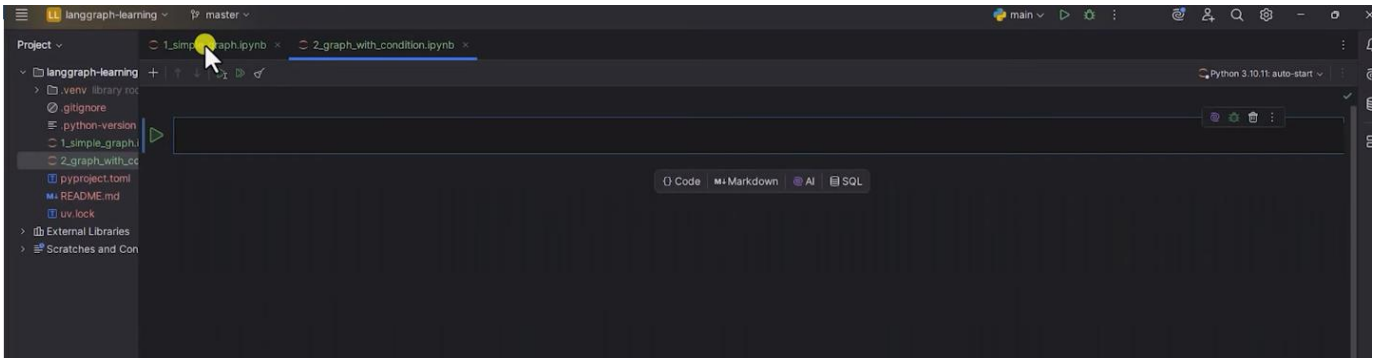



convert_to_inr_node
__end__
graph.invoke({"amount_usd": 1000})
graph
[15] 40ms
{'amount_usd': 1000, 'total_usd': 1080.0, 'total_inr': 91800.0}
Code
Markdown
AI
SQL


amount = 1000 USD
target_currency = "INR" "EUR"
node1
calc_total
INR
EUR


langgraph-learning
master
Project
1_simple_graph.ipynb
New
Cut
Copy
Copy Path/Reference...
Paste
Find Usages
Find in Files...
Python File
Jupyter Notebook
File
Scratch File
Directory
Python Package
JavaScript File
pyproject
README.
uv.lock


nvert_to_inr(state: PortfolioState)
ate['total_inr'] = state['total_usd
New Jupyter Notebook
2_graph_with_condition


langgraph-learning
master
main
Project
2_graph_with_condition.ipynb
Python 3.10.11: auto-start
.venv
.gitignore
.python-version
pyproject.toml
README.md
uv.lock
External Libraries
Code
Markdown
AI
SQL

```python
from typing import TypedDict

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

```python
from typing import TypedDict, Literal

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

[1] 15ms

```python
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
```

```python
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"]
```

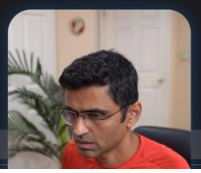
[2] 11ms

```python
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)
```

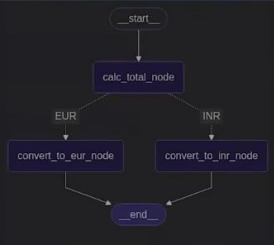
Next, we add the edges

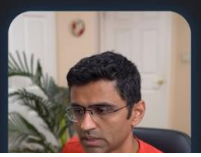
```
builder.add_node("convert_to_inr_node", convert_to_inr)
builder.add_node("convert_to_eur_node", convert_to_eur)

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)
```

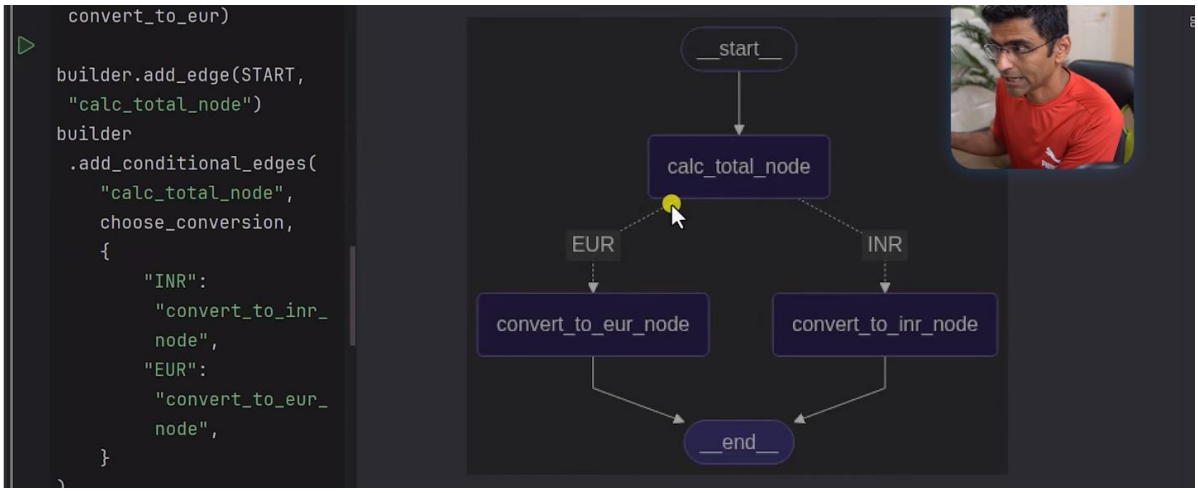
```
builder.add_node("convert_to_inr_node", convert_to_inr)
builder.add_node("convert_to_eur_node", convert_to_eur)

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()
```

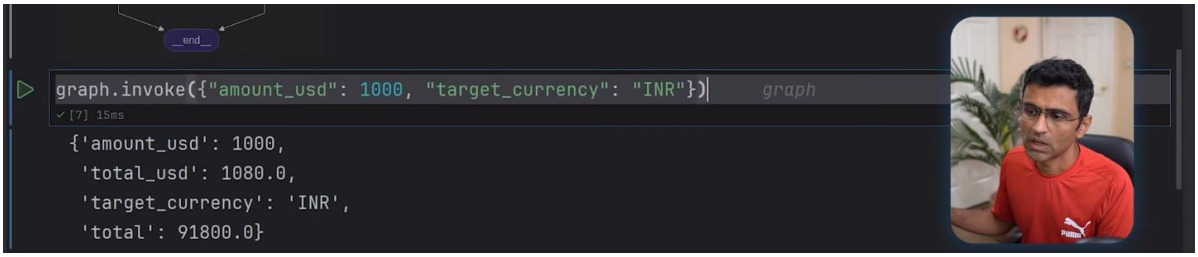


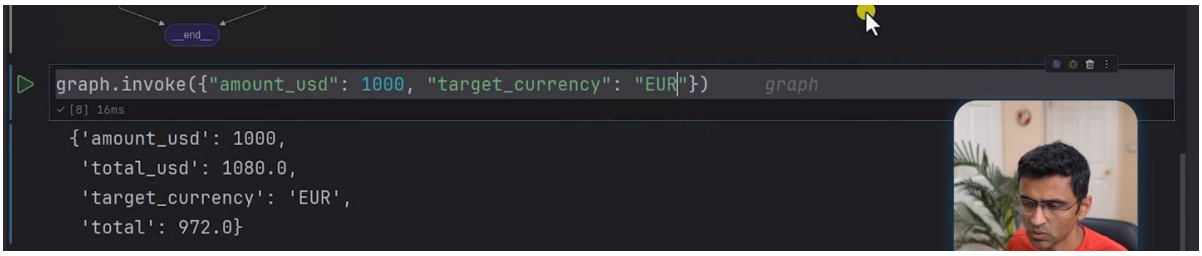
```
graph = builder.compile()
[5] 16ms

from IPython.display import Image, display

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



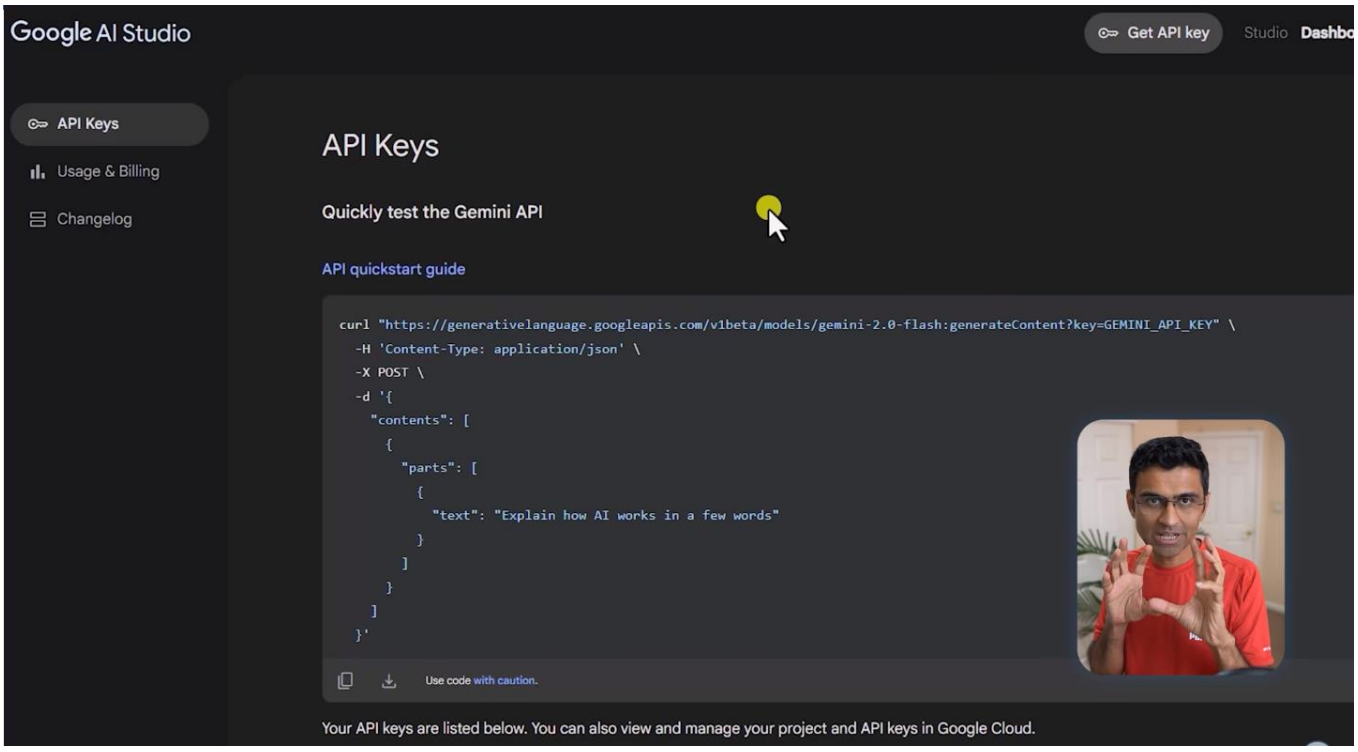




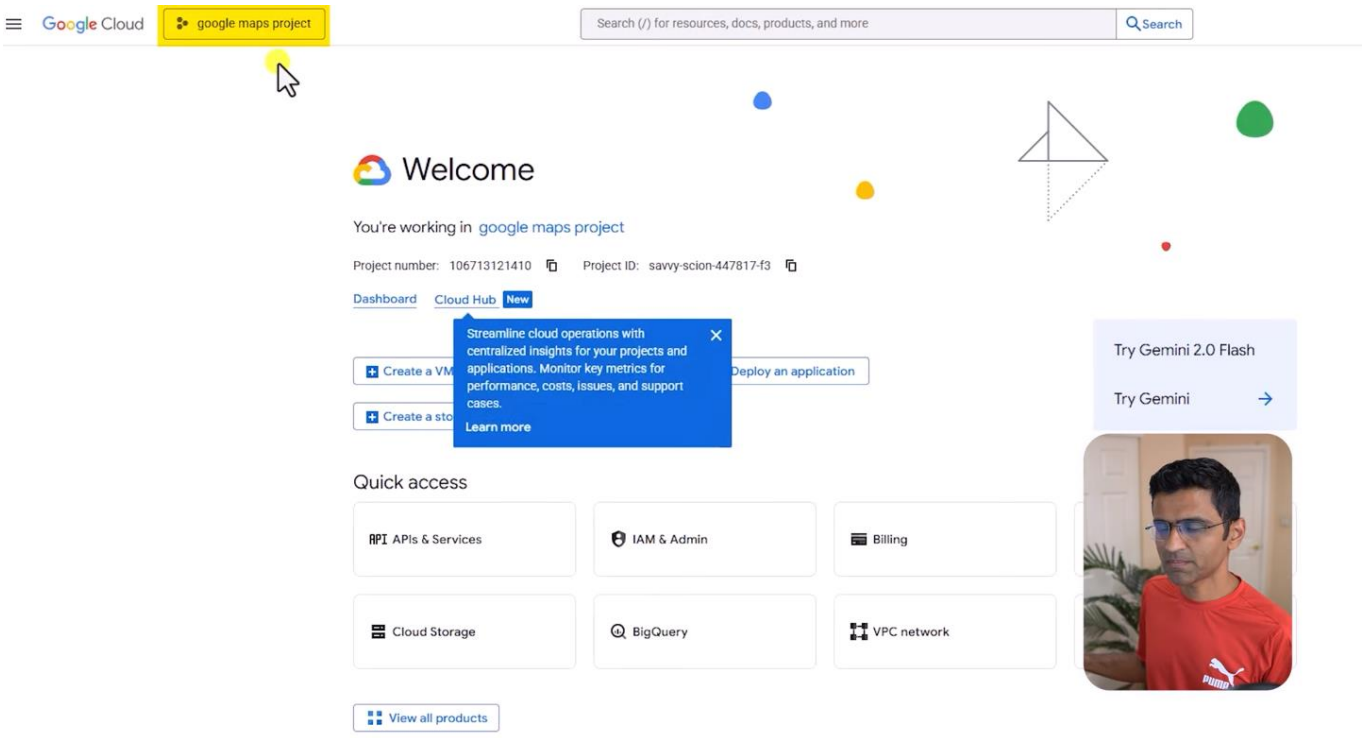


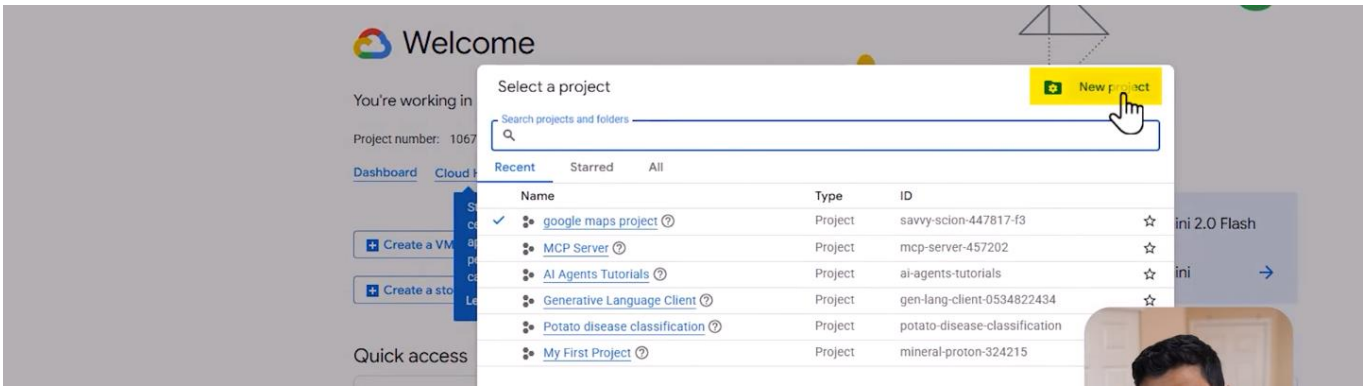


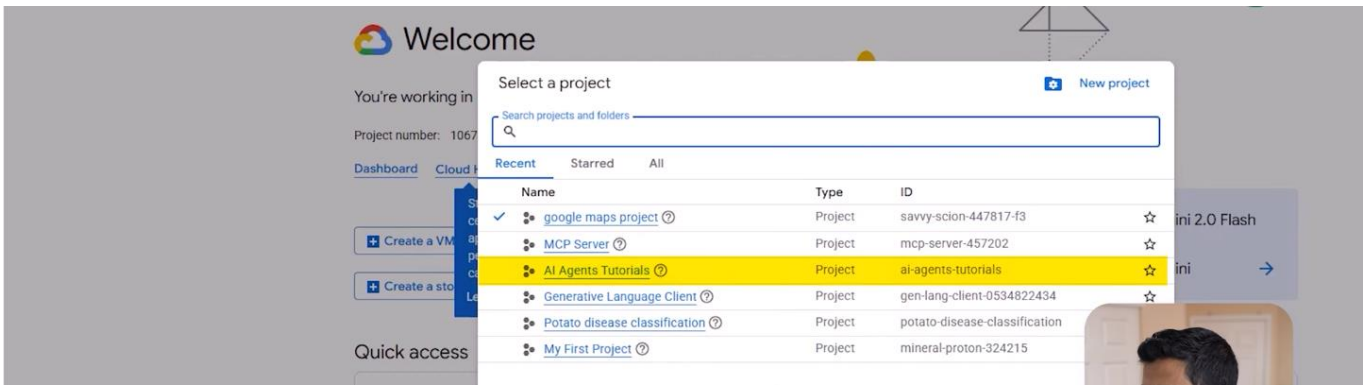
It is now correctly going through the correct conditional path based on the selected output currency.

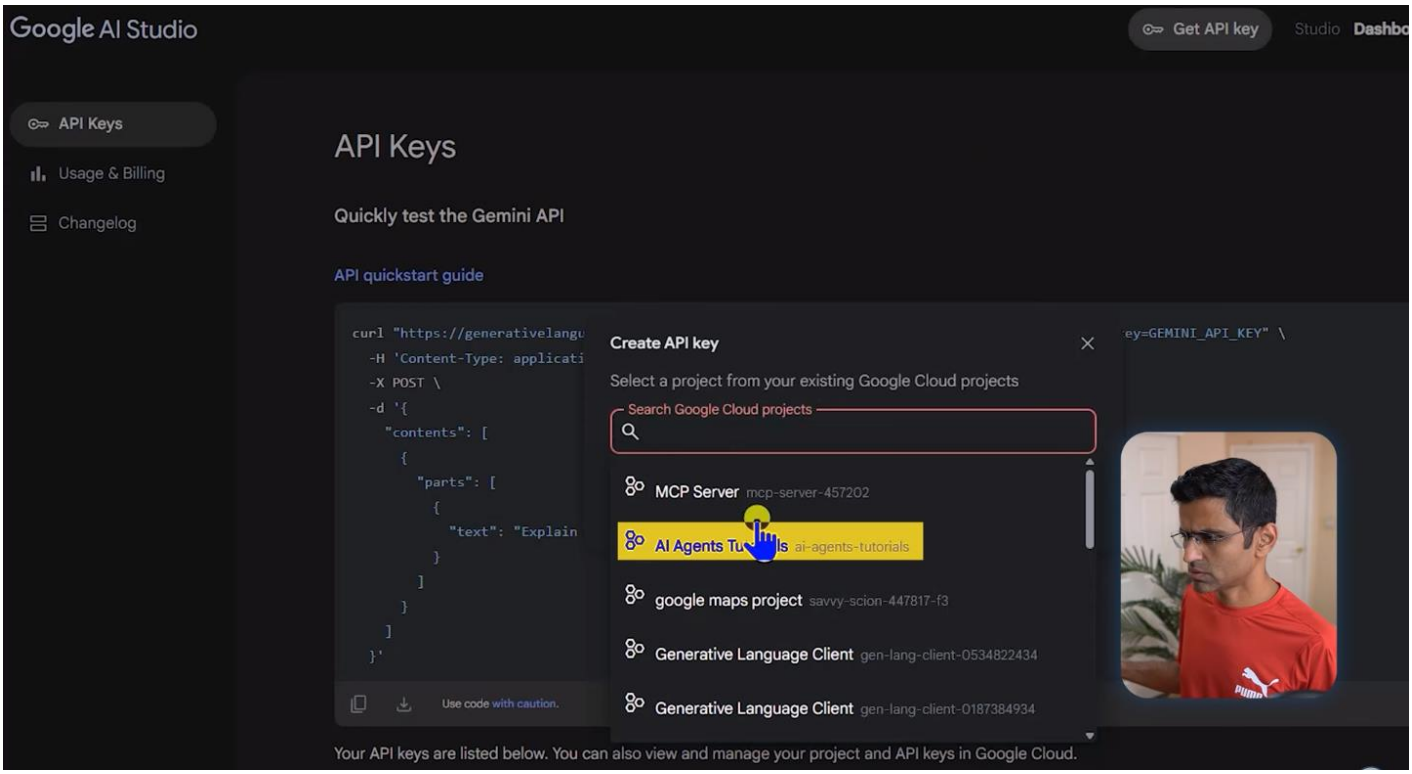


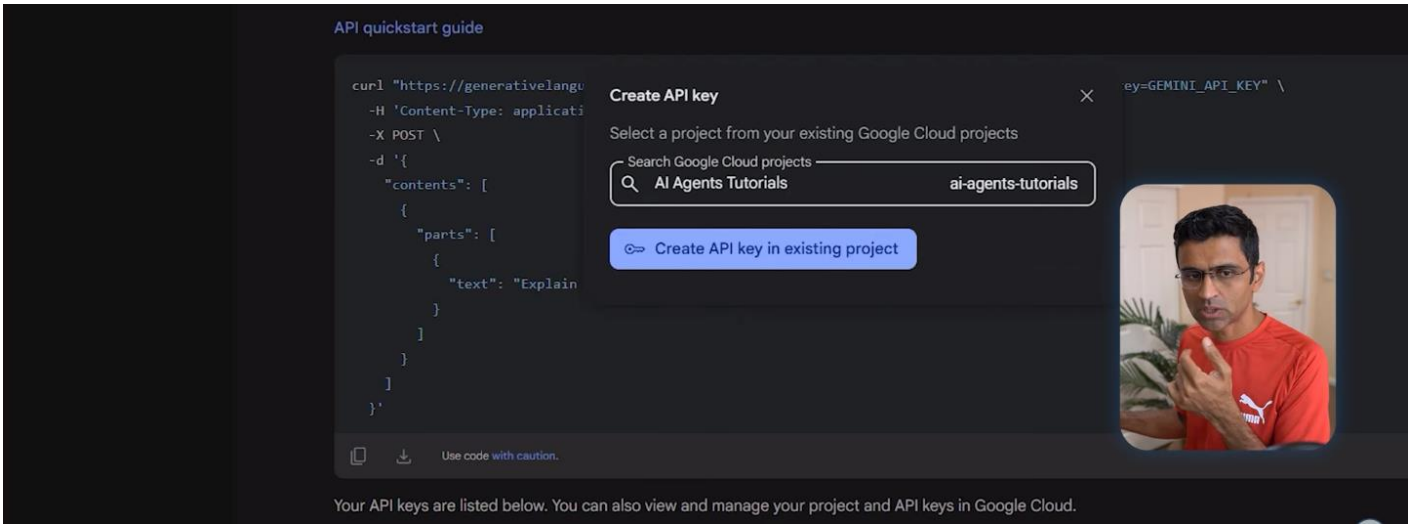
Next, let us create a chatbot using Gemini

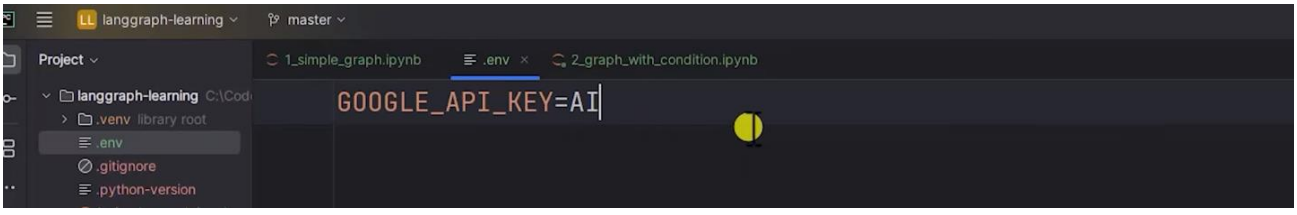

Google Cloud
google maps project
Search (/) for resources, docs, products, and more
Search
Welcome
You're working in google maps project
Project number: 106713121410
Project ID: savvy-scion-447817-f3
Dashboard
Cloud Hub
New
Streamline cloud operations with centralized insights for your projects and applications. Monitor key metrics for performance, costs, issues, and support cases.
Learn more
Create a VM
Deploy an application
Create a sto
Try Gemini 2.0 Flash
Try Gemini
Quick access
APIs & Services
IAM & Admin
Billing
Cloud Storage
BigQuery
VPC network
View all products


Welcome
You're working in
Project number: 1067
Dashboard
Select a project
New project
Search projects and folders
Recent
Starred
All
Name
Type
ID
google maps project
Project
savvy-scion-447817-f3
MCP Server
mcp-server-457202
AI Agents Tutorials
ai-agents-tutorials
Generative Language Client
gen-lang-client-0534822434
Potato disease classification
potato-disease-classification
My First Project
mineral-proton-324215
Create a VM
Create a sto
Quick access

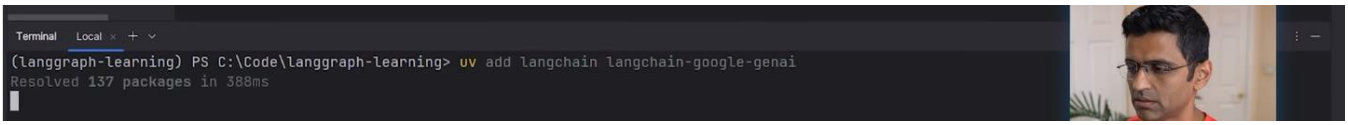

Welcome
You're working in
Project number: 1067
Dashboard
Select a project
New project
Search projects and folders
Recent
Starred
All
Name
Type
ID
google maps project
Project
savvy-scion-447817-f3
MCP Server
mcp-server-457202
AI Agents Tutorials
ai-agents-tutorials
Generative Language Client
gen-lang-client-0534822434
Potato disease classification
potato-disease-classification
My First Project
mineral-proton-324215
Create a VM
Create a sto
Quick access

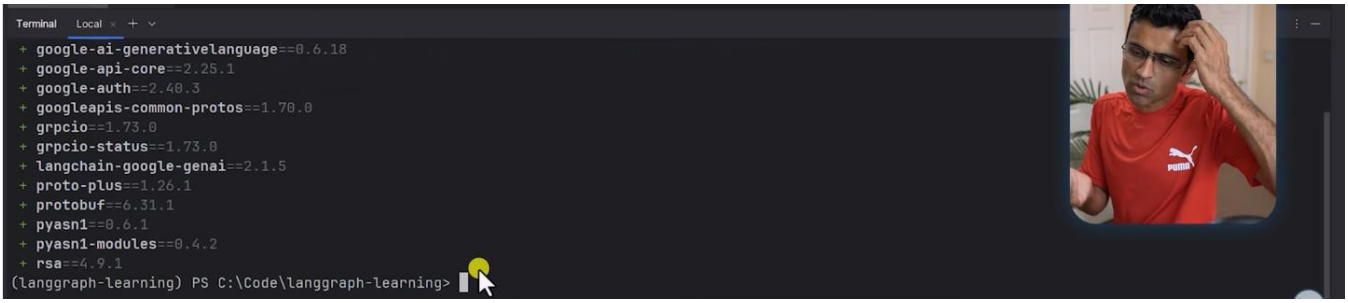


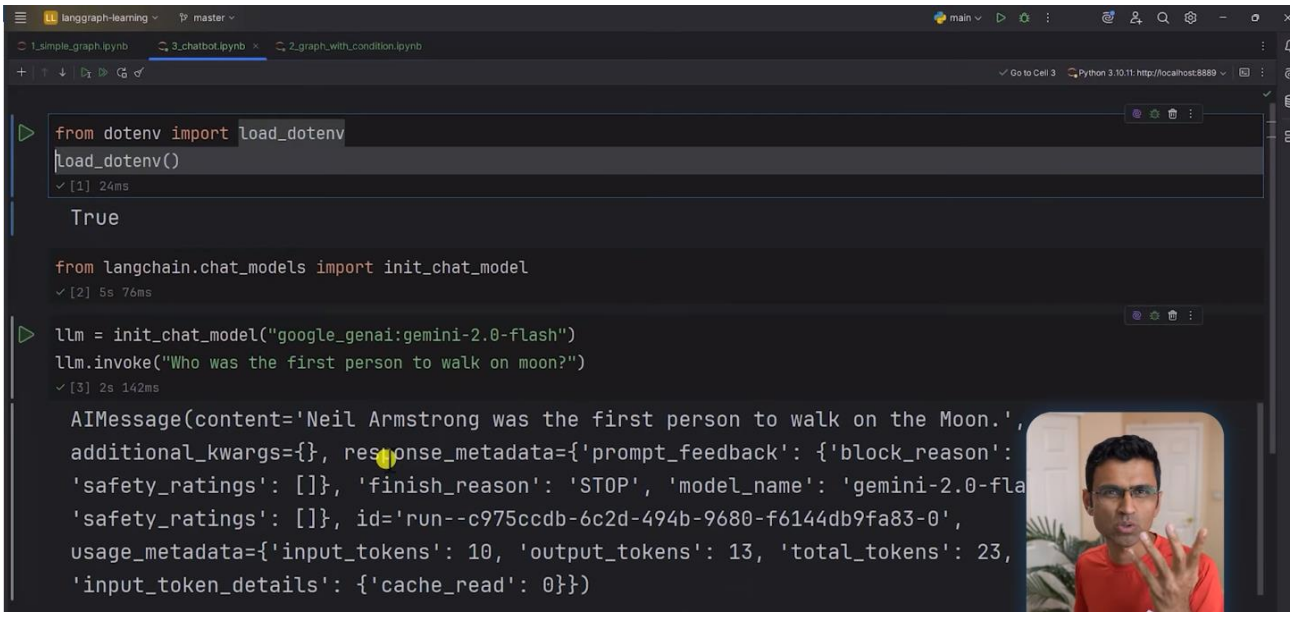




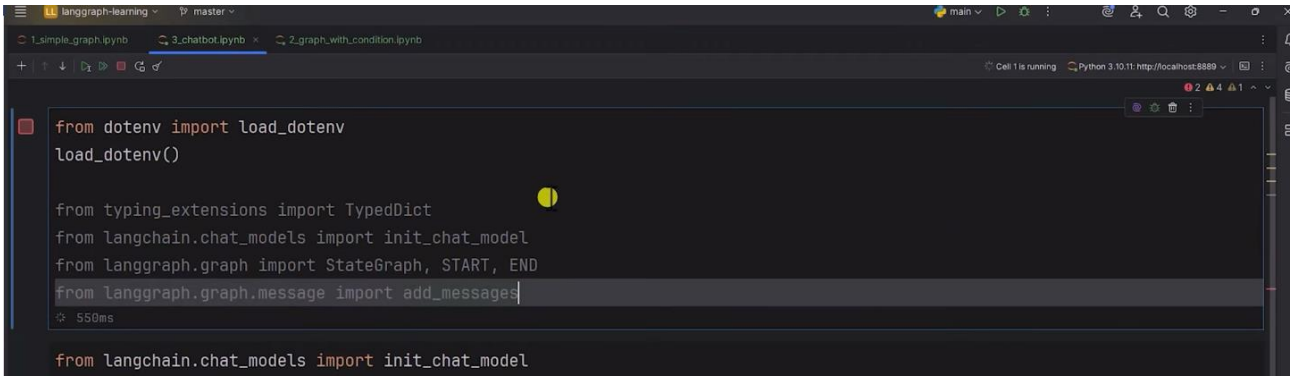
Create a new project file as above







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



```
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"}
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-8cfe7d28f82c-0', usage_metadata={'input_tokens': 14, 'output_tokens': 3, 'total_tokens':
17, 'input_token_details': {'cache_read': 0}})]
```

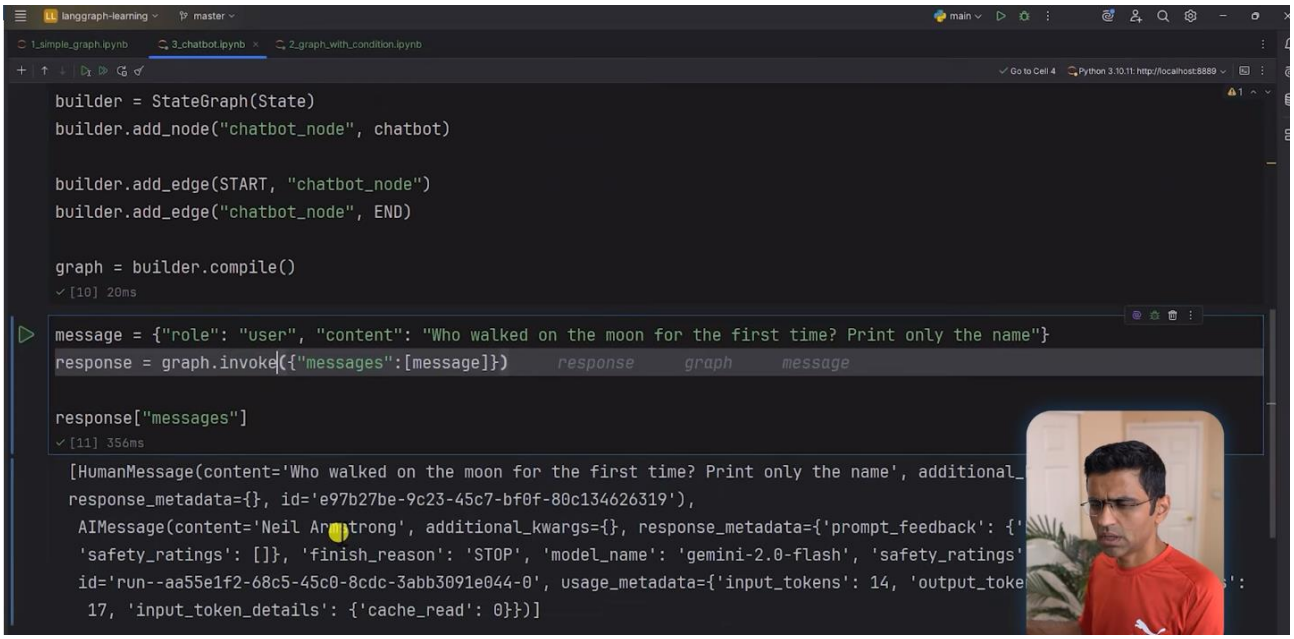
Next, we want to preserve the messages being added over the conversation

langgraph-learning · master · main

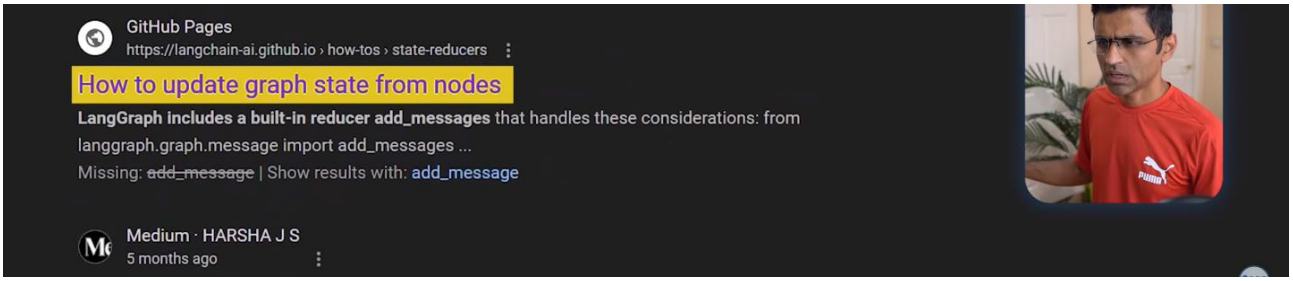
1_simple_graph.ipynb | 3_chatbot.ipynb | 2_graph_with_condition.ipynb

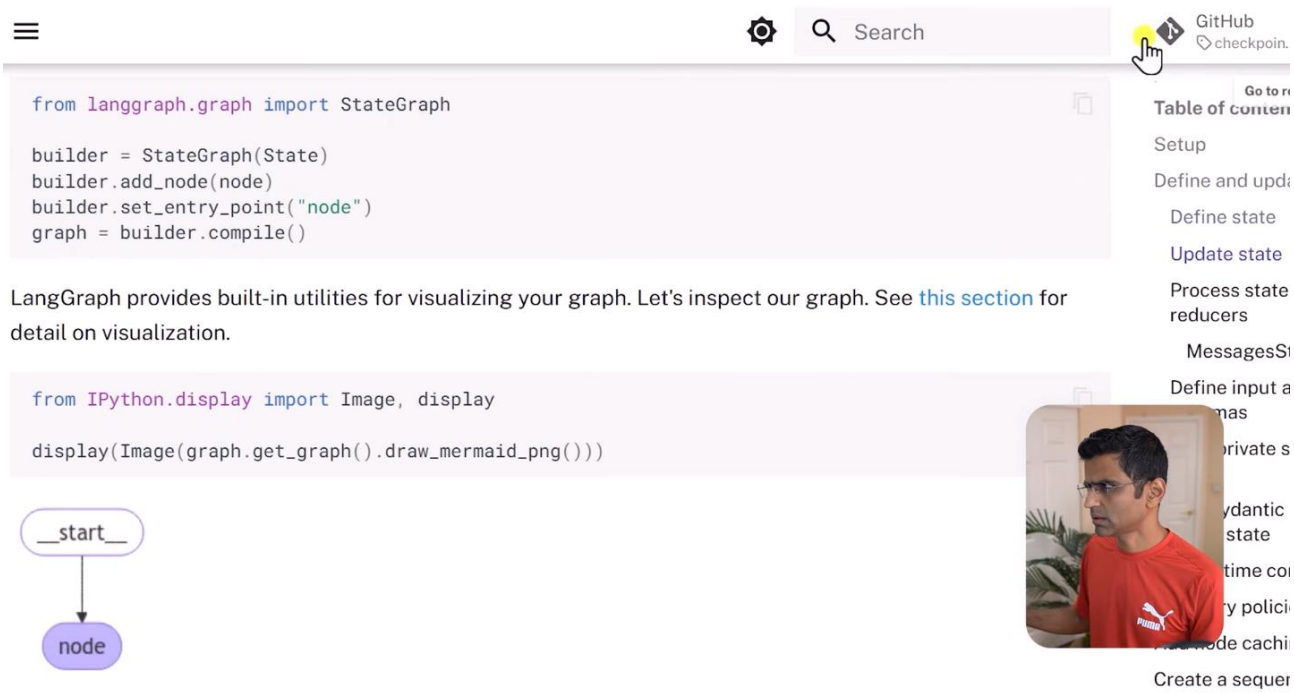
Go to Cell 3 · Python 3.10.11: http://localhost:8889

```
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)
```



We can now see both HumanMessage and AIMessage in the **response[“messages”]** list





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 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":`

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 w reducer function (see `messages` key in the example below). The reducer function is vital to telling the g how to update the list of `Message` objects in the state with each state update (for example, when a noc an update). If you don't specify a reducer, every state update will overwrite the list of messages with th recently provided value. If you wanted to simply append messages to the existing list, you could use `operator.add` as a reducer.

Back to top

MessagesState

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 argumen state, and (optionally), the **second** positional argument is a "config", containing optional configurable parameters (such as a `thread_id`).

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:

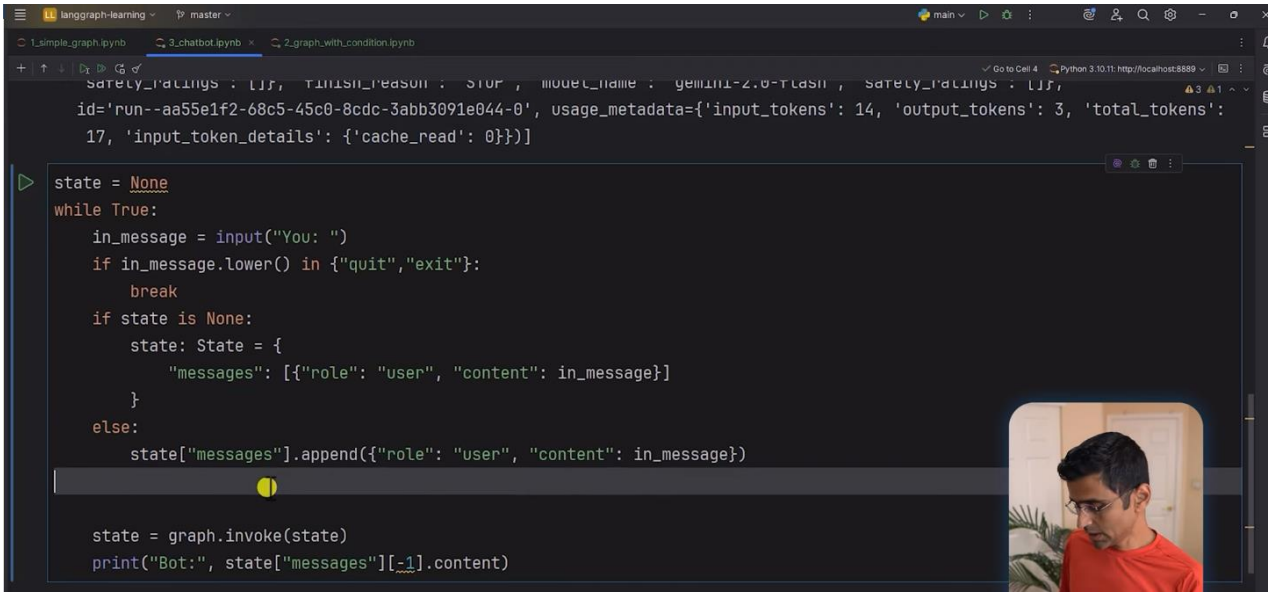
```
# this is supported
{"messages": [HumanMessage(content="message")]}

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

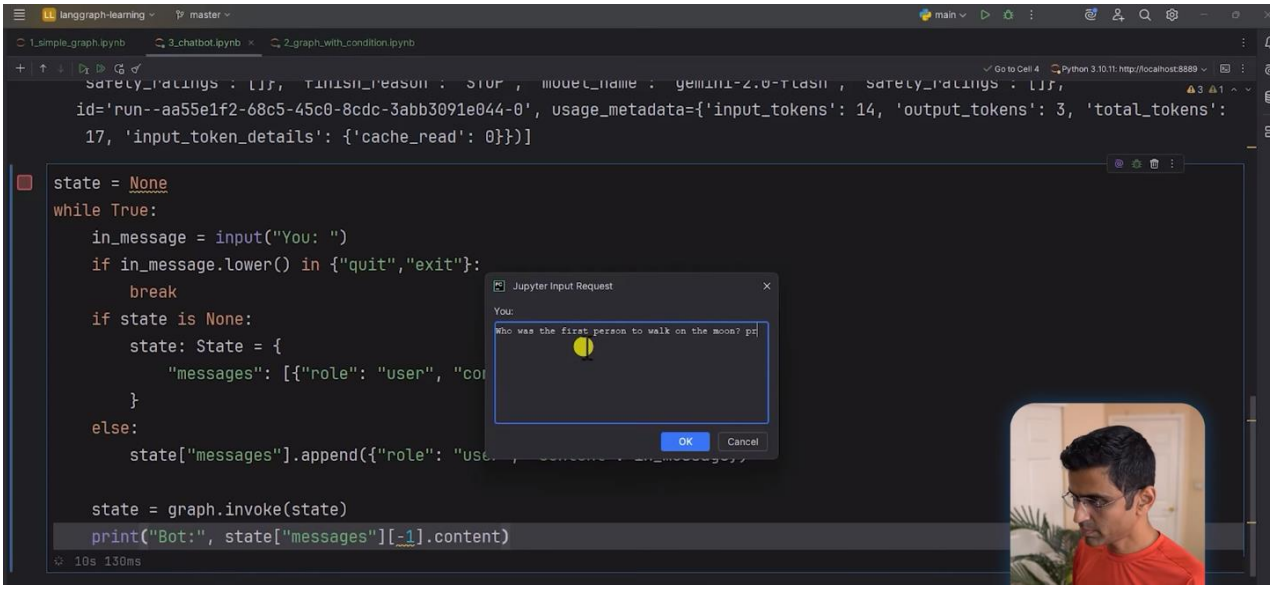
Since the state updates are always deserialized into LangChain `Messages` when using `add_messages`, you should use dot notation to access message attributes, like `state["messages"][-1].content`. Below is a example of a graph that uses `add_messages` as it's reducer function.

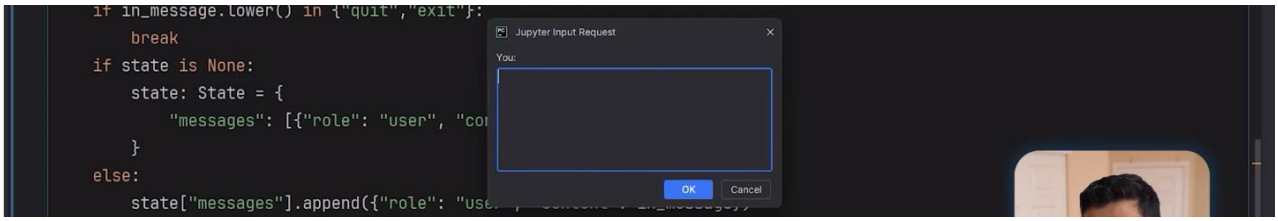
API Reference: AnyMessage | add_messages

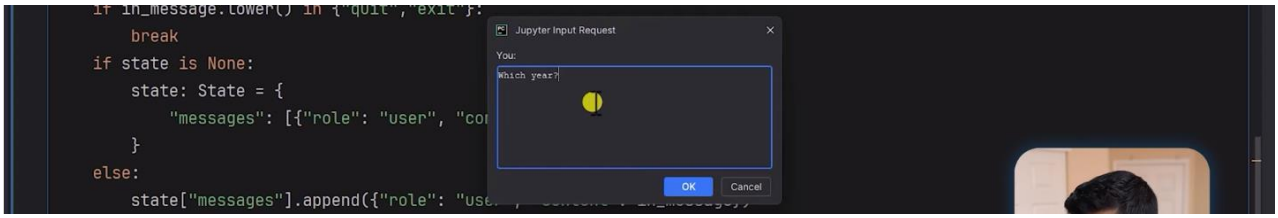
```
from langchain_core.messages import AnyMessage
from langgraph.graph.message import add_messages
from typing import Annotated
from typing_extensions import TypedDict

class GraphState(TypedDict):
```

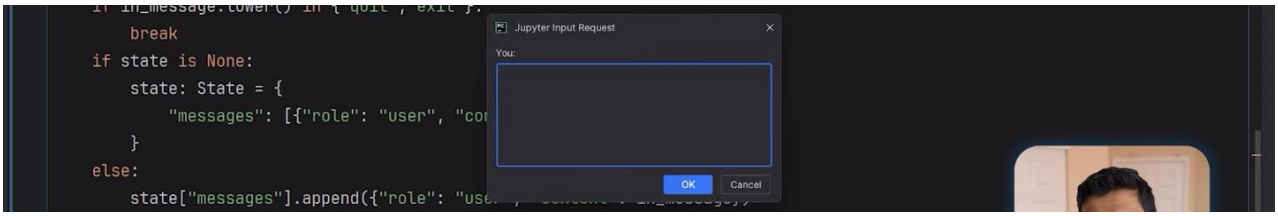
Table of conten
Graphs
StateGraph
Compiling you
State
Schema
Multiple sch
Reducers
Default Red
ng with
State
use me
g Messa
ph
alizatior
sagesSt
Nodes
START Node



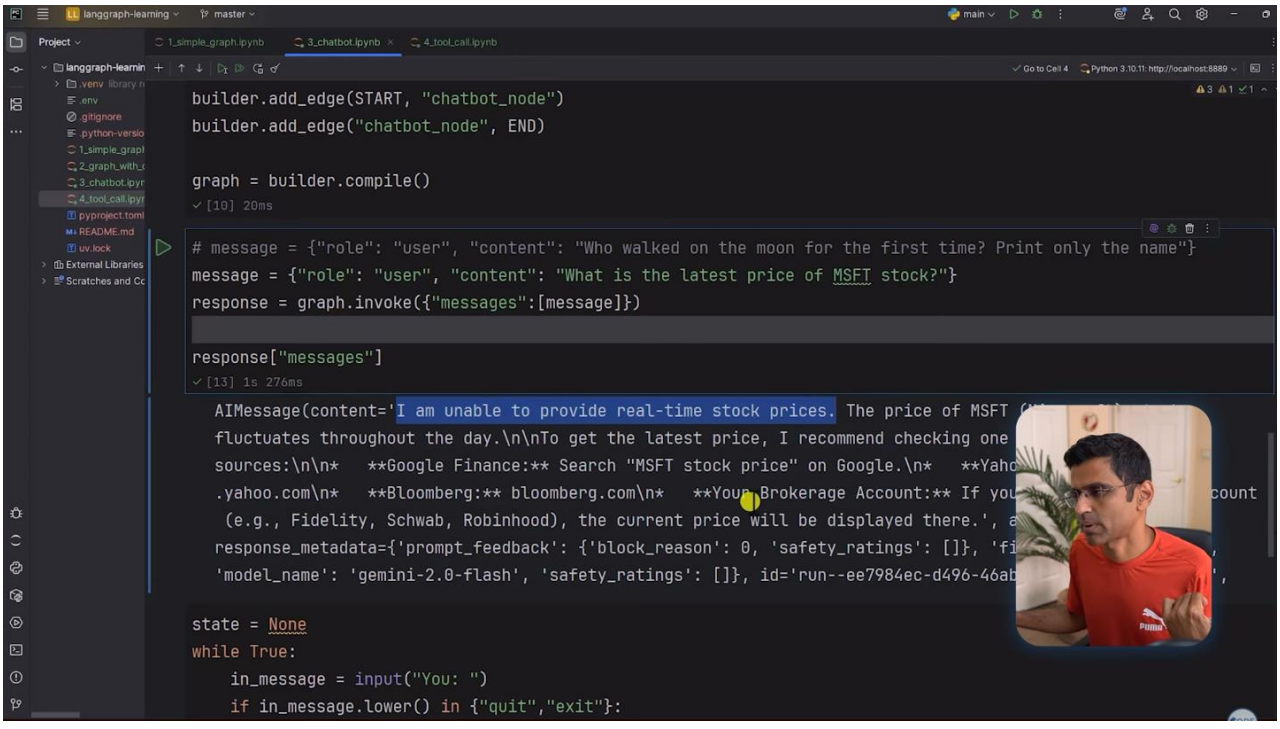




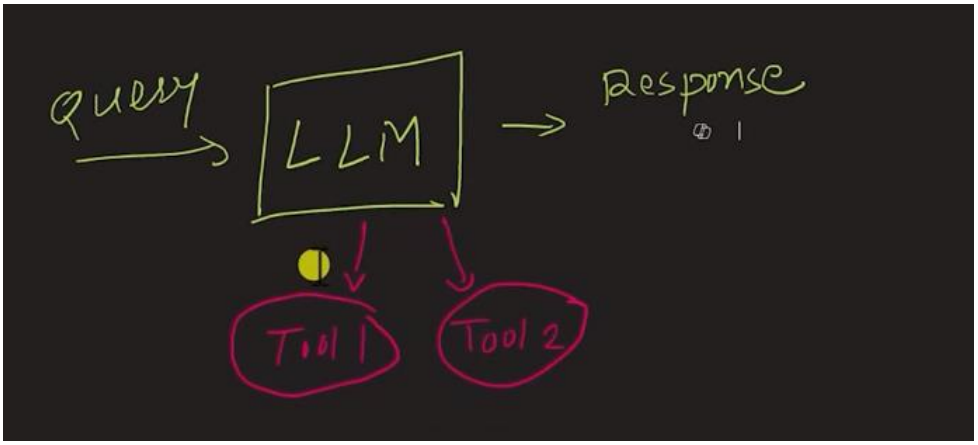


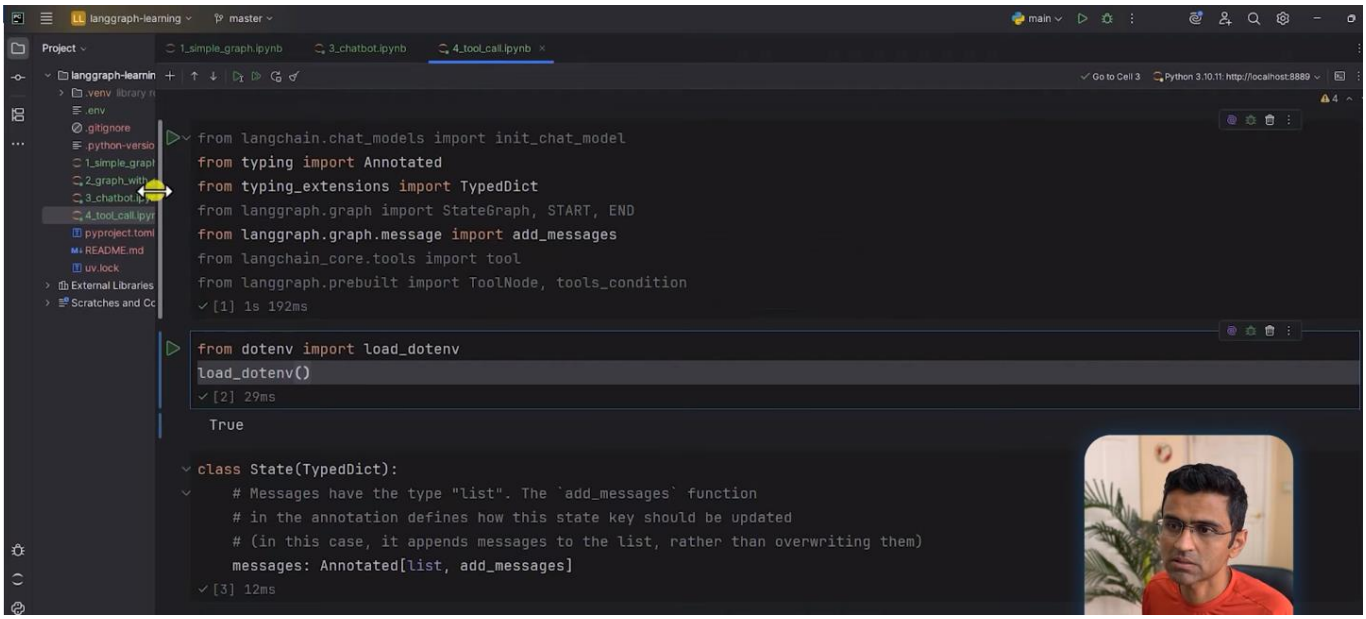


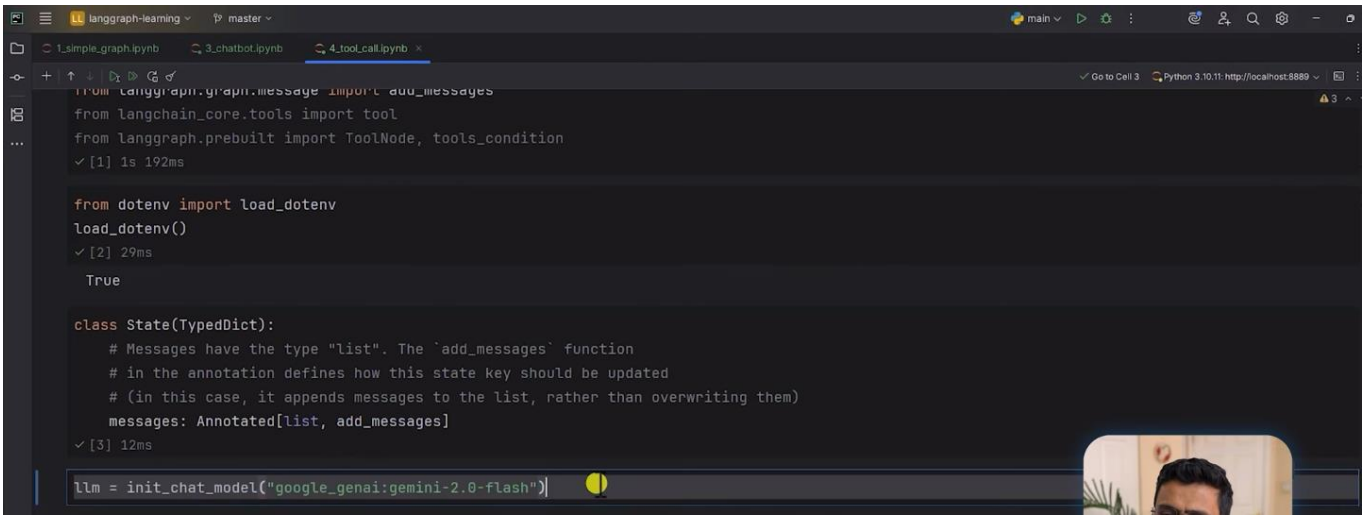
There is now a context being preserved in the messages list and the LLM can answer using it as a form of memory



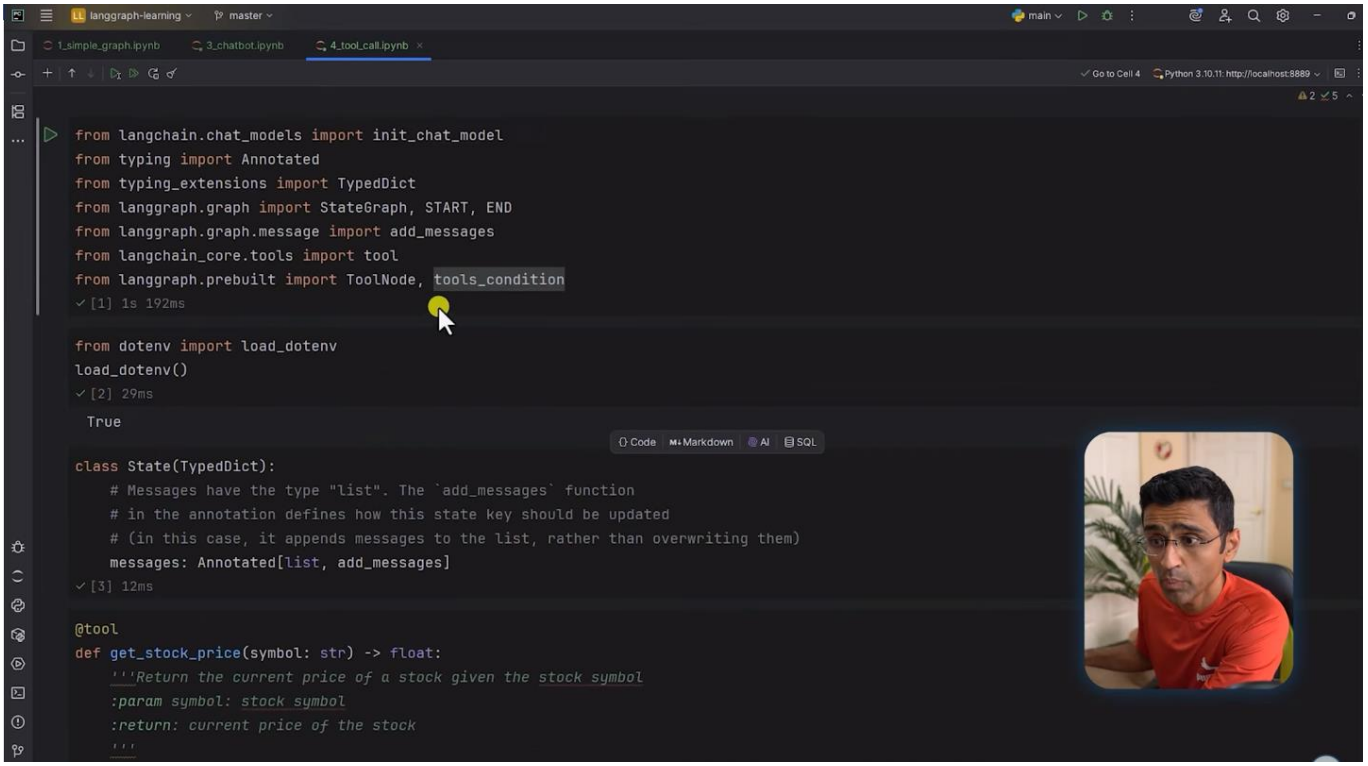
This is because the LLM’s cut off is in the past, we need to add tools for the chatbot to use

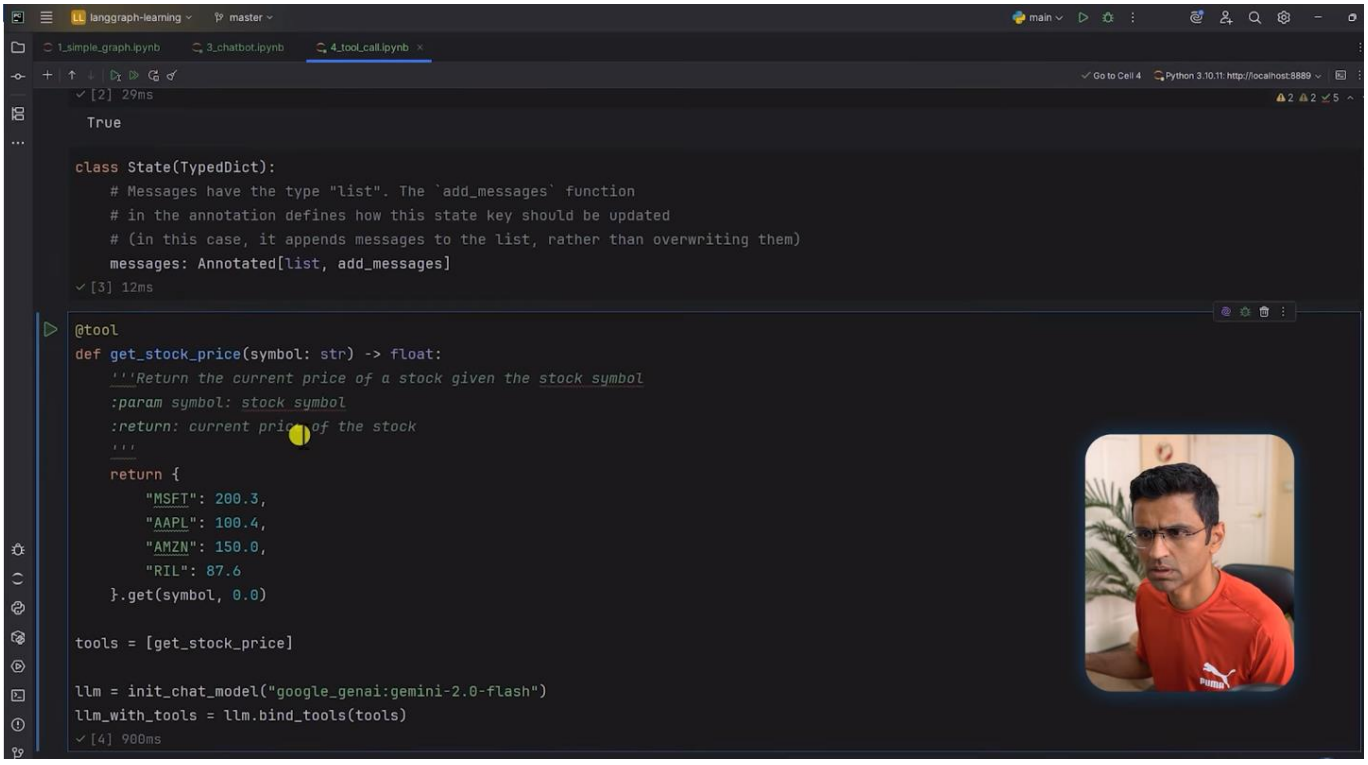


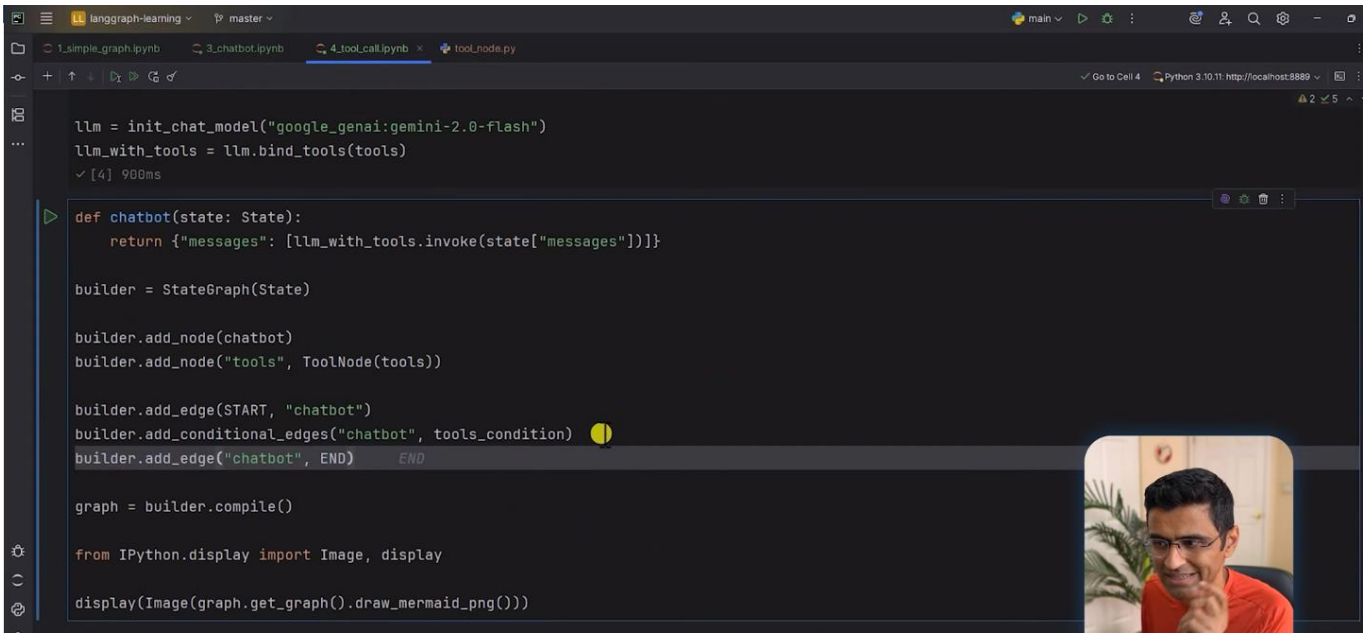




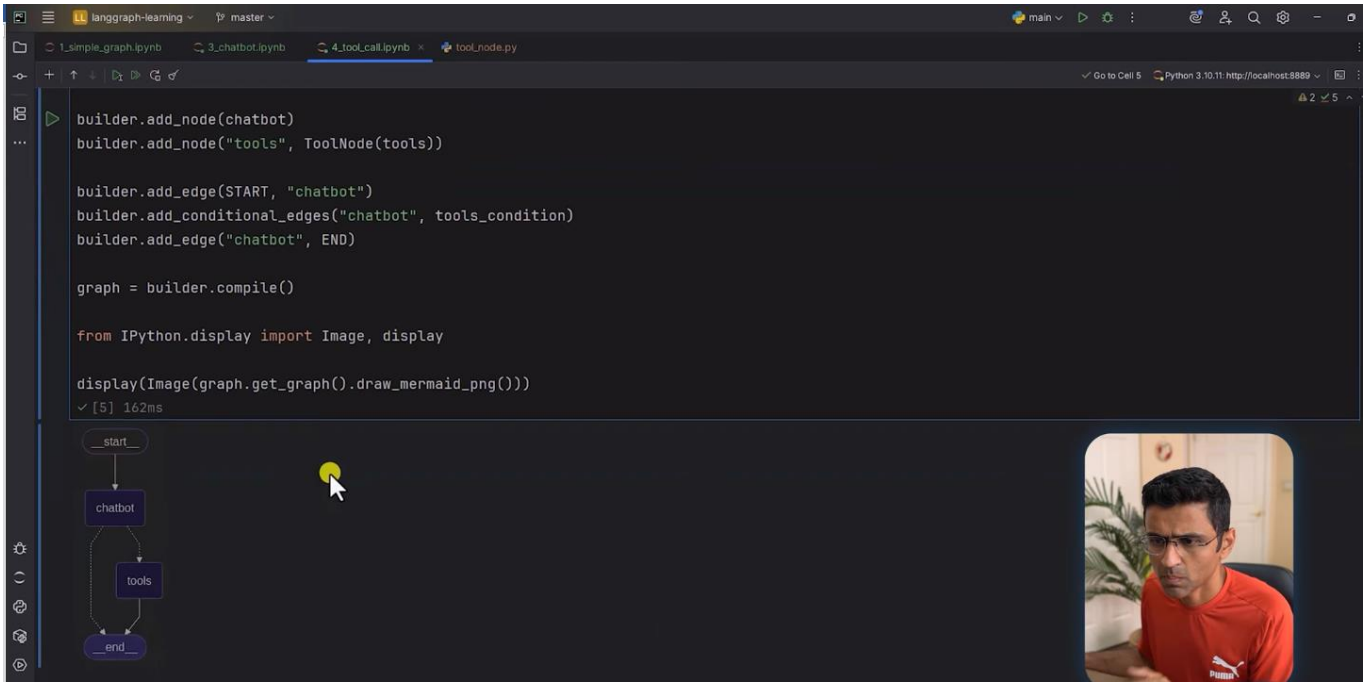
But we need to provide a tool that can get stock prices from a python function as a mock

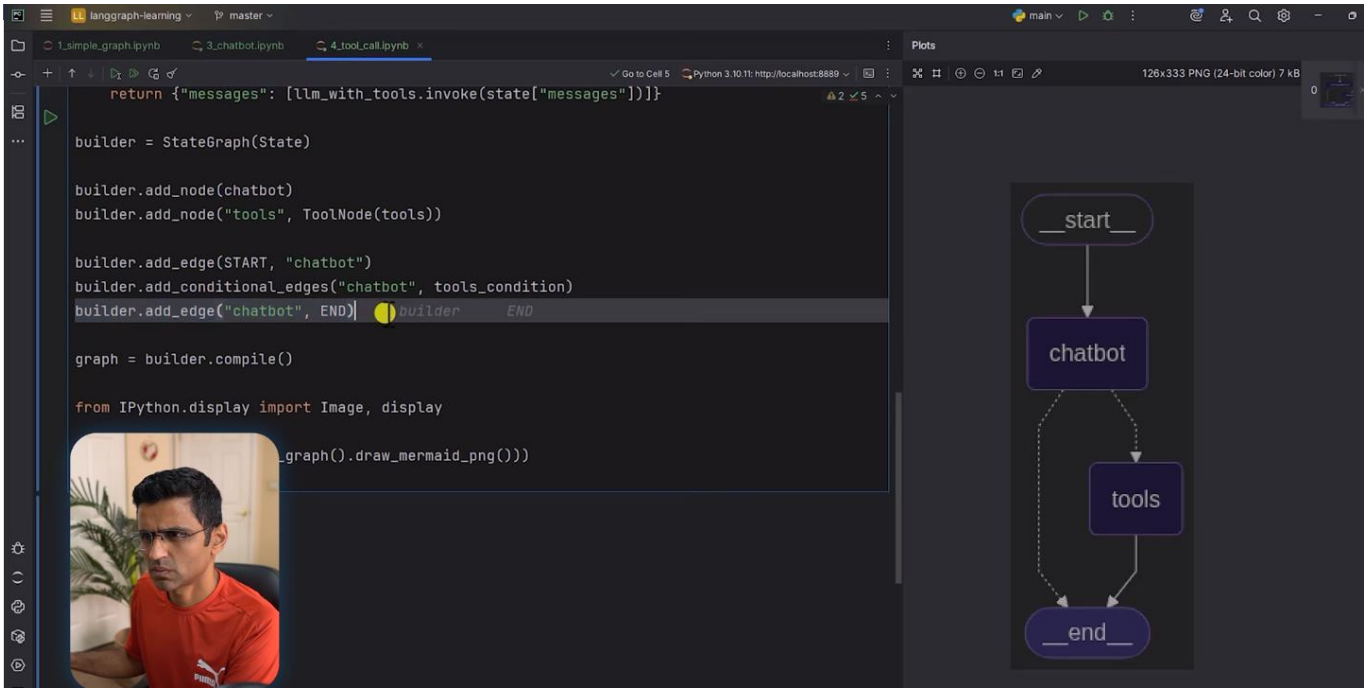

langgraph-learning
master
main
1_simple_graph.ipynb
3_chatbot.ipynb
4_tool_call.ipynb
Go to Cell 4
Python 3.10.11: http://localhost:8889
from langchain.chat_models import init_chat_model
from typing import Annotated
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, 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
Code
Markdown
AI
SQL
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
'''

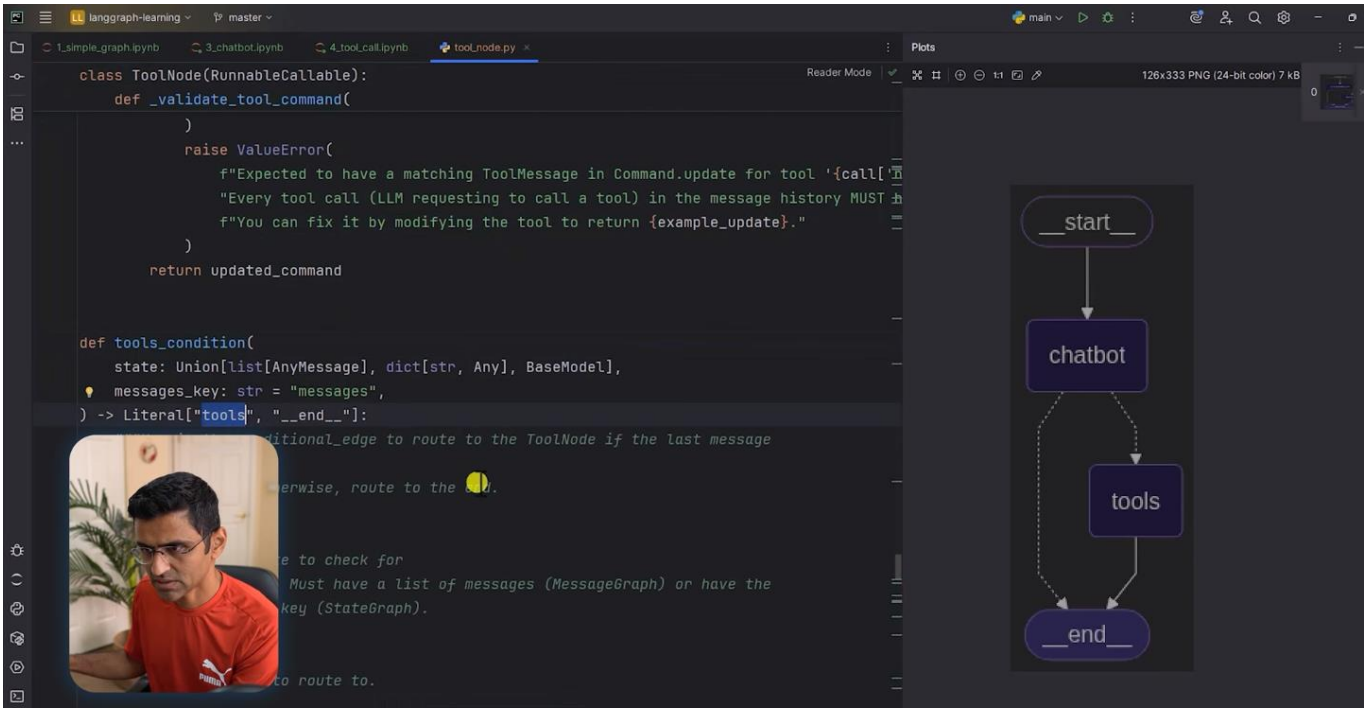

langgraph-learning
master
main
1_simple_graph.ipynb
3_chatbot.ipynb
4_tool_call.ipynb
Go to Cell 4
Python 3.10.11: http://localhost:8889
[2] 29ms
True
class State(TypedDict):
# Messages have the type "list". The `add_messages` function
# in the annotation defines how this state key should be updated
# (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

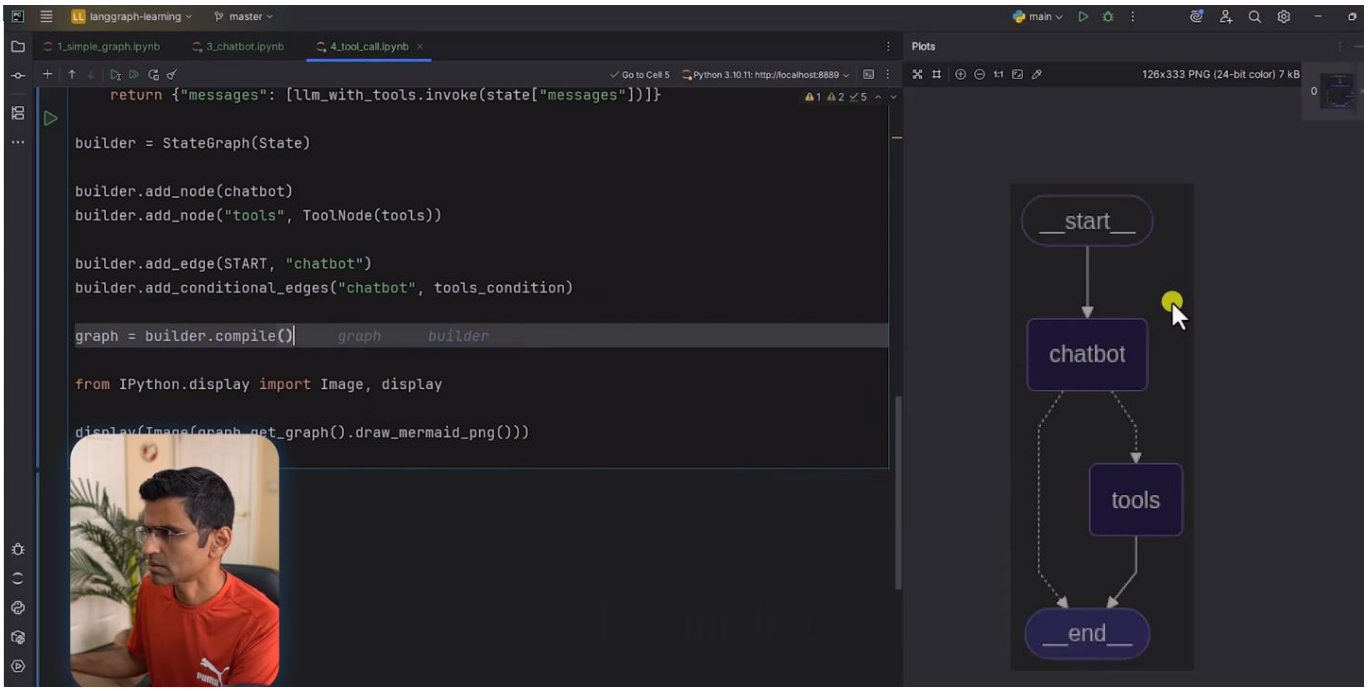


We are using a **tools** node and a **conditional edge** here because it will only be used if there is need for using the stock price tool

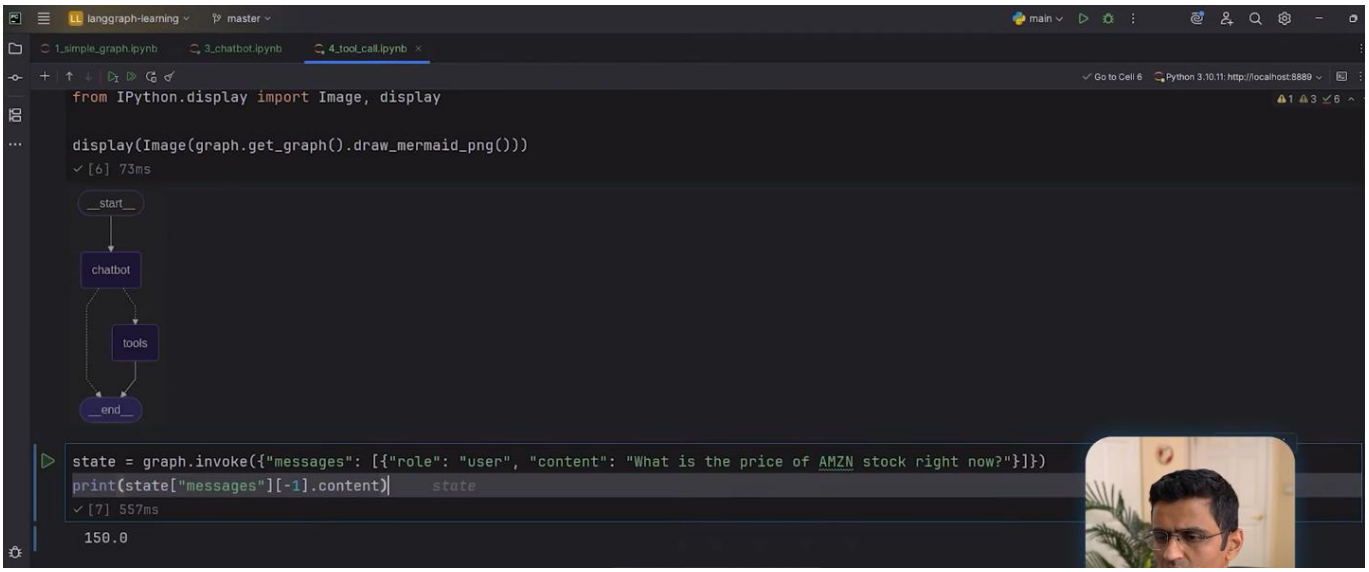


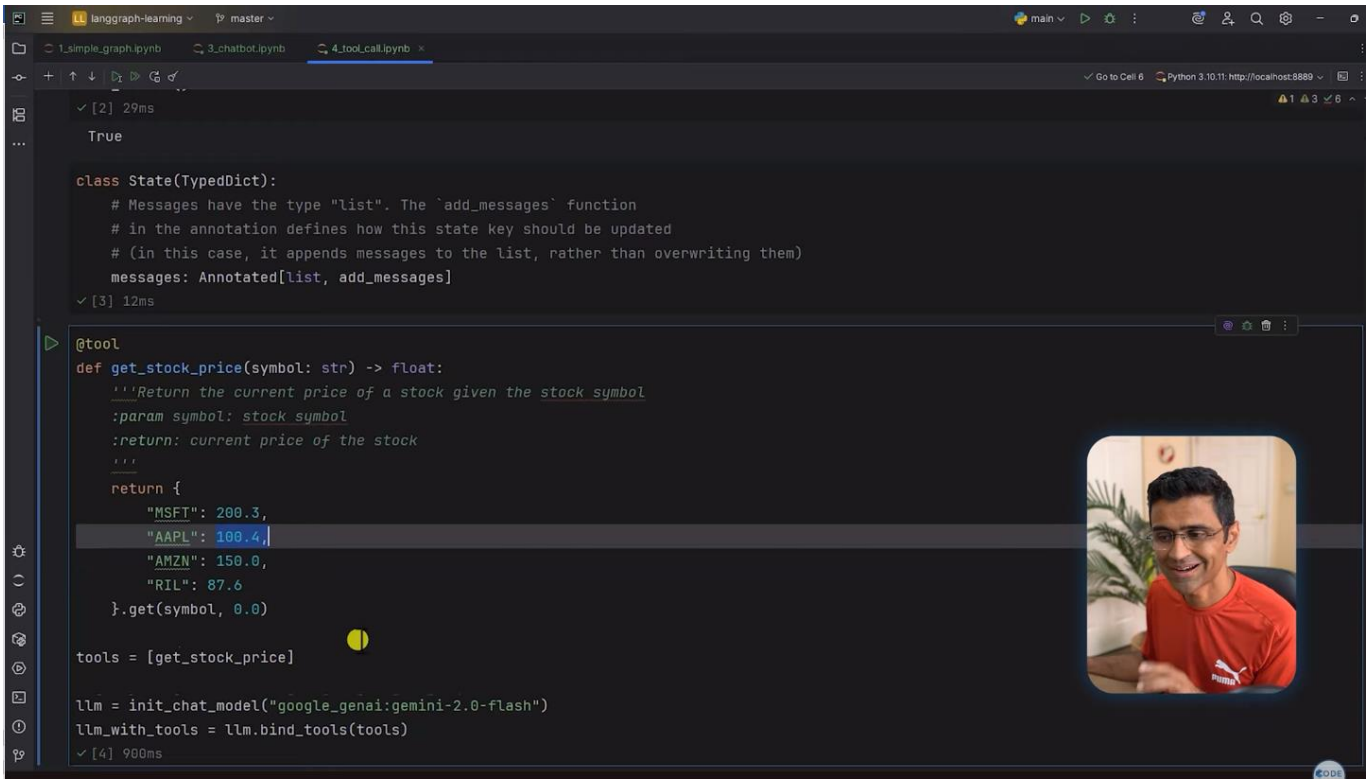

langgraph-learning
master
main
1_simple_graph.ipynb
3_chatbot.ipynb
4_tool_call.ipynb
Plots
126x333 PNG (24-bit color) 7 kB
return {"messages": [llm_with_tools.invoke(state["messages"])]}
builder = StateGraph(State)
builder.add_node(chatbot)
builder.add_node("tools", ToolNode(tools))
builder.add_edge(START, "chatbot")
builder.add_conditional_edges("chatbot", tools_condition)
builder.add_edge("chatbot", END)
graph = builder.compile()
from IPython.display import Image, display
__start__
chatbot
tools
__end__

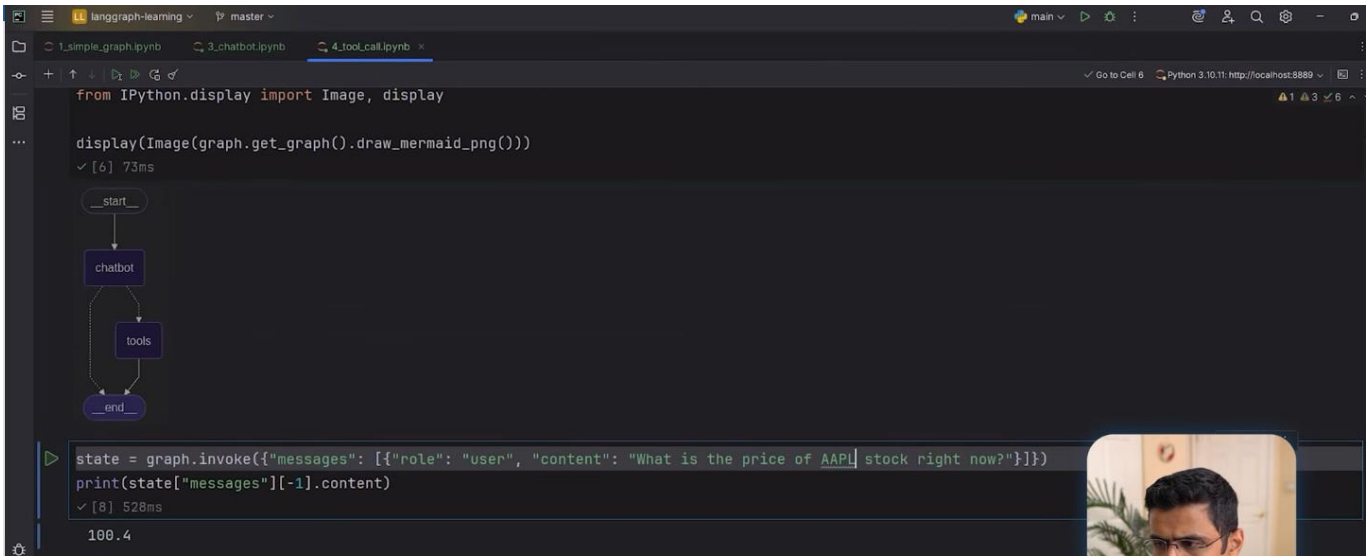

langgraph-learning
master
main
1_simple_graph.ipynb
3_chatbot.ipynb
4_tool_call.ipynb
tool_node.py
Plots
126x333 PNG (24-bit color) 7 kB
Reader Mode
class ToolNode(RunnableCallable):
def _validate_tool_command(
raise ValueError(
f"Expected to have a matching ToolMessage in Command.update for tool '{call[
"Every tool call (LLM requesting to call a tool) in the message history MUST h
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__"]:
__start__
chatbot
tools
__end__

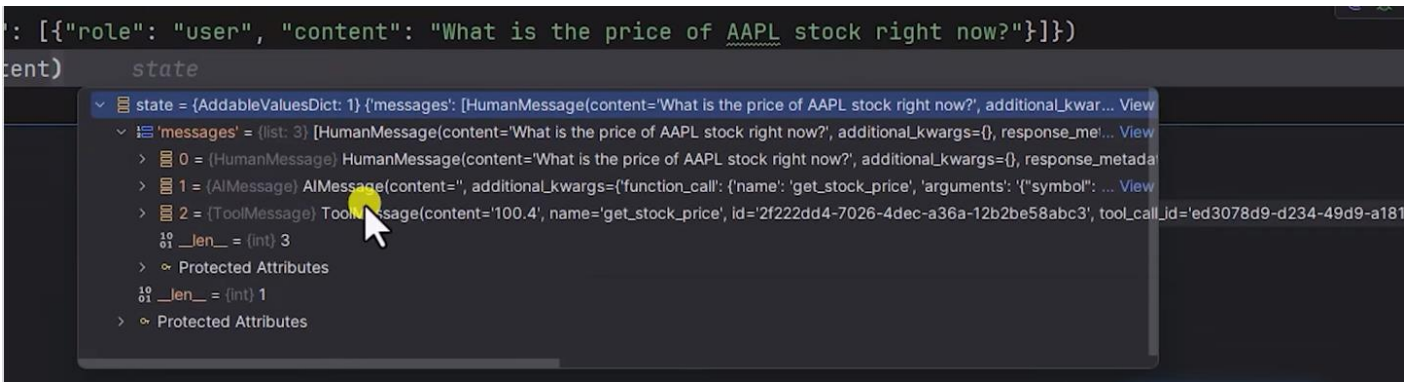


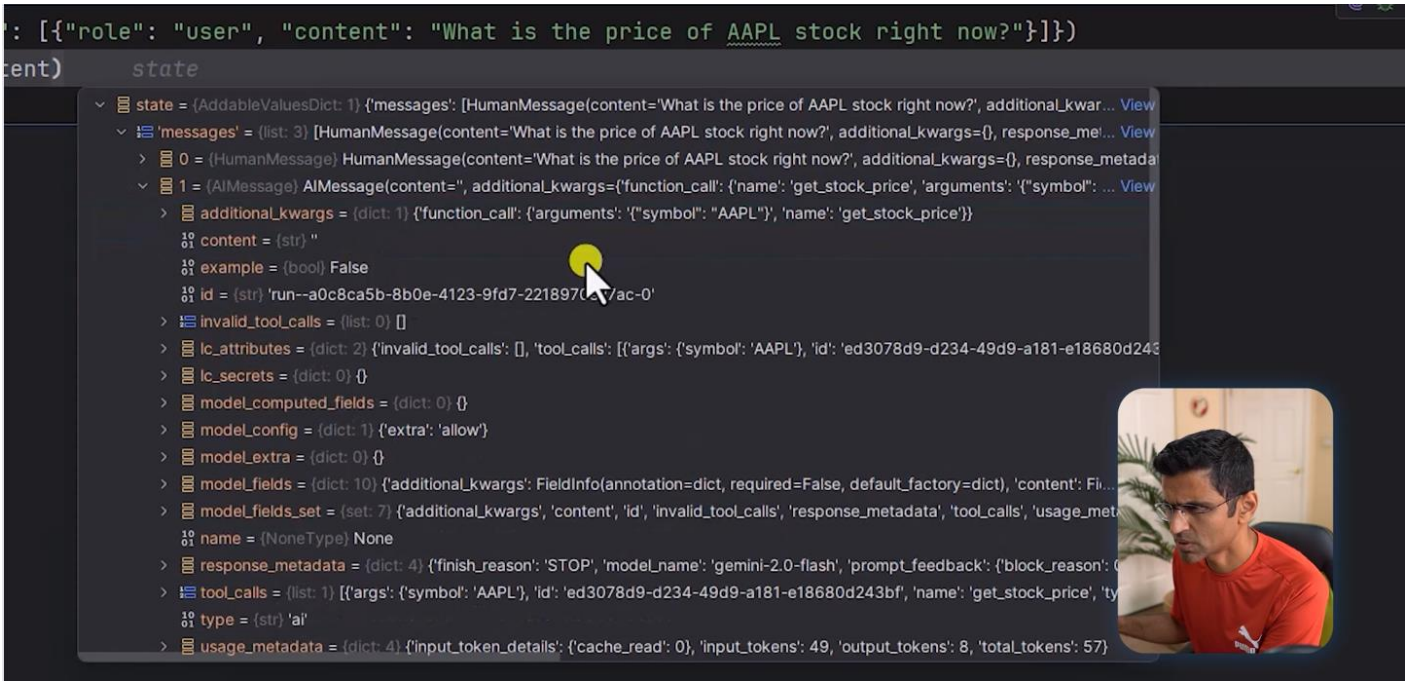
We can even remove the END line as above

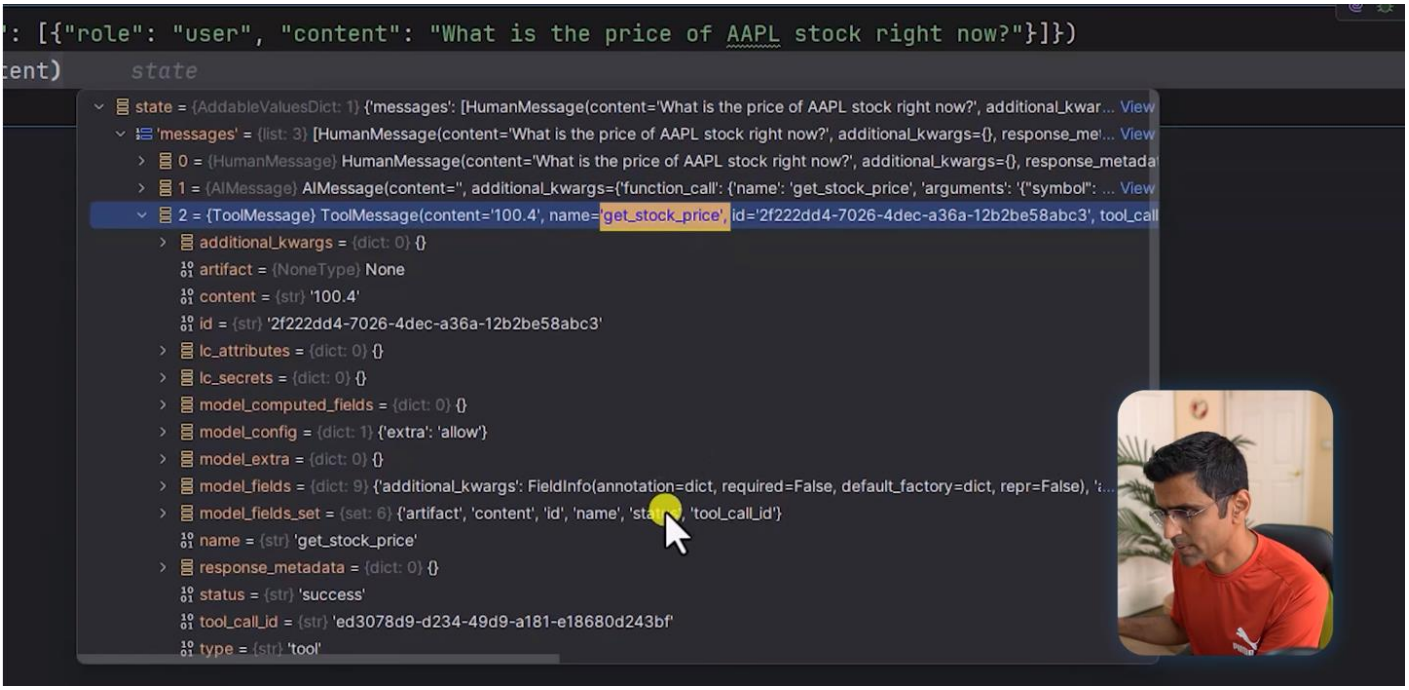


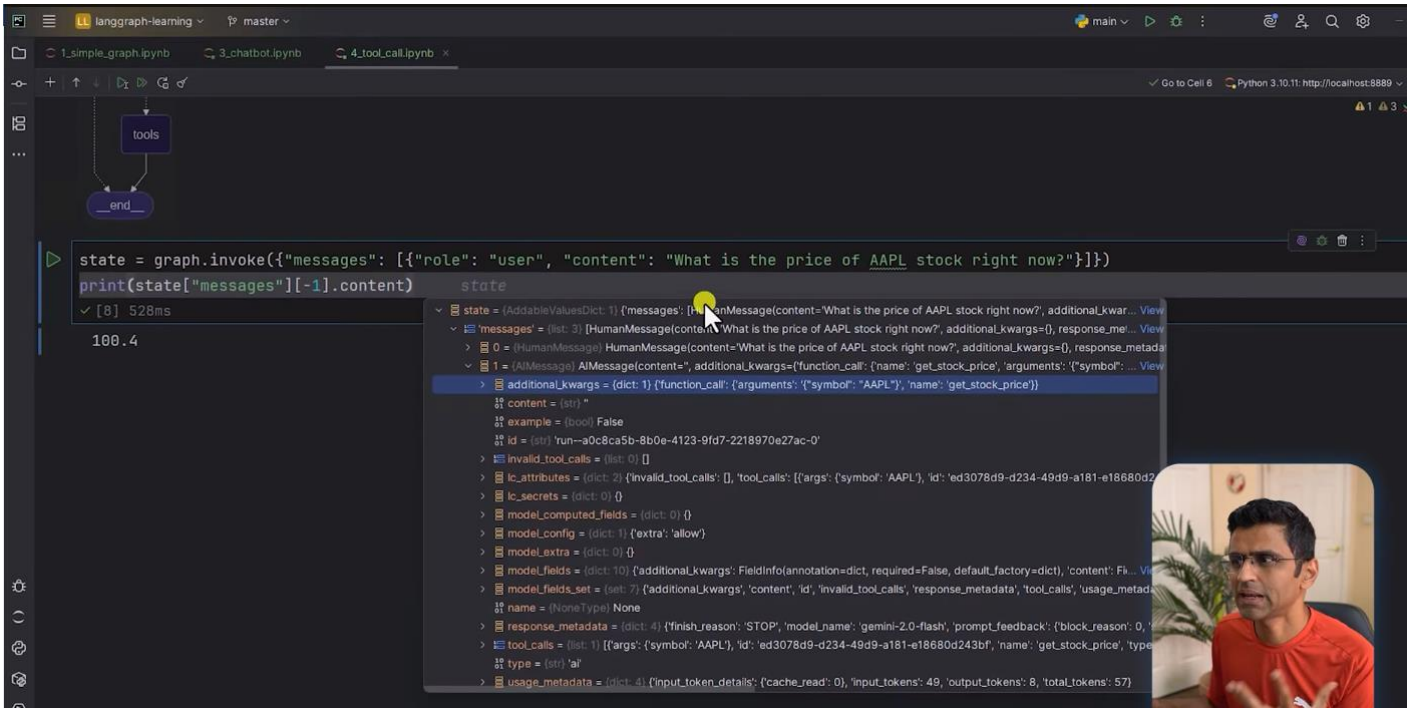

langgraph-learning
master
main
1_simple_graph.ipynb
3_chatbot.ipynb
4_tool_call.ipynb
Go to Cell 6
Python 3.10.11: http://localhost:8889
[2] 29ms
True
class State(TypedDict):
# Messages have the type "list". The `add_messages` function
# in the annotation defines how this state key should be updated
# (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

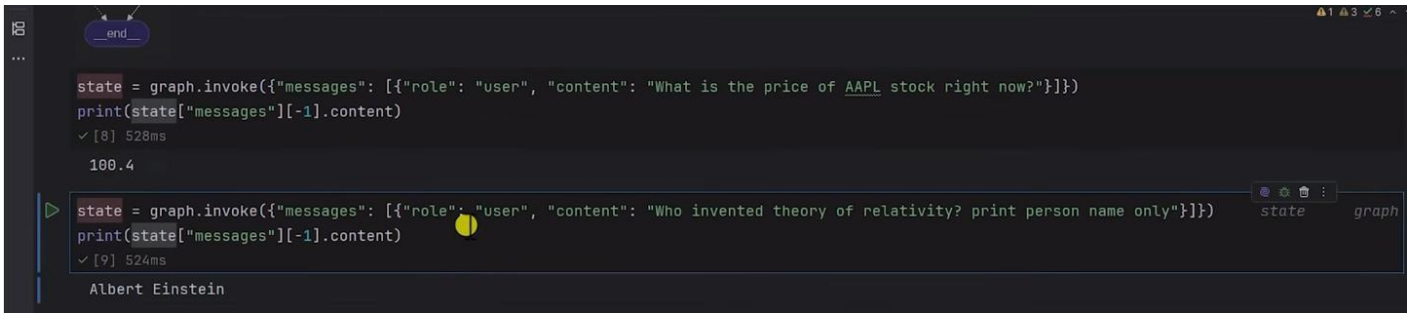

langgraph-learning
master
main
1_simple_graph.ipynb
3_chatbot.ipynb
4_tool_call.ipynb
Go to Cell 6
Python 3.10.11: http://localhost:8889
from IPython.display import Image, display
display(Image(graph.get_graph().draw_mermaid_png()))
[6] 73ms
__start__
chatbot
tools
__end__
state = graph.invoke({"messages": [{"role": "user", "content": "What is the price of AAPL stock right now?"}]})
print(state["messages"][-1].content)
[8] 528ms
100.4

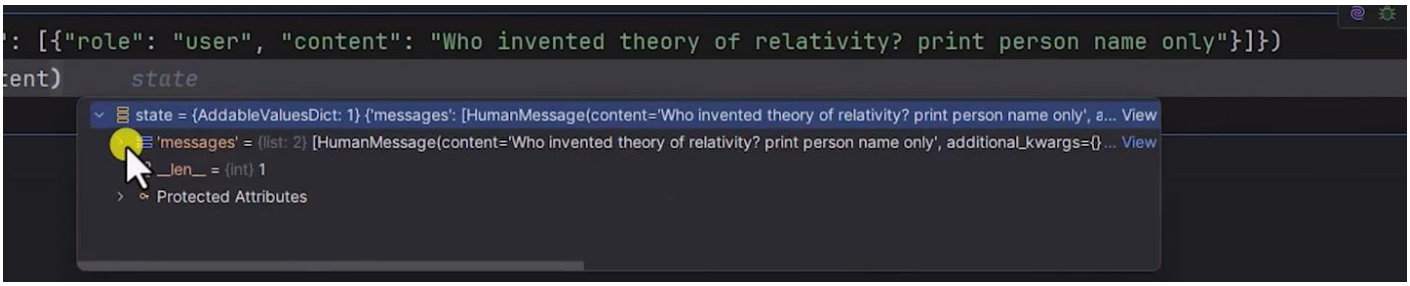

: [{"role": "user", "content": "What is the price of AAPL stock right now?"}]})
tent)
state
state = {AddableValuesDict: 1} {'messages': [HumanMessage(content='What is the price of AAPL stock right now?', additional_kwar... View
'messages' = {list: 3} [HumanMessage(content='What is the price of AAPL stock right now?', additional_kwargs={}, response_me... View
0 = {HumanMessage} HumanMessage(content='What is the price of AAPL stock right now?', additional_kwargs={}, response_metada
1 = {AIMessage} AIMessage(content='', additional_kwargs={'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
__len__ = {int} 3
Protected Attributes
__len__ = {int} 1
Protected Attributes

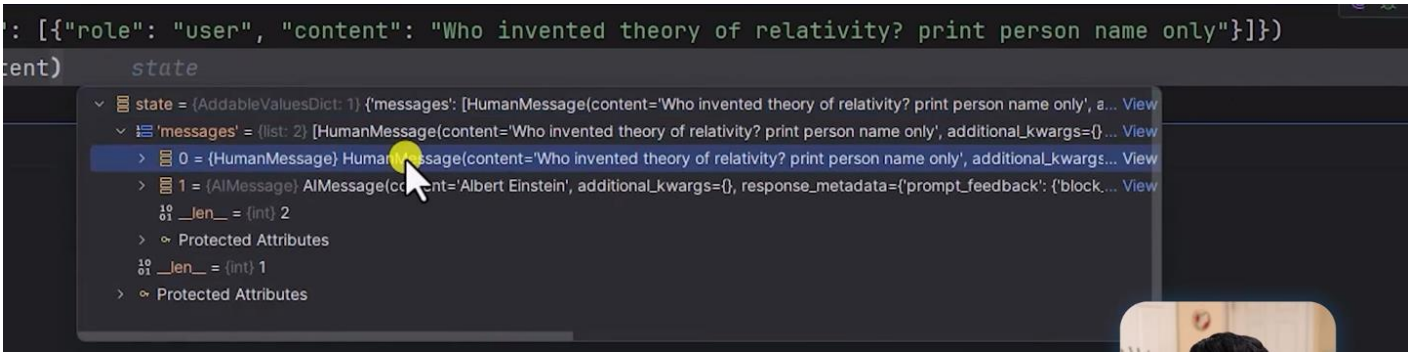

: [{"role": "user", "content": "What is the price of AAPL stock right now?"}]})
tent) state
state = {AddableValuesDict: 1} {'messages': [HumanMessage(content='What is the price of AAPL stock right now?', additional_kwar... View
'messages' = {list: 3} [HumanMessage(content='What is the price of AAPL stock right now?', additional_kwargs={}, response_me... View
0 = {HumanMessage} HumanMessage(content='What is the price of AAPL stock right now?', additional_kwargs={}, response_metada
1 = {AIMessage} AIMessage(content='', additional_kwargs={'function_call': {'name': 'get_stock_price', 'arguments': '{"symbol": ... View
additional_kwargs = {dict: 1} {'function_call': {'arguments': '{"symbol": "AAPL"}', 'name': 'get_stock_price'}}
content = {str} ''
example = {bool} False
id = {str} 'run--a0c8ca5b-8b0e-4123-9fd7-221897
/ac-0'
invalid_tool_calls = {list: 0} []
lc_attributes = {dict: 2} {'invalid_tool_calls': [], 'tool_calls': [{'args': {'symbol': 'AAPL'}, 'id': 'ed3078d9-d234-49d9-a181-e18680d243
lc_secrets = {dict: 0} {}
model_computed_fields = {dict: 0} {}
model_config = {dict: 1} {'extra': 'allow'}
model_extra = {dict: 0} {}
model_fields = {dict: 10} {'additional_kwargs': FieldInfo(annotation=dict, required=False, default_factory=dict), 'content': Fi...
model_fields_set = {set: 7} {'additional_kwargs', 'content', 'id', 'invalid_tool_calls', 'response_metadata', 'tool_calls', 'usage_met
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
type = {str} 'ai'
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?"}]})
tent) state
state = {AddableValuesDict: 1} {'messages': [HumanMessage(content='What is the price of AAPL stock right now?', additional_kwar... View
'messages' = {list: 3} [HumanMessage(content='What is the price of AAPL stock right now?', additional_kwargs={}, response_me... View
0 = {HumanMessage} HumanMessage(content='What is the price of AAPL stock right now?', additional_kwargs={}, response_metada
1 = {AIMessage} AIMessage(content='', additional_kwargs={'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
additional_kwargs = {dict: 0} {}
artifact = {NoneType} None
content = {str} '100.4'
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_kwargs': FieldInfo(annotation=dict, required=False, default_factory=dict, repr=False), 'a...
model_fields_set = {set: 6} {'artifact', 'content', 'id', 'name', 'sta
', 'tool_call_id'}
name = {str} 'get_stock_price'
response_metadata = {dict: 0} {}
status = {str} 'success'
tool_call_id = {str} 'ed3078d9-d234-49d9-a181-e18680d243bf'
type = {str} 'tool'

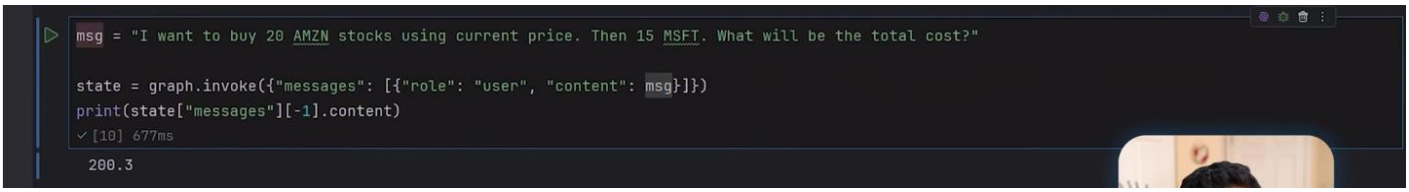






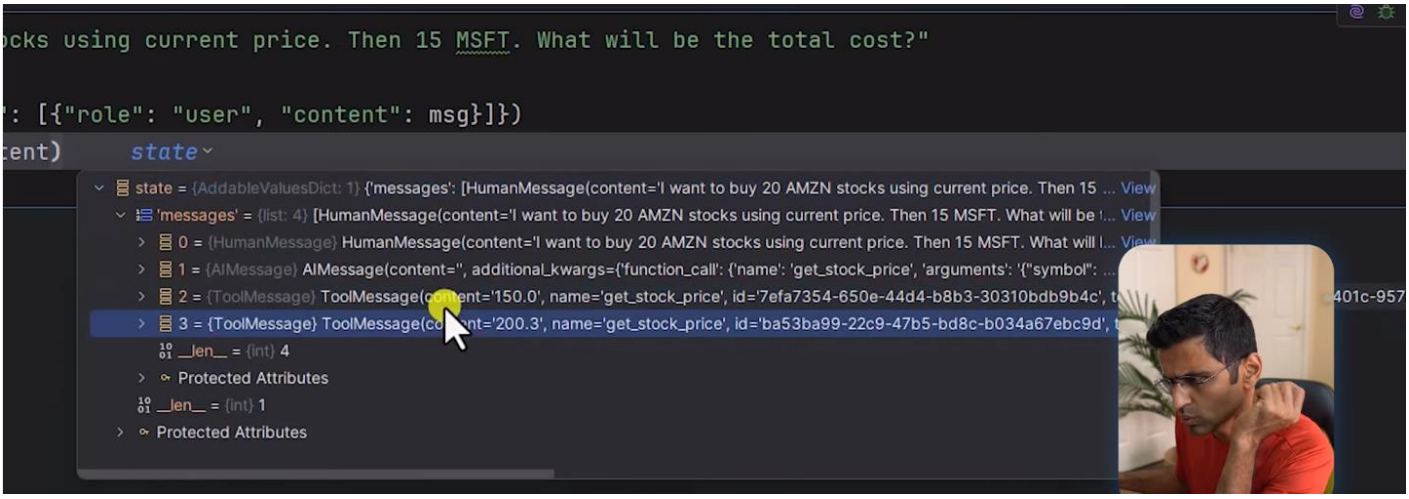


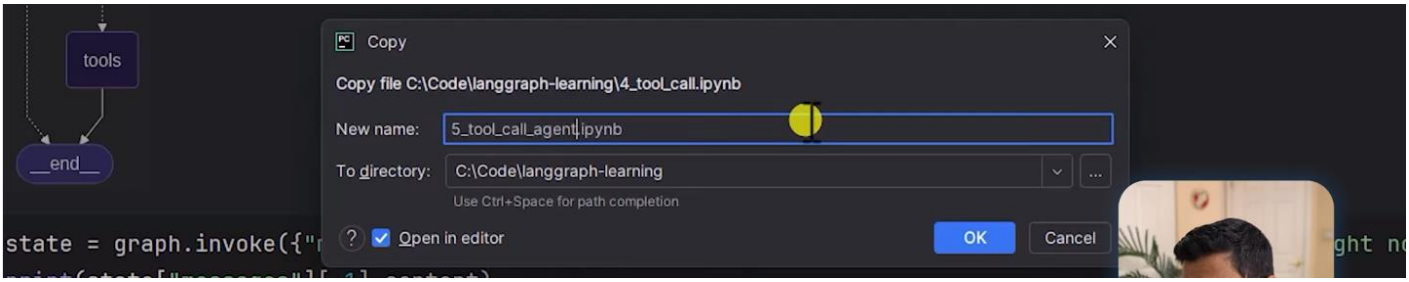
No ToolMessage seen here because the LLM correctly knows it doesn’t need a tool for that question.

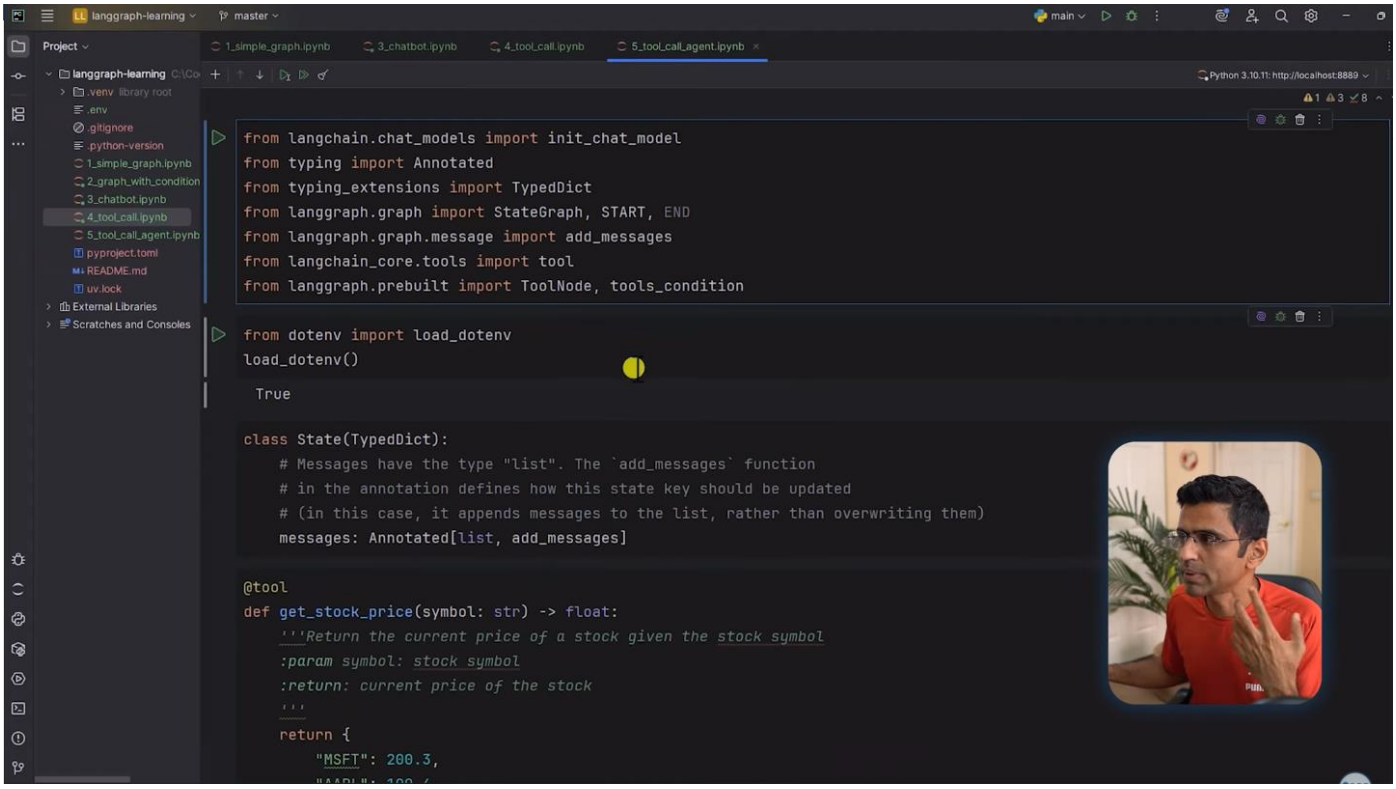


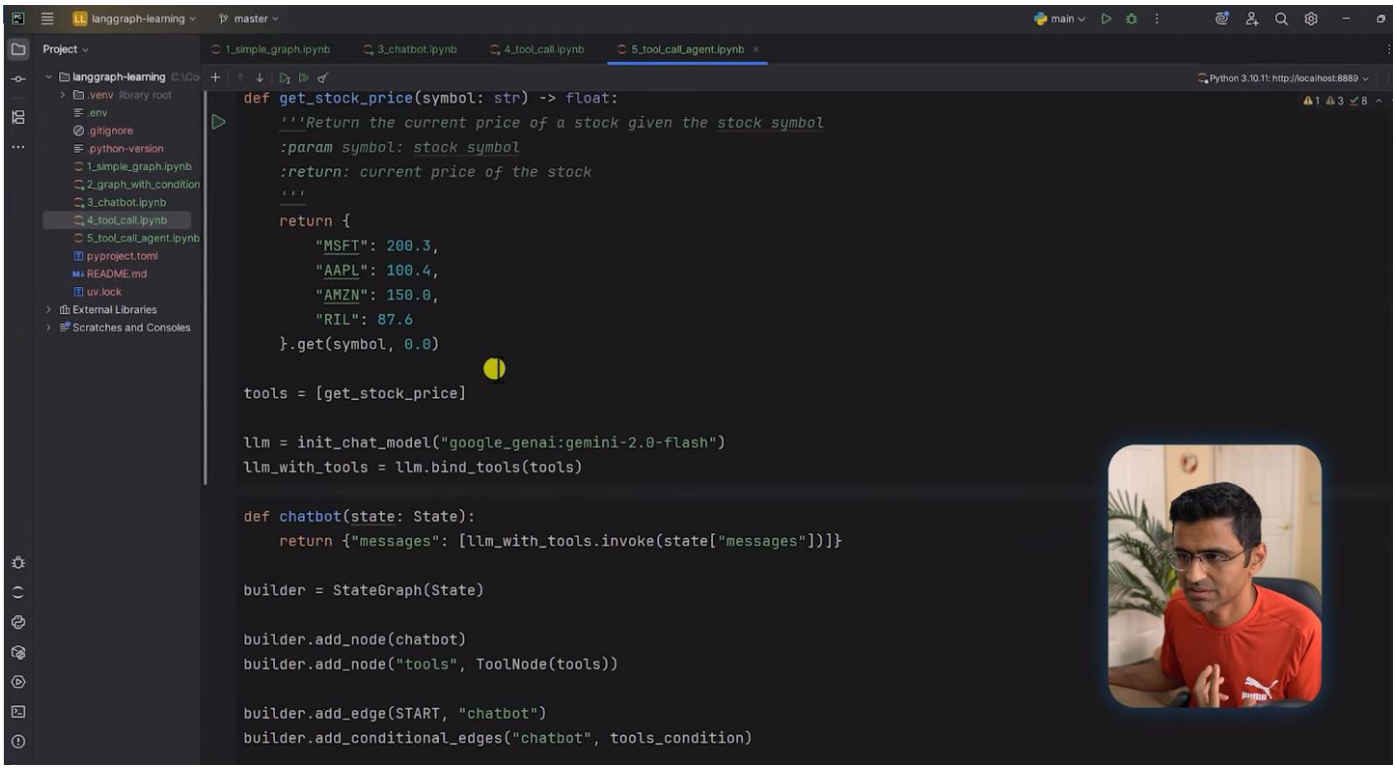
The LLM just returned the MSFT stock price using the tool and did not do the expected Math since it did not return the value back to the LLM to reflect on.

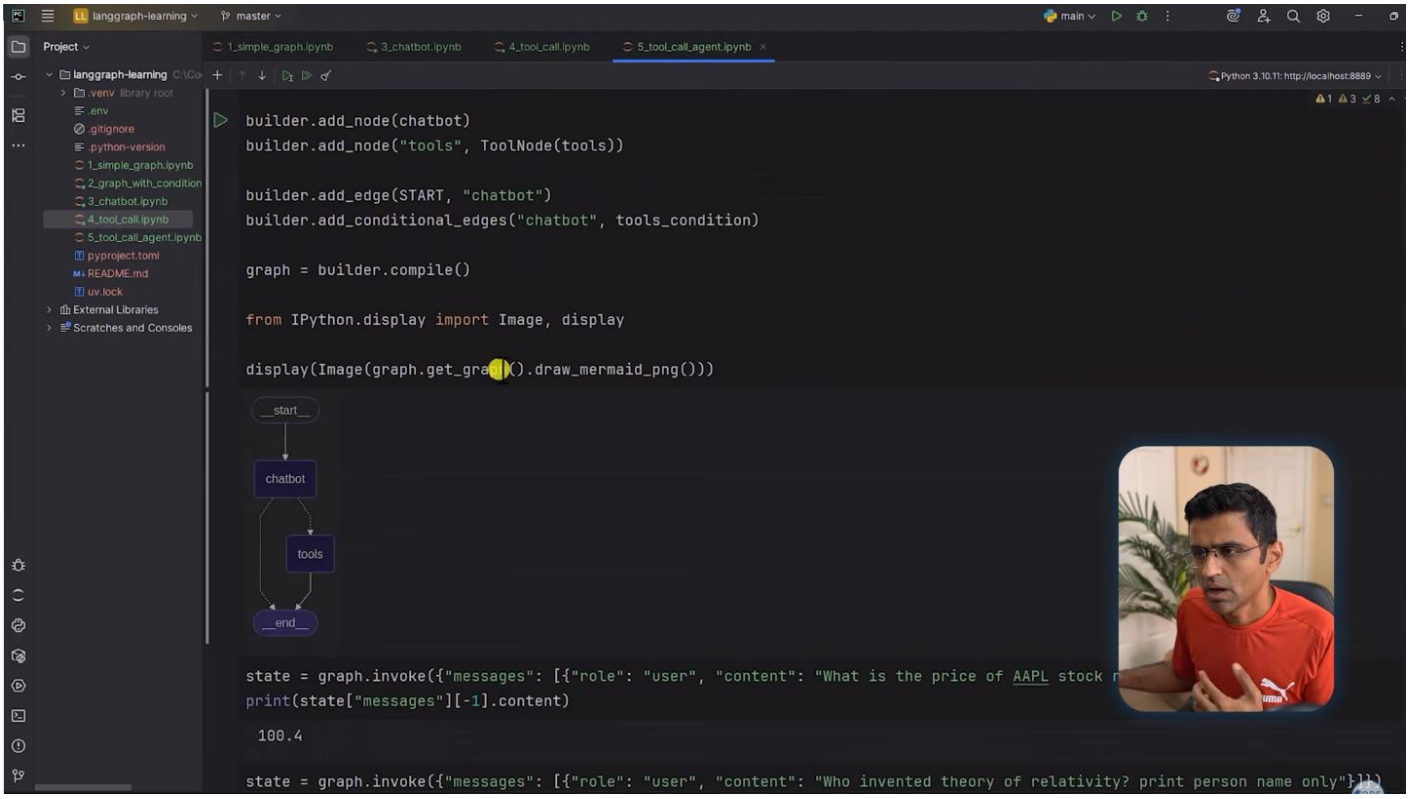
Let us copy the chatbot code and use it to create our agent as below



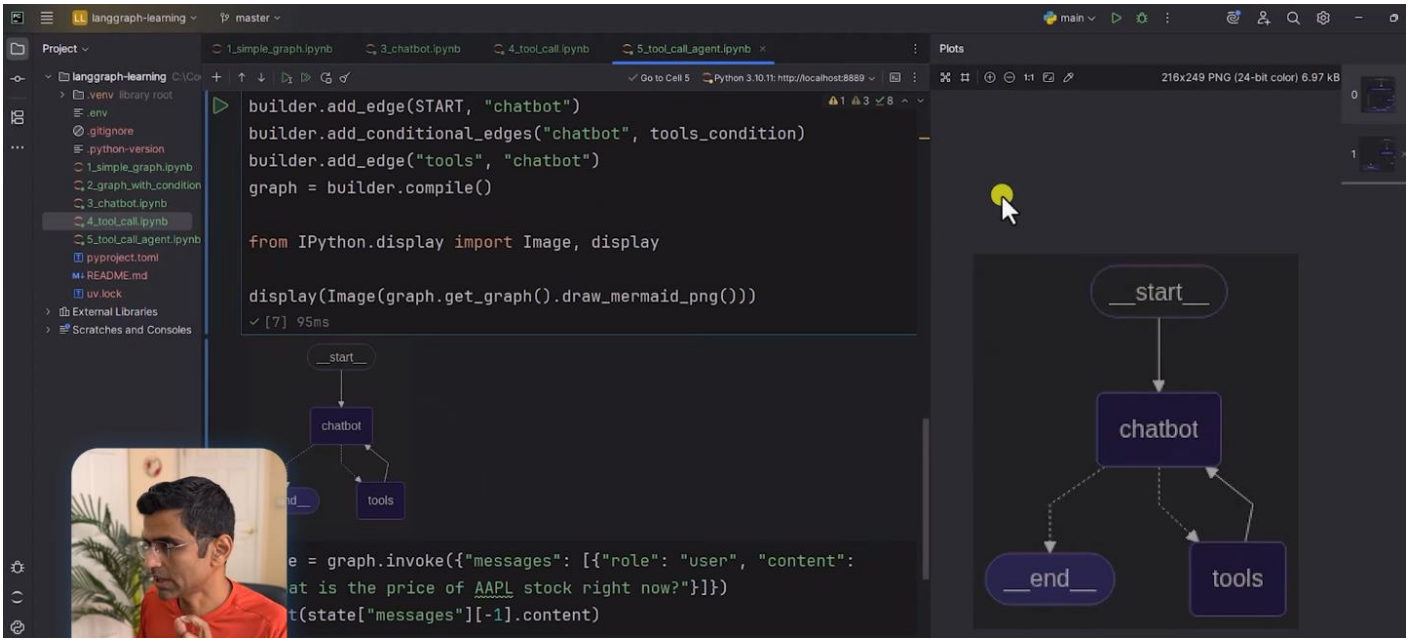


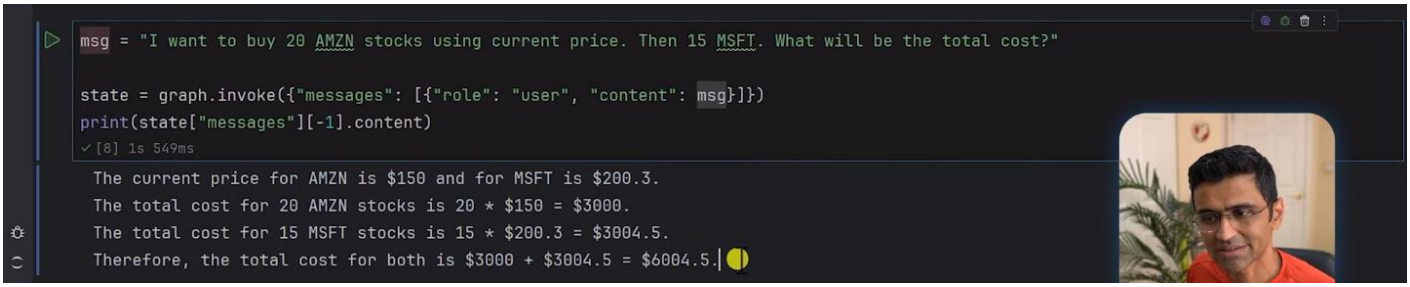

langgraph-learning
master
main
Project
1_simple_graph.ipynb
3_chatbot.ipynb
4_tool_call.ipynb
5_tool_call_agent.ipynb
Python 3.10.11: http://localhost:8889
langgraph-learning
.venv library root
.env
.gitignore
.python-version
1_simple_graph.ipynb
2_graph_with_condition
3_chatbot.ipynb
4_tool_call.ipynb
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.graph.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]
@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,

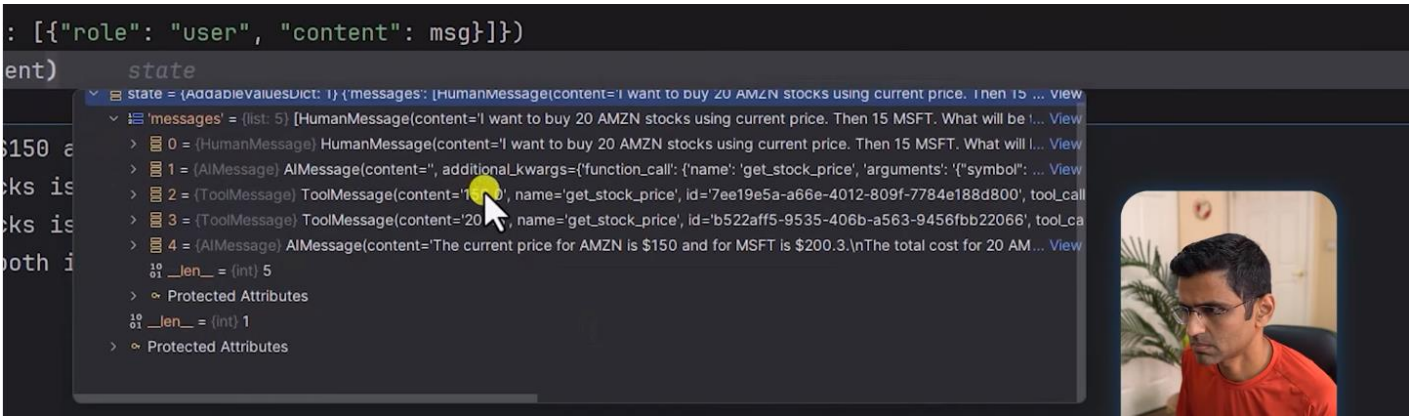

langgraph-learning
master
main
Project
1_simple_graph.ipynb
3_chatbot.ipynb
4_tool_call.ipynb
5_tool_call_agent.ipynb
Python 3.10.11: http://localhost:8889
langgraph-learning
.venv library root
.env
.gitignore
.python-version
1_simple_graph.ipynb
2_graph_with_condition
3_chatbot.ipynb
4_tool_call.ipynb
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)



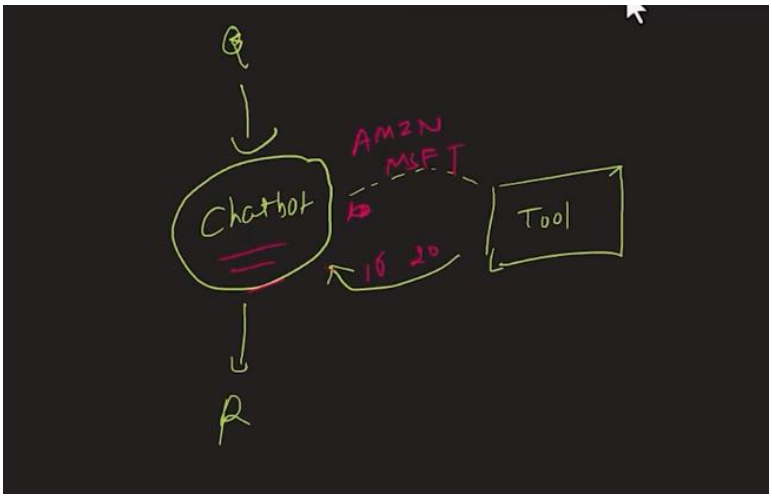
We want the tool response to be returned to the chatbot LLM to reflect on and make the next condition or go to END



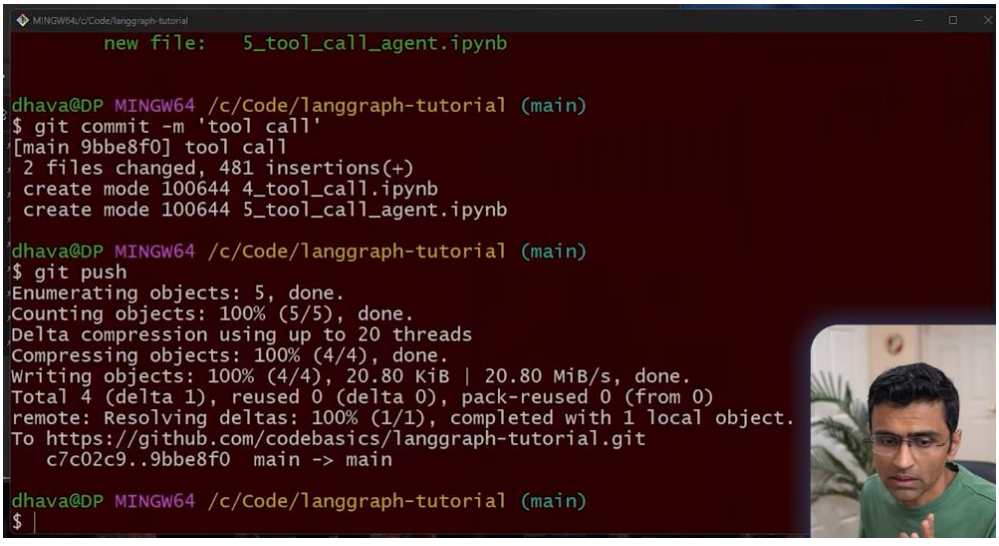


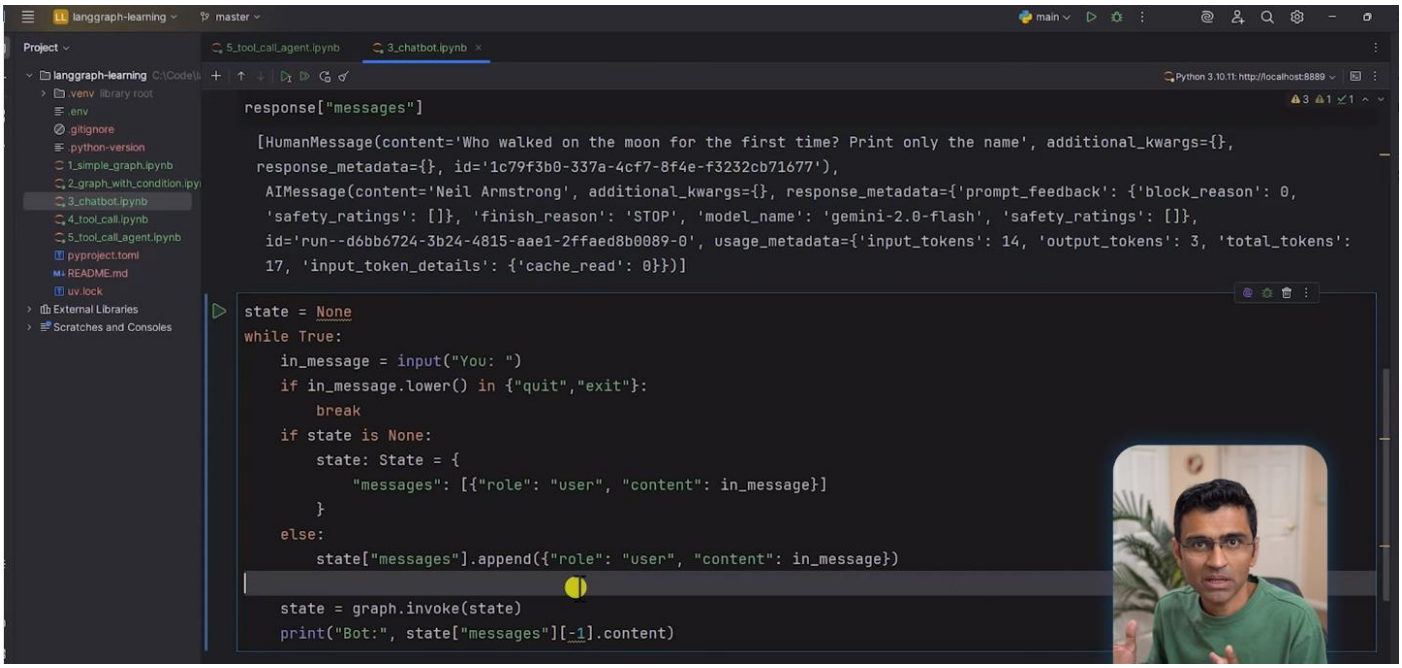


It now correctly reasoning on what next best step to make in an agentic manner.

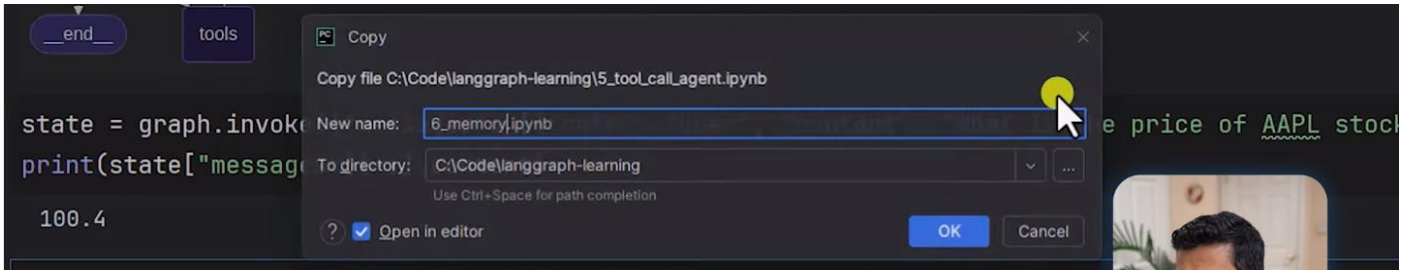


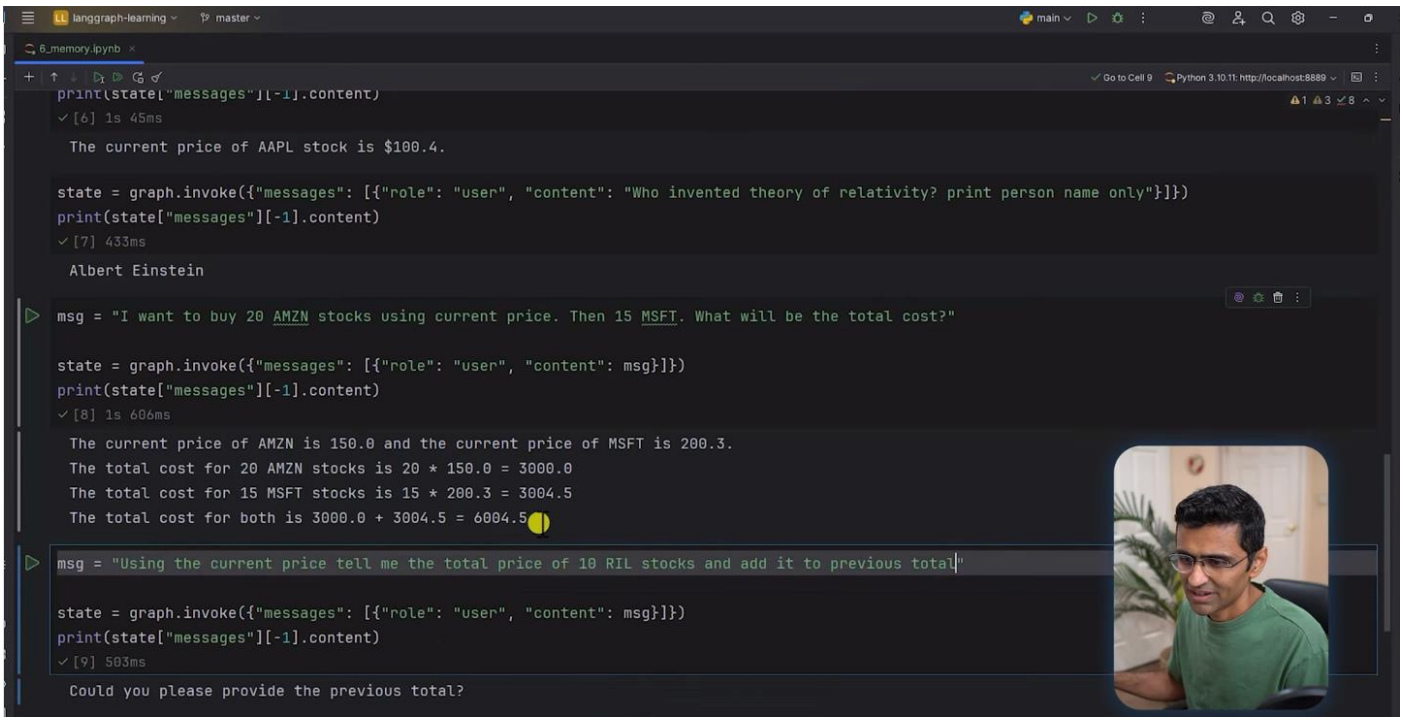
memory





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.





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.

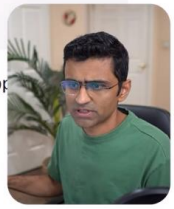
LangGraph solves this problem through **persistent checkpointing**. If you provide a `checkpointer` when compiling the graph and a `thread_id` when calling your graph, LangGraph automatically saves t each step. When you invoke the graph again using the same `thread_id`, the graph loads its save allowing 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 in and more. But first, let's add checkpointing to enable multi-turn conversations.

1. Create a `MemorySaver` checkpointer

Create a `MemorySaver` checkpointer:

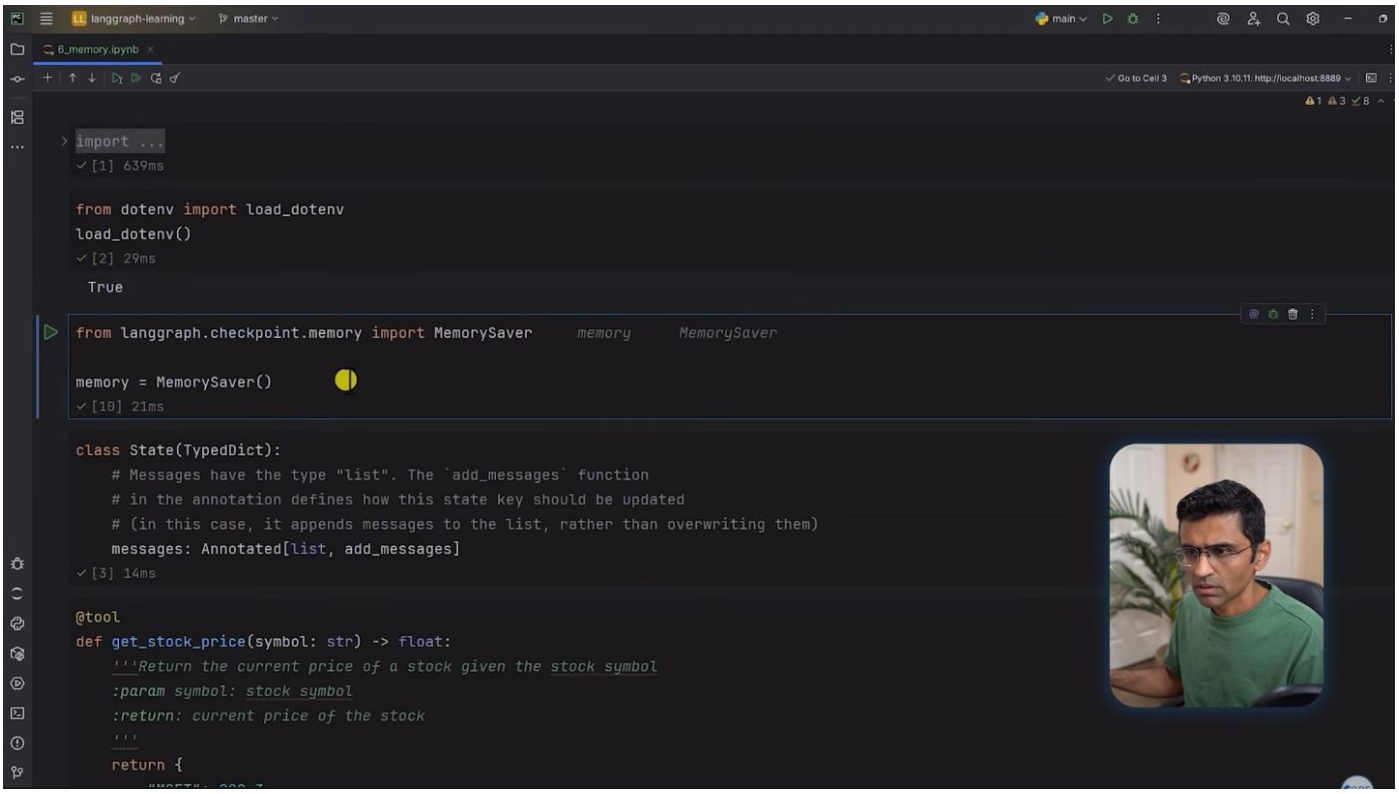
```
from langgraph.checkpoint.memory import MemorySaver

memory = MemorySaver()
```

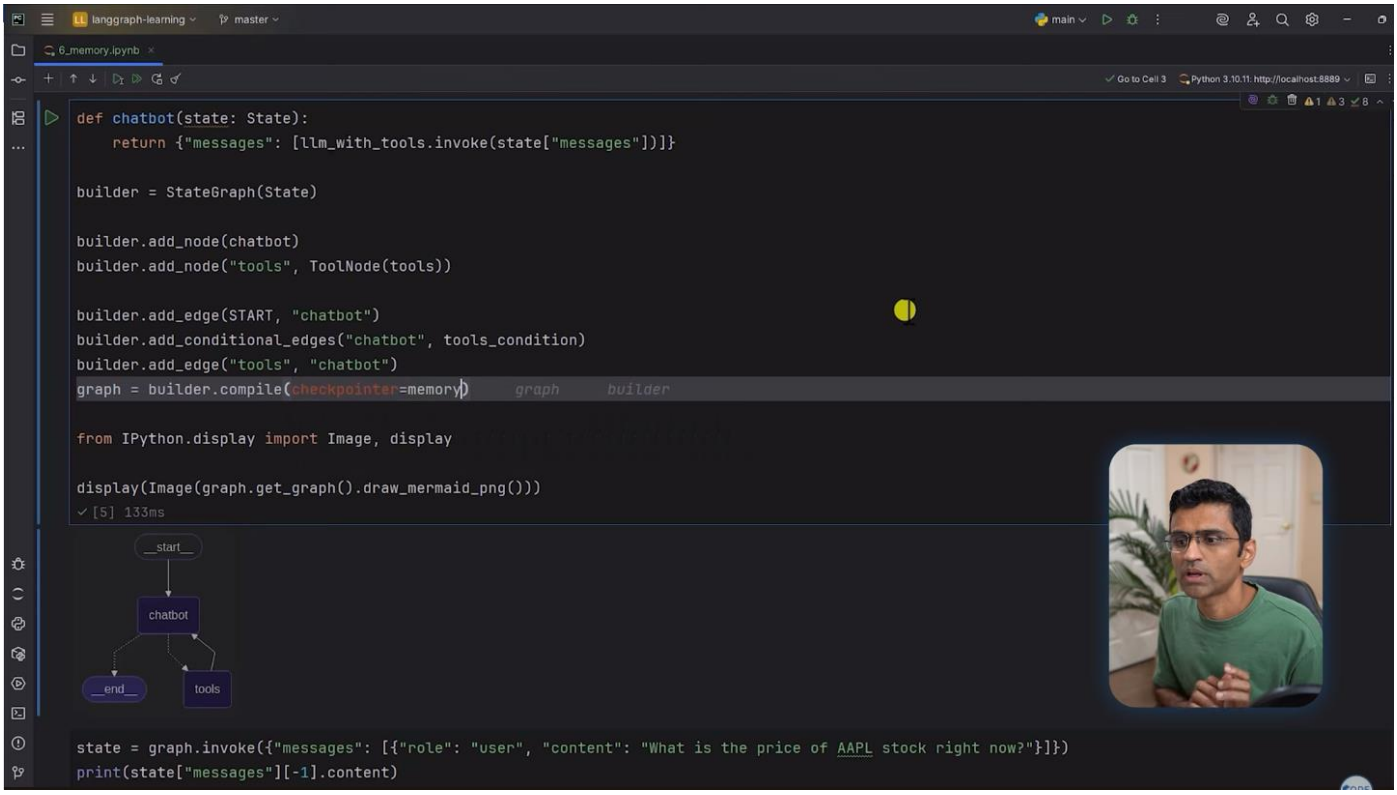


This is in-memory checkpointer, which is convenient for the tutorial. However, in a production app would likely change this to use `SqliteSaver` or `PostgresSaver` and connect a database.

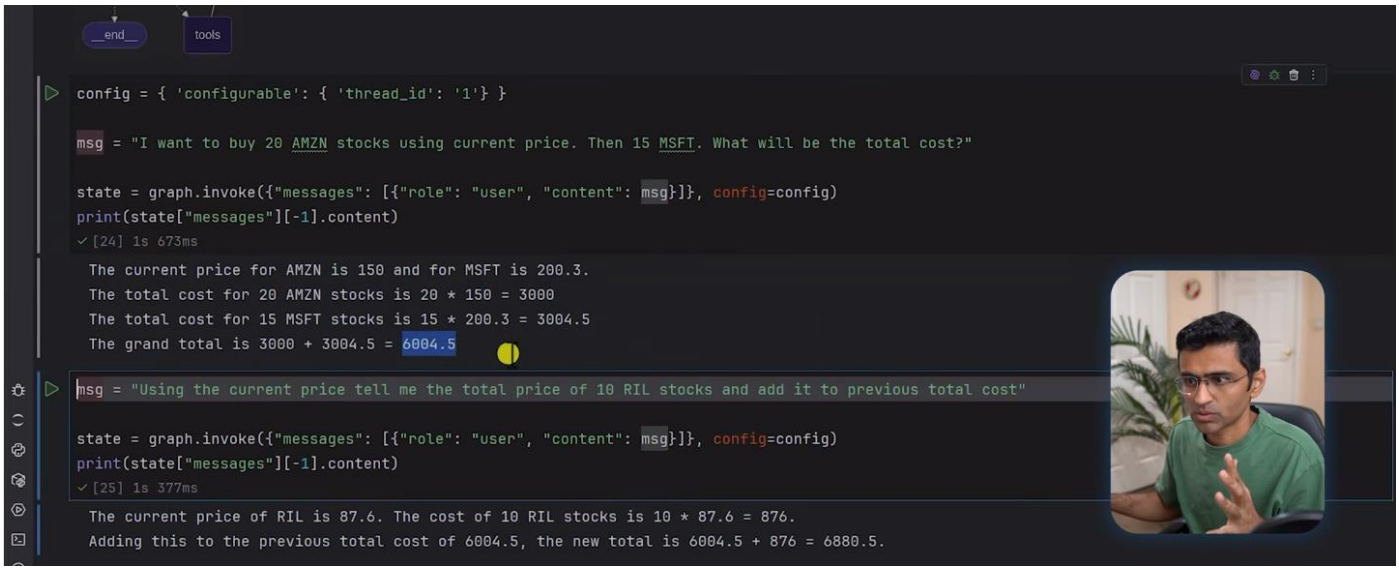
2. Compile the graph ¶

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

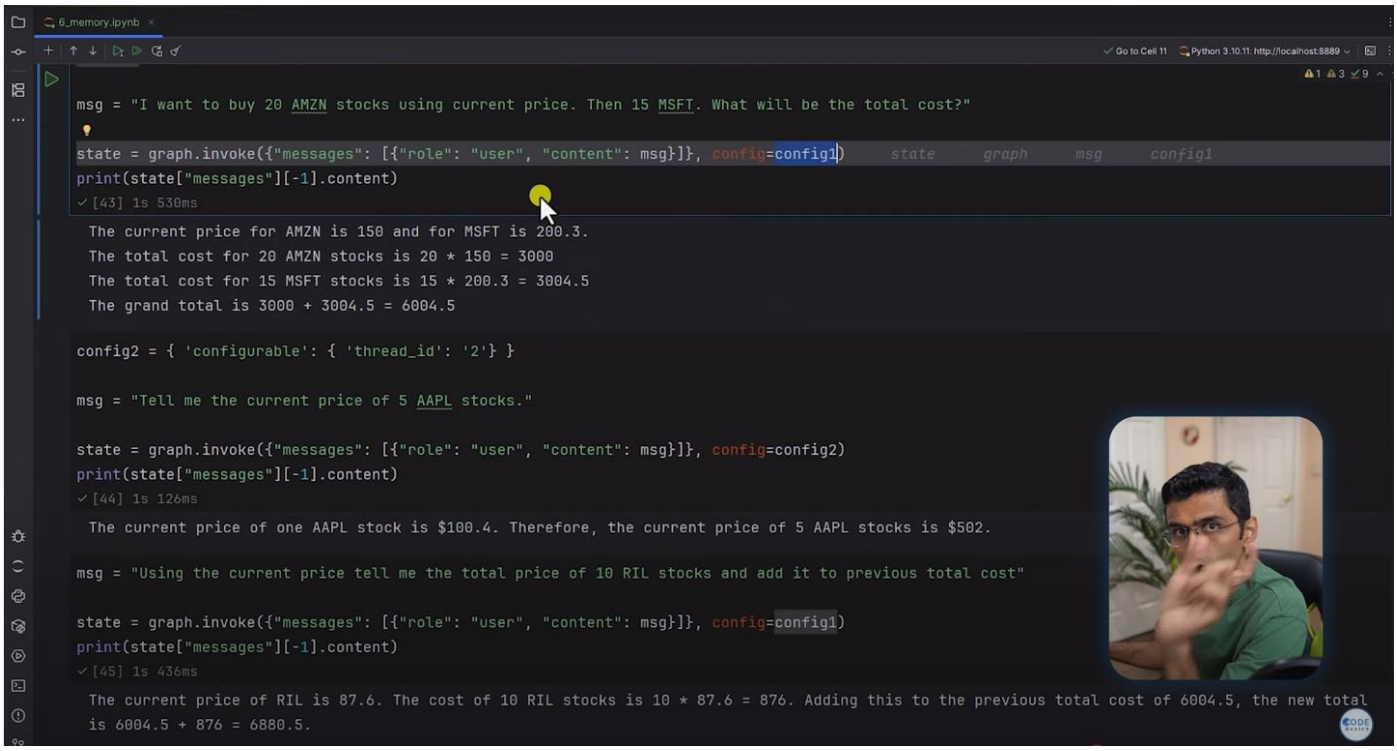




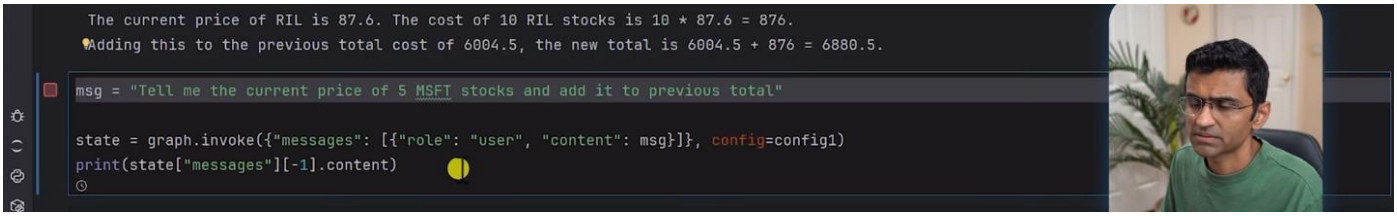
When you compile the graph, add the **memory** object to the **checkpointer** as above. Then add the **config** as below

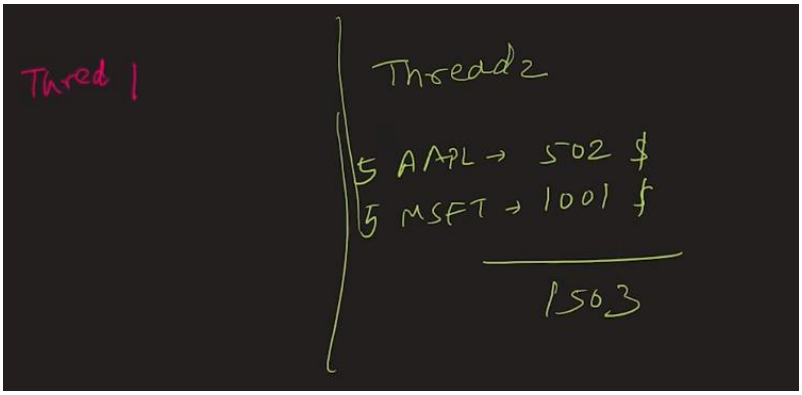


After adding the **config** to the **graph.invoke()**, we can see that the results are correct because the memory is passed



We can even have **multiple thread_ids** for maintaining **multiple memory contexts** to be used when needed





Add memory

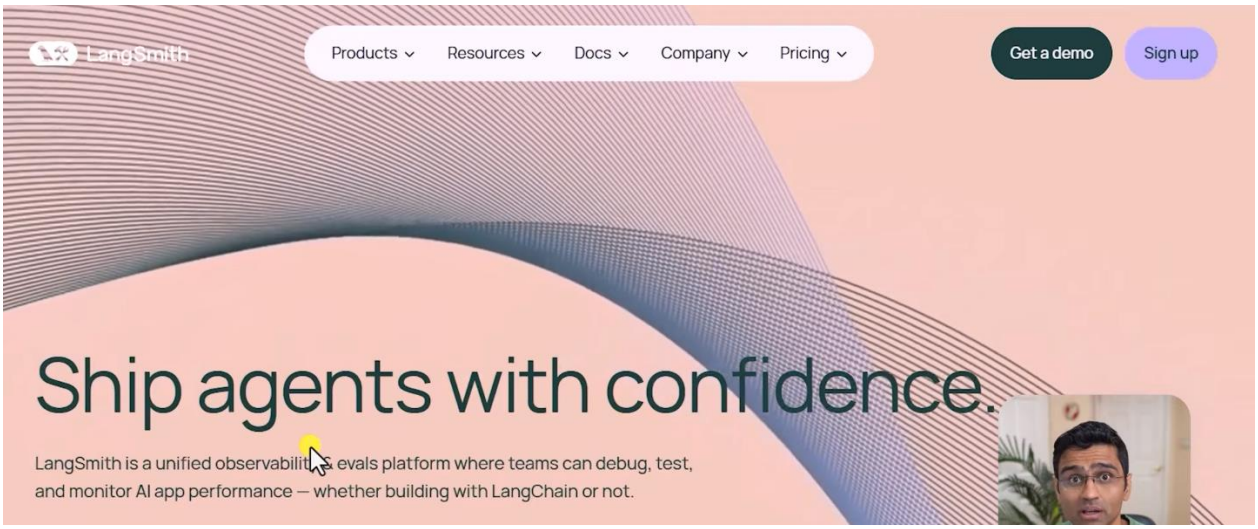
Back to top

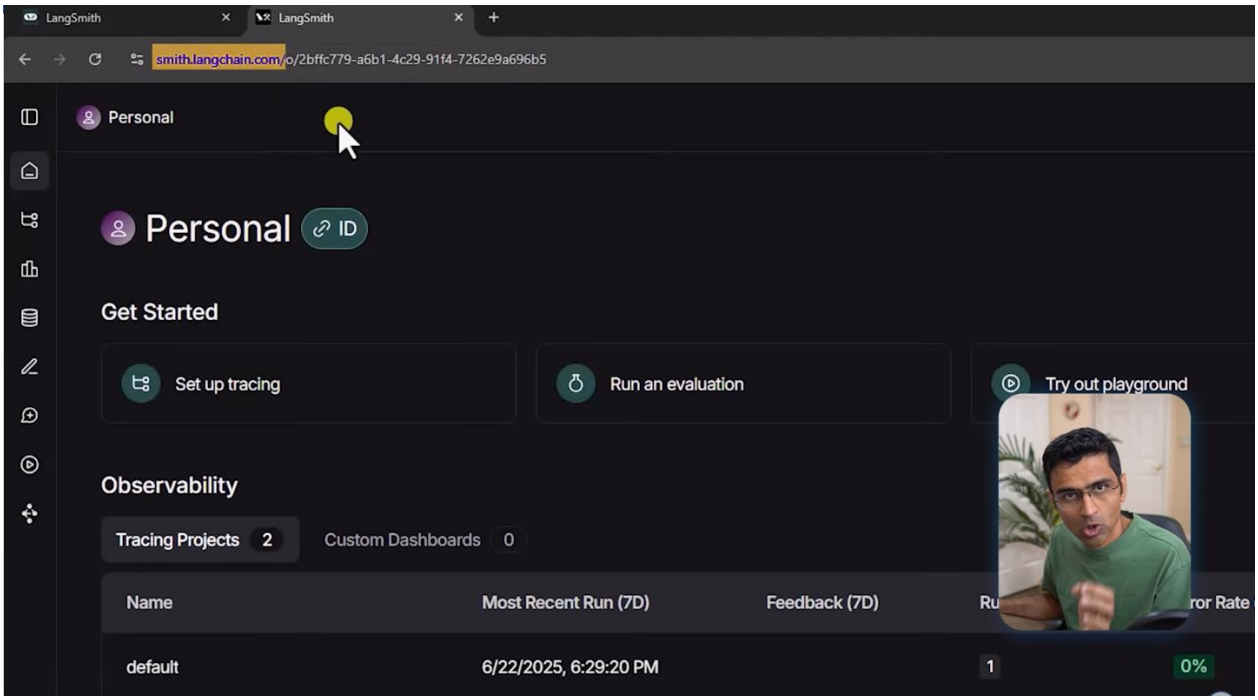
The chatbot can now use tools to answer user questions, but it does not remember the context of previous interactions. This limits its ability to have coherent, multi-turn conversations.

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.

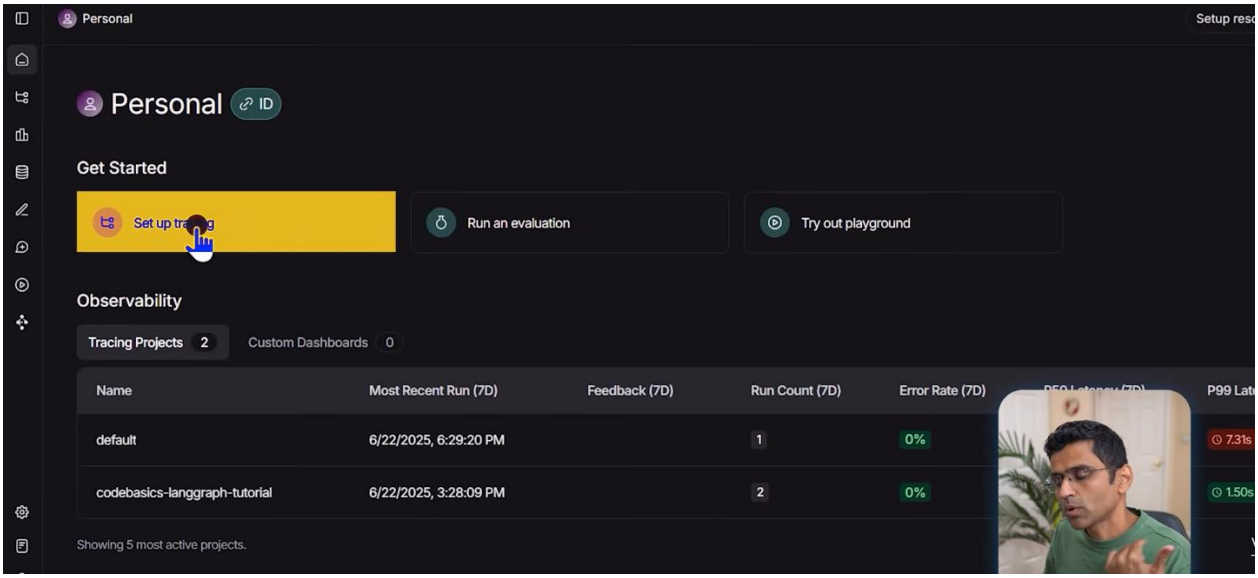
We will see later that **checkpointing** is *much* more powerful than simple chat memory - it lets you sav
resume complex state at any time for error recovery, human-in-the-loop workflows, time travel intera
and more. But first, let's add checkpointing to enable multi-turn conversations.

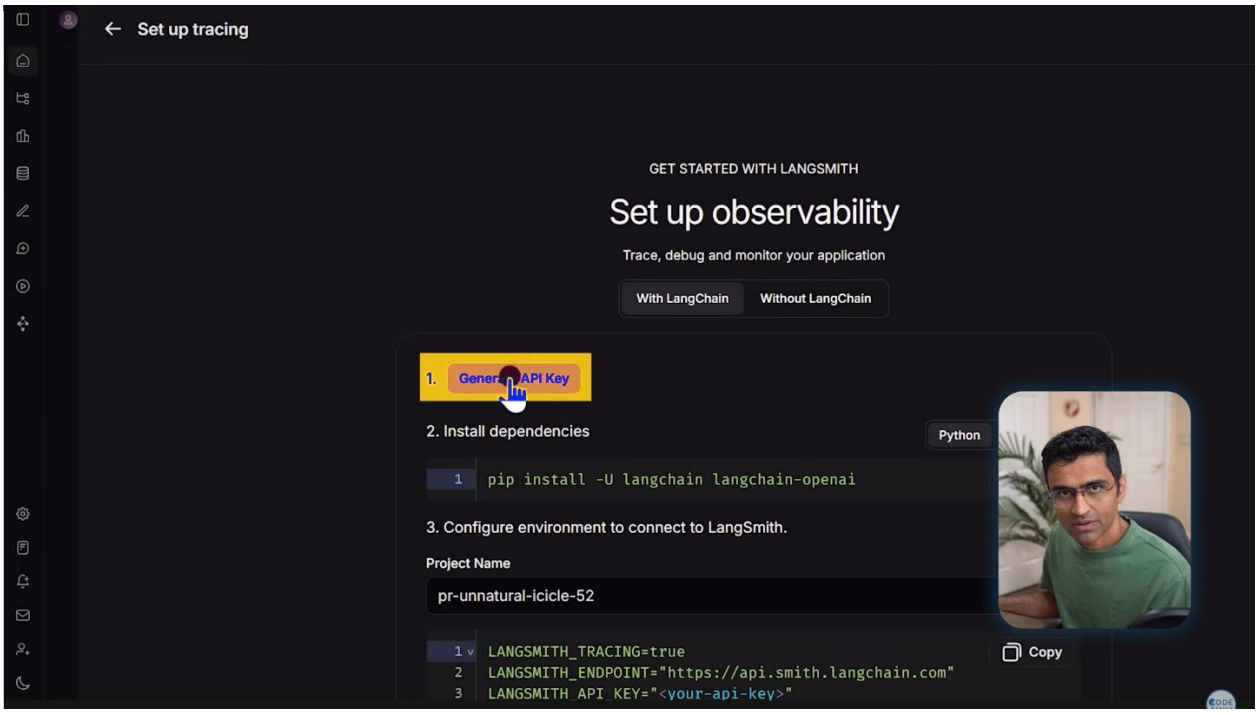
Table of conte
1. Create a Me
checkpointer
2. Compile the
3. Interact witl
4. Ask a follow
5. Inspect the
Next steps

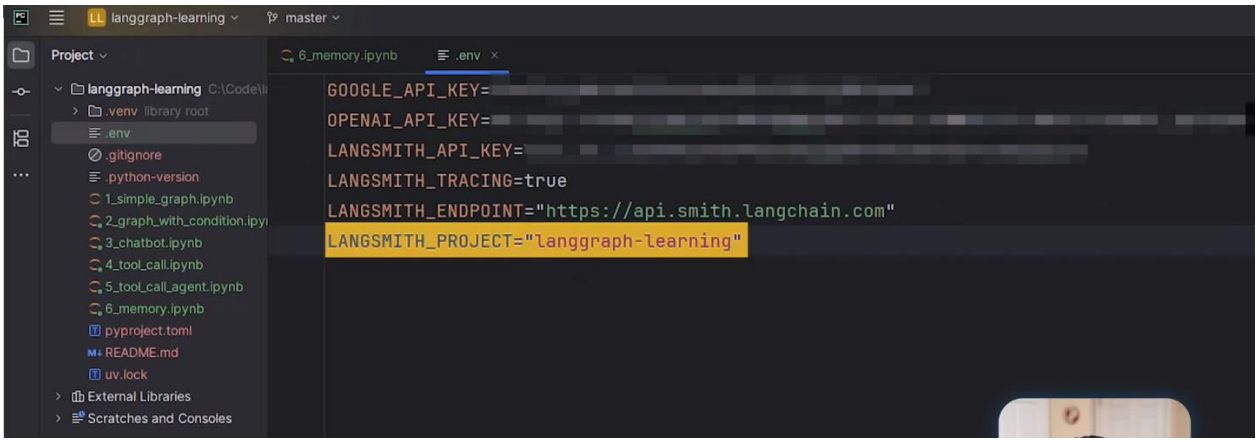


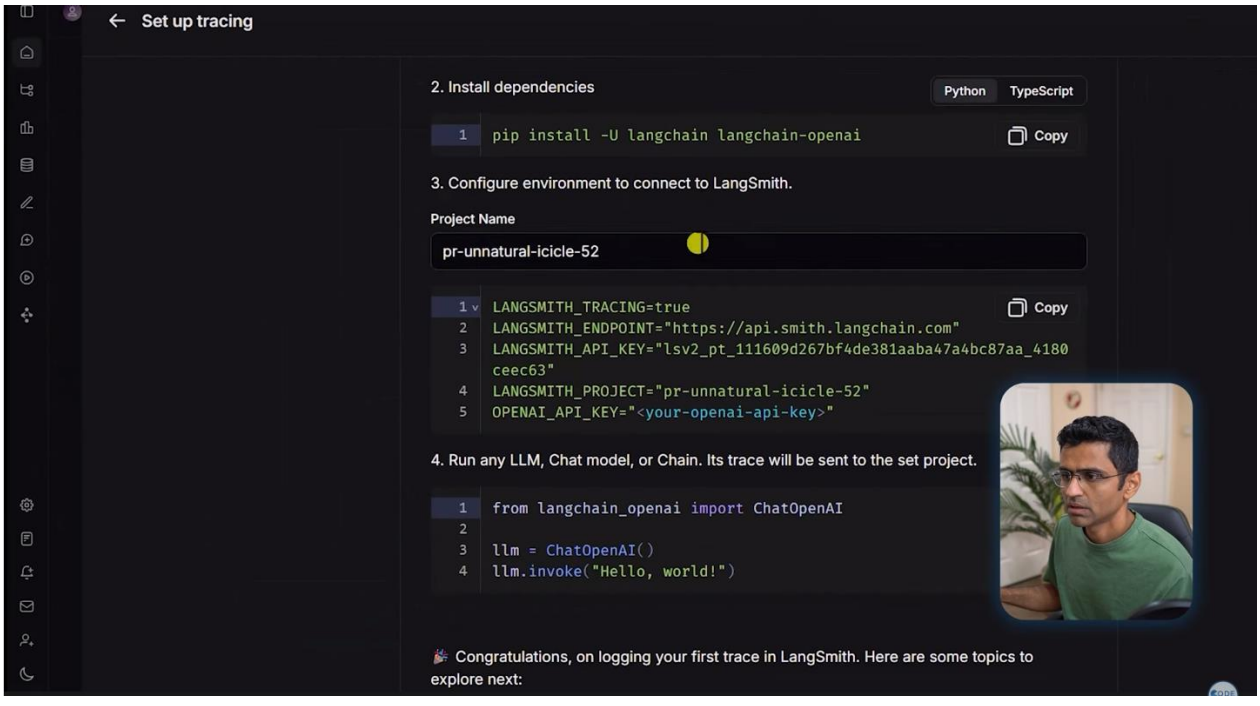


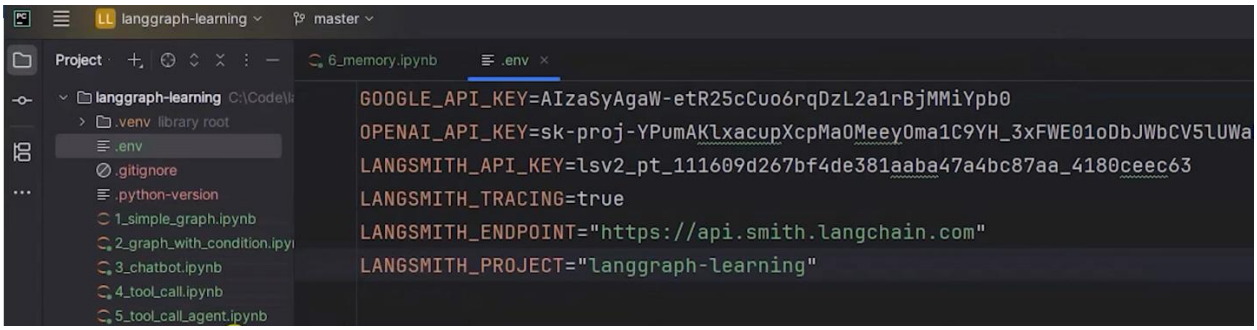
Log in with your Google ID

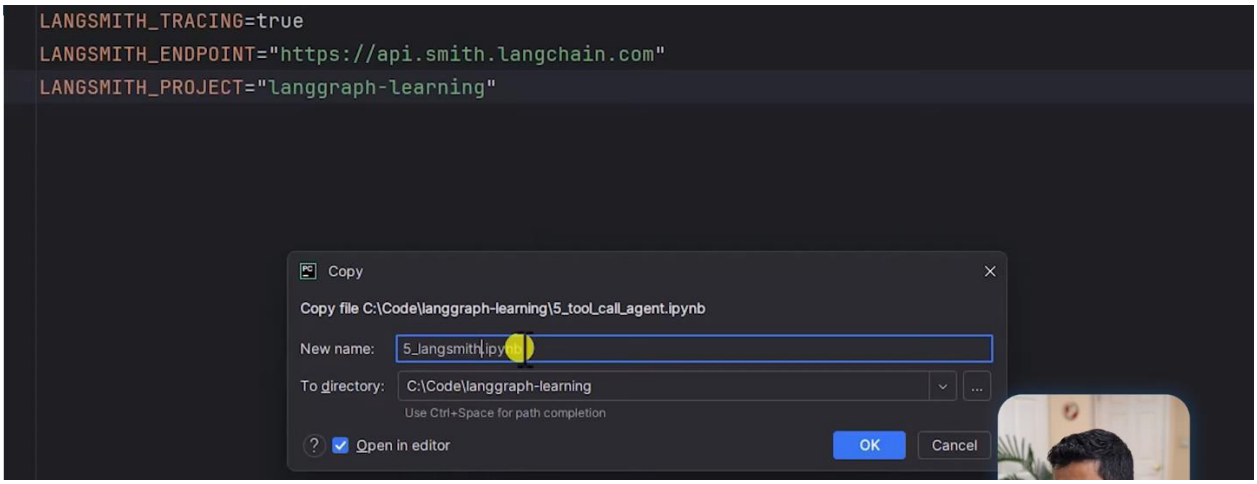


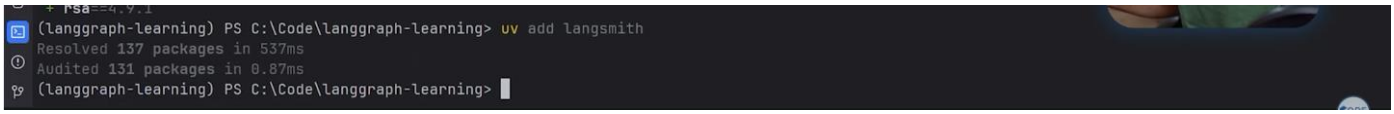

Set up tracing
GET STARTED WITH LANGSMITH
Set up observability
Trace, debug and monitor your application
With LangChain
Without LangChain
1. Generate API Key
2. Install dependencies
Python
pip install -U langchain langchain-openai
3. Configure environment to connect to LangSmith.
Project Name
pr-unnatural-icicle-52
LANGSMITH_TRACING=true
LANGSMITH_ENDPOINT="https://api.smith.langchain.com"
LANGSMITH_API_KEY="<your-api-key>"
Copy

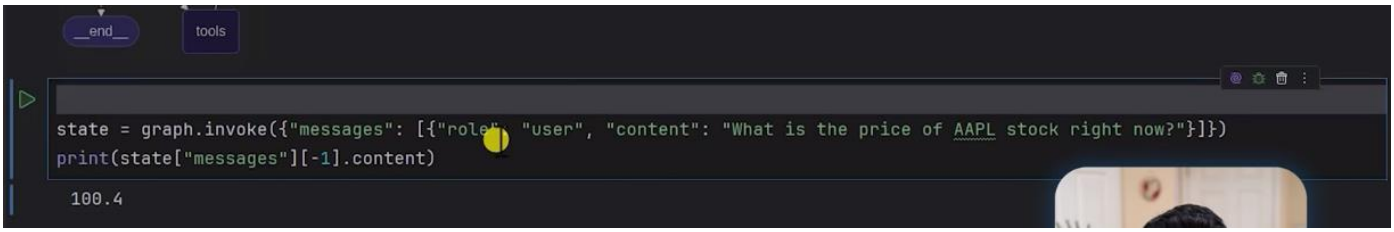

langgraph-learning
master
Project
6_memory.ipynb
.env
langgraph-learning C:\Code\li
.venv library root
.env
.gitignore
.python-version
1_simple_graph.ipynb
2_graph_with_condition.ipy
3_chatbot.ipynb
4_tool_call.ipynb
5_tool_call_agent.ipynb
6_memory.ipynb
pyproject.toml
README.md
uv.lock
External Libraries
Scratches and Consoles
GOOGLE_API_KEY=
OPENAI_API_KEY=
LANGSMITH_API_KEY=
LANGSMITH_TRACING=true
LANGSMITH_ENDPOINT="https://api.smith.langchain.com"
LANGSMITH_PROJECT="langgraph-learning"


Set up tracing
2. Install dependencies
Python
TypeScript
pip install -U langchain langchain-openai
Copy
3. Configure environment to connect to LangSmith.
Project Name
pr-unnatural-icicle-52
LANGSMITH_TRACING=true
LANGSMITH_ENDPOINT="https://api.smith.langchain.com"
LANGSMITH_API_KEY="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"
LANGSMITH_PROJECT="pr-unnatural-icicle-52"
OPENAI_API_KEY="<your-openai-api-key>"
4. Run any LLM, Chat model, or Chain. Its trace will be sent to the set project.
from langchain_openai import ChatOpenAI
llm = ChatOpenAI()
llm.invoke("Hello, world!")
Congratulations, on logging your first trace in LangSmith. Here are some topics to explore next:

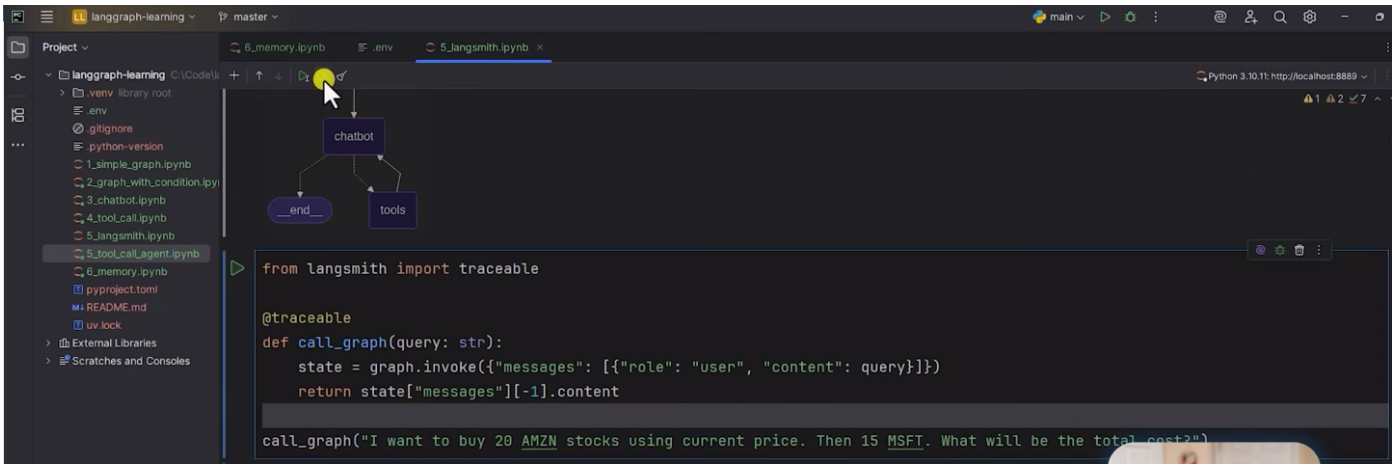


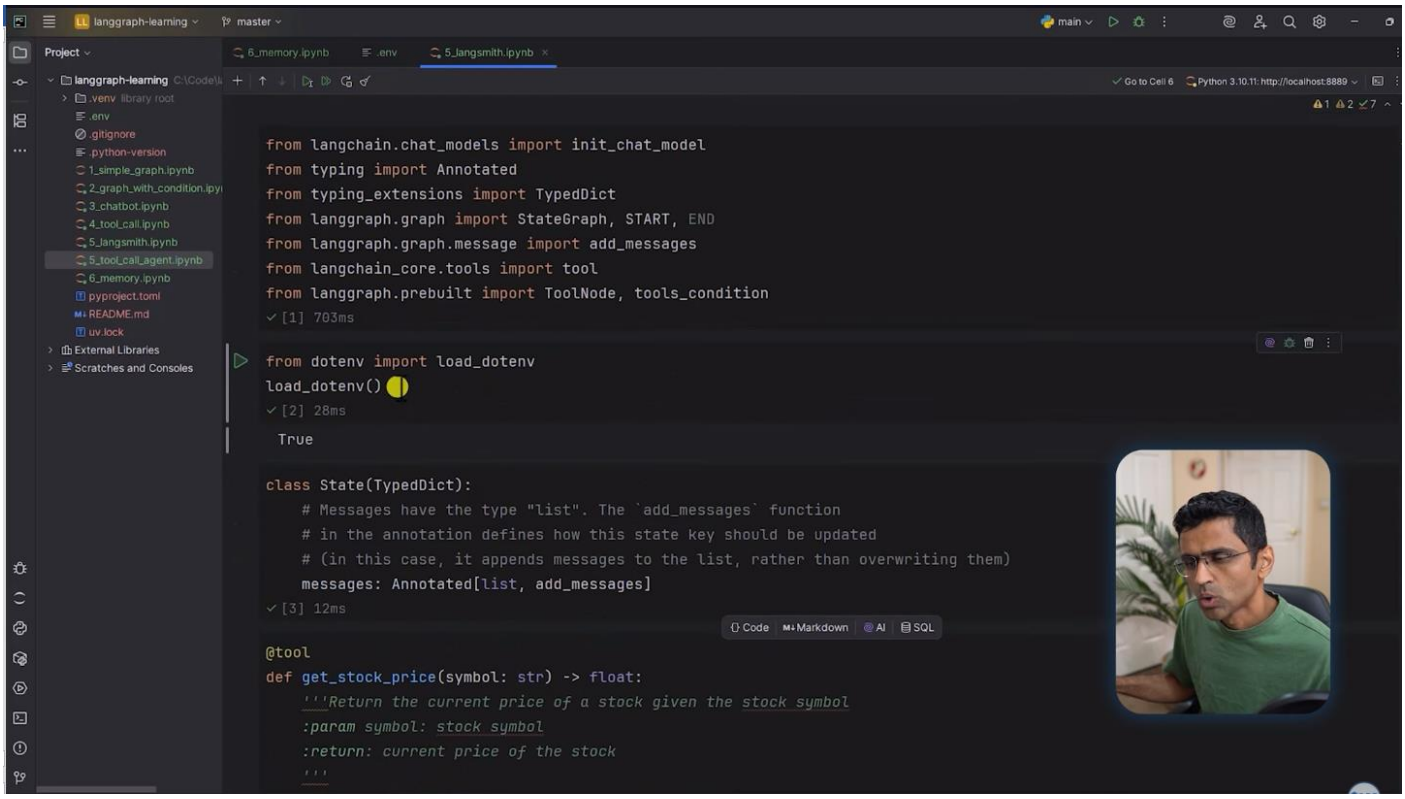




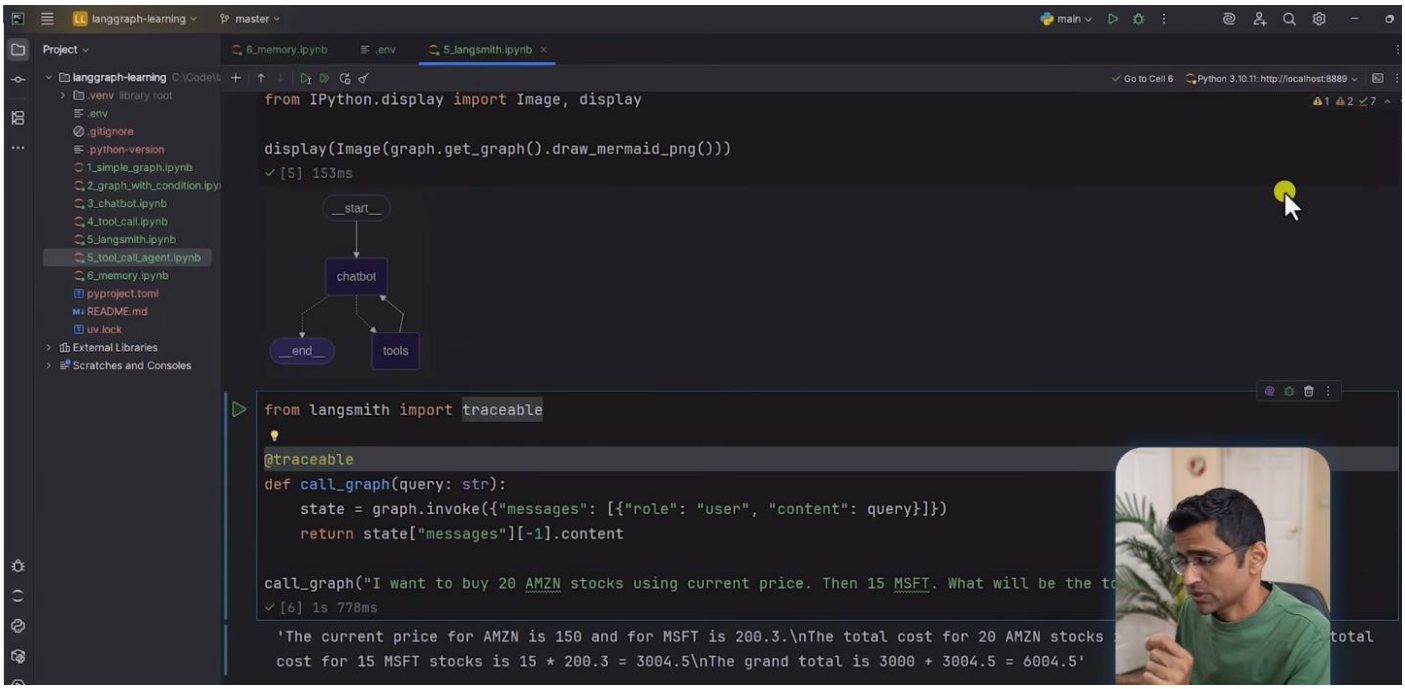


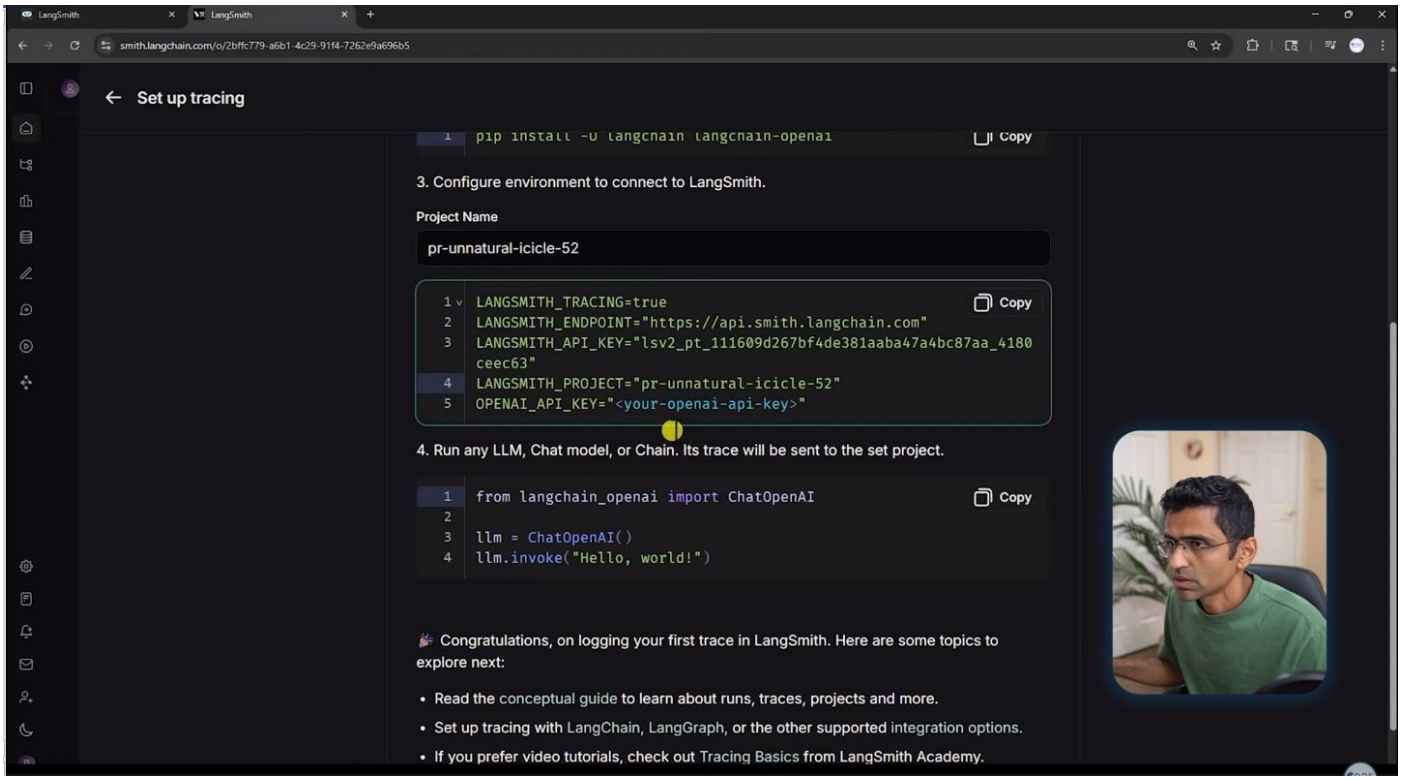
We can now add annotations for Langsmith to pick up traces of our calls

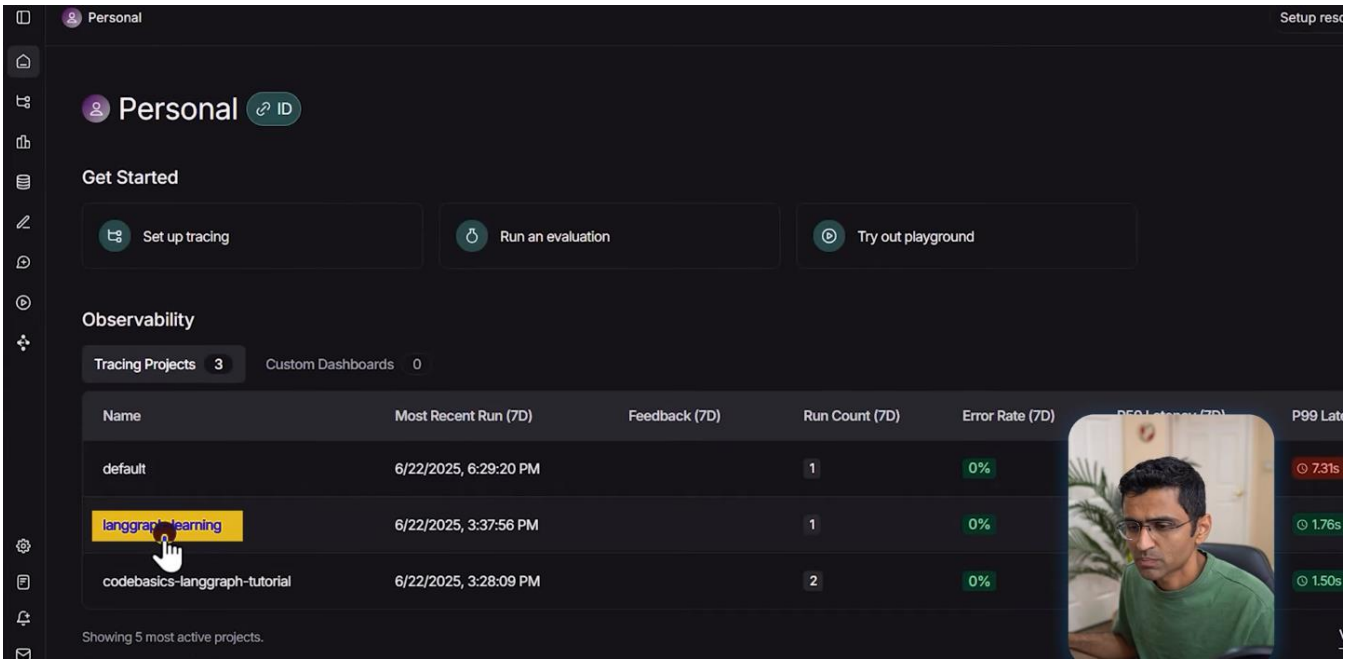


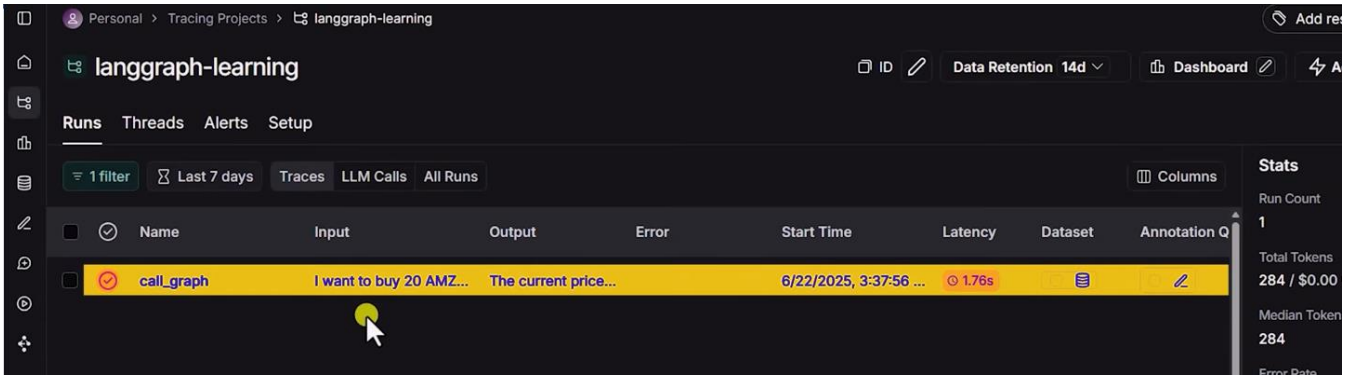


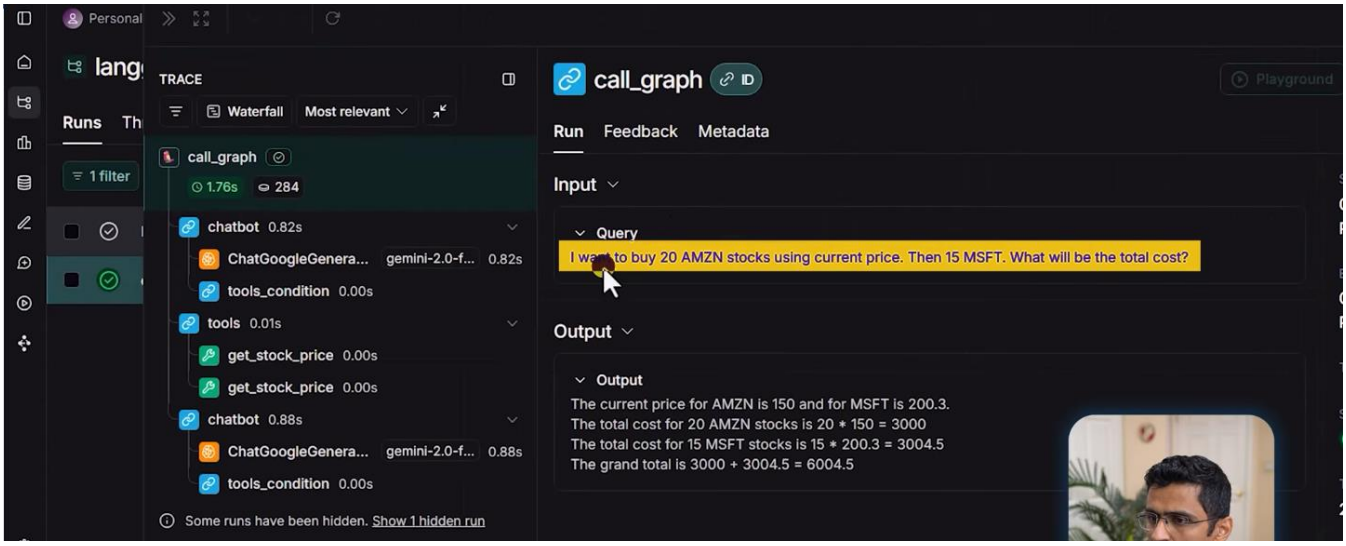
Clicking Run All will add all the Langsmith libraries for us

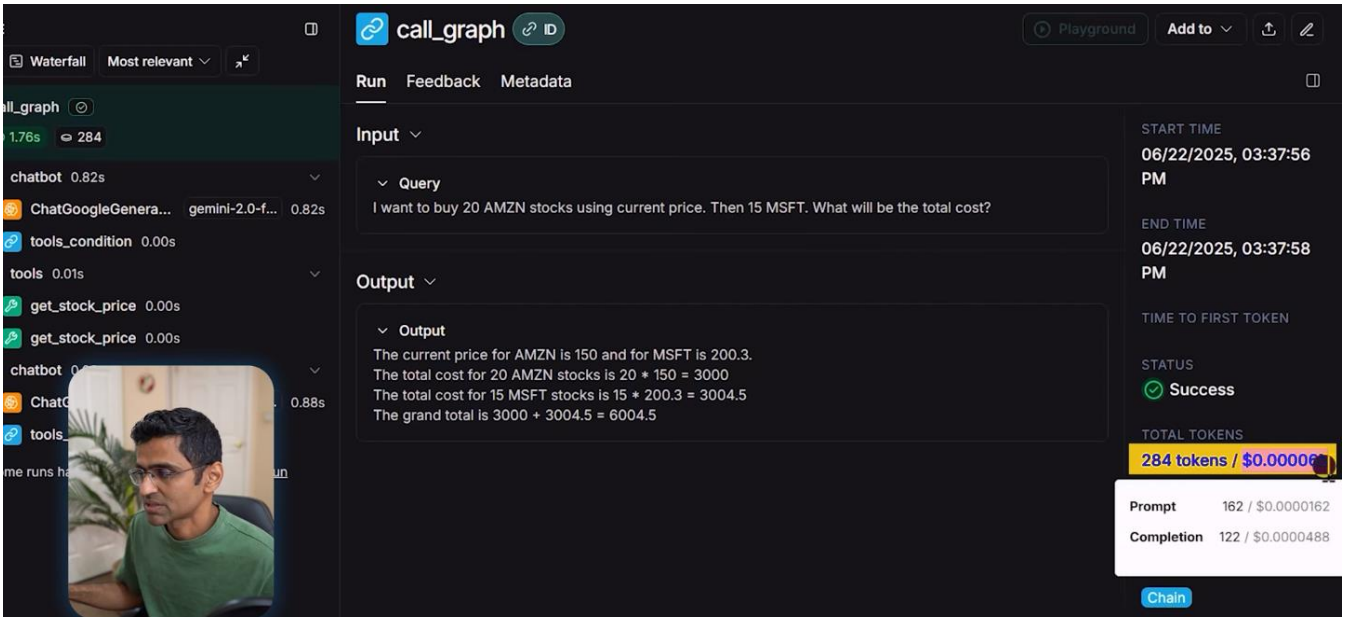


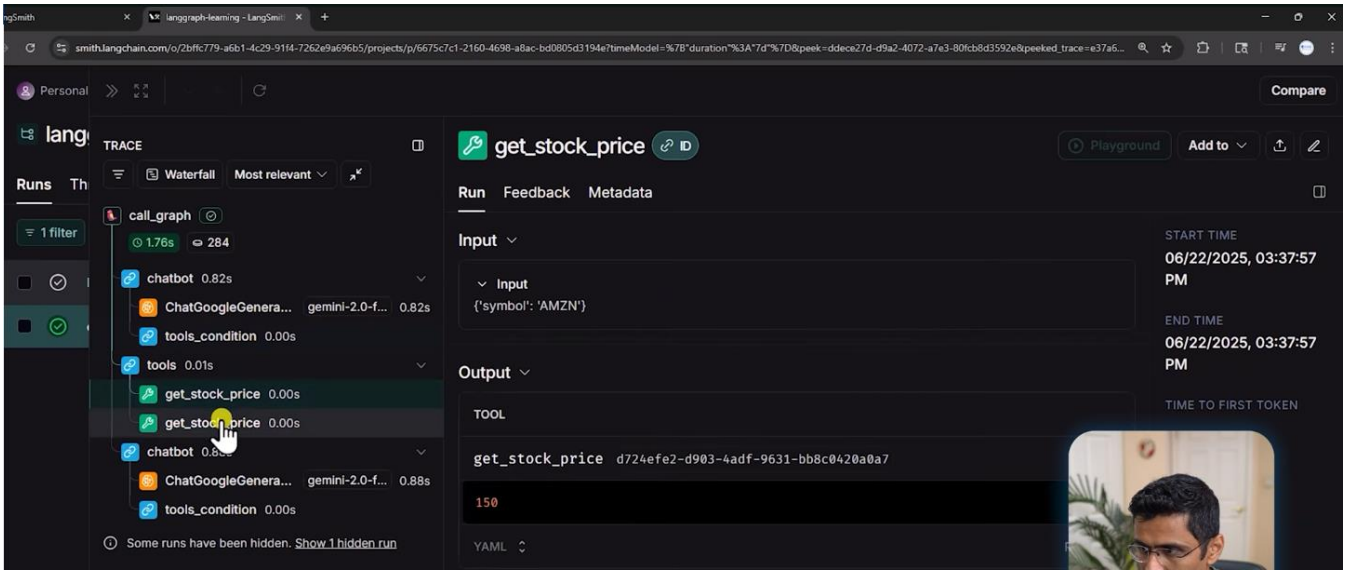

Set up tracing
pip install -U langchain langchain-openai
Copy
3. Configure environment to connect to LangSmith.
Project Name
pr-unnatural-icicle-52
LANGSMITH_TRACING=true
LANGSMITH_ENDPOINT="https://api.smith.langchain.com"
LANGSMITH_API_KEY="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"
LANGSMITH_PROJECT="pr-unnatural-icicle-52"
OPENAI_API_KEY="<your-openai-api-key>"
4. Run any LLM, Chat model, or Chain. Its trace will be sent to the set project.
from langchain_openai import ChatOpenAI
llm = ChatOpenAI()
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.

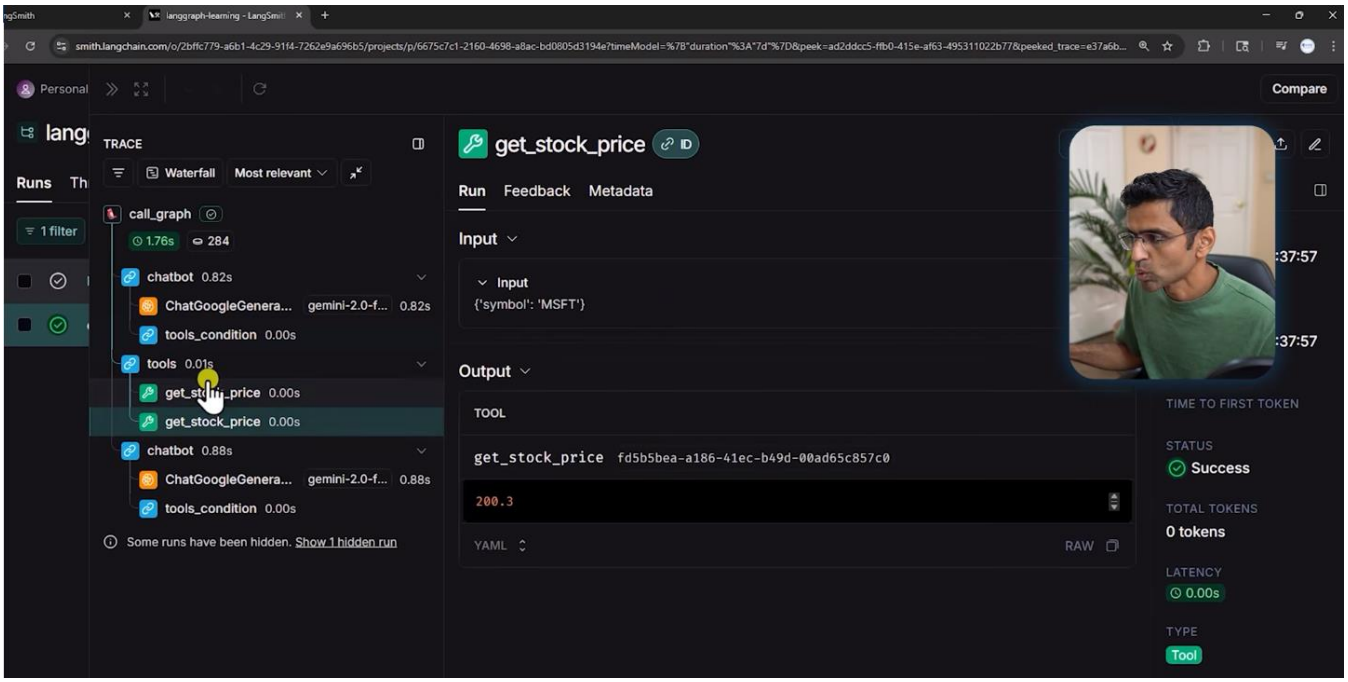

Personal
Personal
ID
Get Started
Set up tracing
Run an evaluation
Try out playground
Observability
Tracing Projects 3
Custom Dashboards 0
Name
Most Recent Run (7D)
Feedback (7D)
Run Count (7D)
Error Rate (7D)
default
6/22/2025, 6:29:20 PM
1
0%
langgraph-learning
6/22/2025, 3:37:56 PM
1
0%
codebasics-langgraph-tutorial
6/22/2025, 3:28:09 PM
2
0%
Showing 5 most active projects.

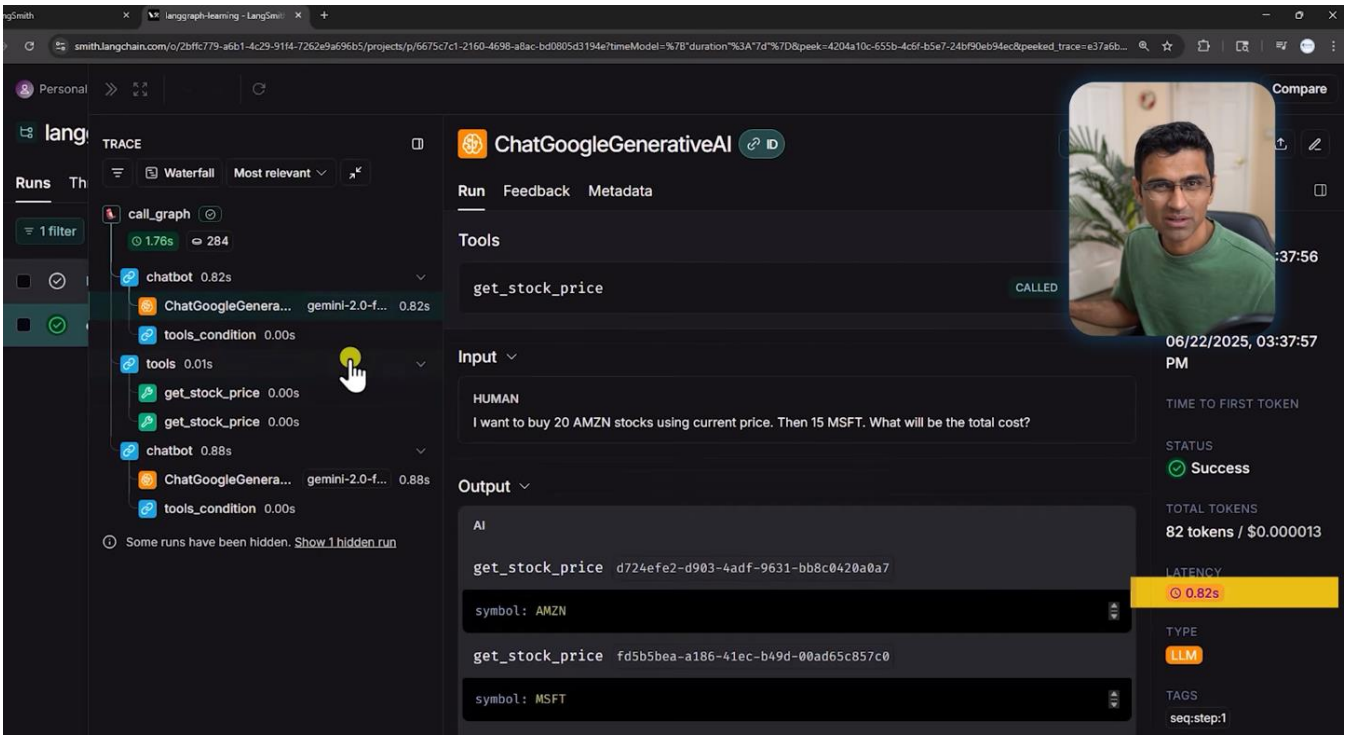

Personal
Tracing Projects
langgraph-learning
langgraph-learning
ID
Data Retention 14d
Dashboard
Runs
Threads
Alerts
Setup
1 filter
Last 7 days
Traces
LLM Calls
All Runs
Columns
Name
Input
Output
Error
Start Time
Latency
Dataset
Annotation Q
call_graph
I want to buy 20 AMZ...
The current price...
6/22/2025, 3:37:56 ...
1.76s
Stats
Run Count
1
Total Tokens
284 / $0.00
Median Token
284


Personal
TRACE
Waterfall
Most relevant
call_graph
1.76s
284
chatbot 0.82s
ChatGoogleGenera... gemini-2.0-f... 0.82s
tools_condition 0.00s
tools 0.01s
get_stock_price 0.00s
get_stock_price 0.00s
chatbot 0.88s
ChatGoogleGenera... gemini-2.0-f... 0.88s
tools_condition 0.00s
Some runs have been hidden. Show 1 hidden run
call_graph
ID
Playground
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

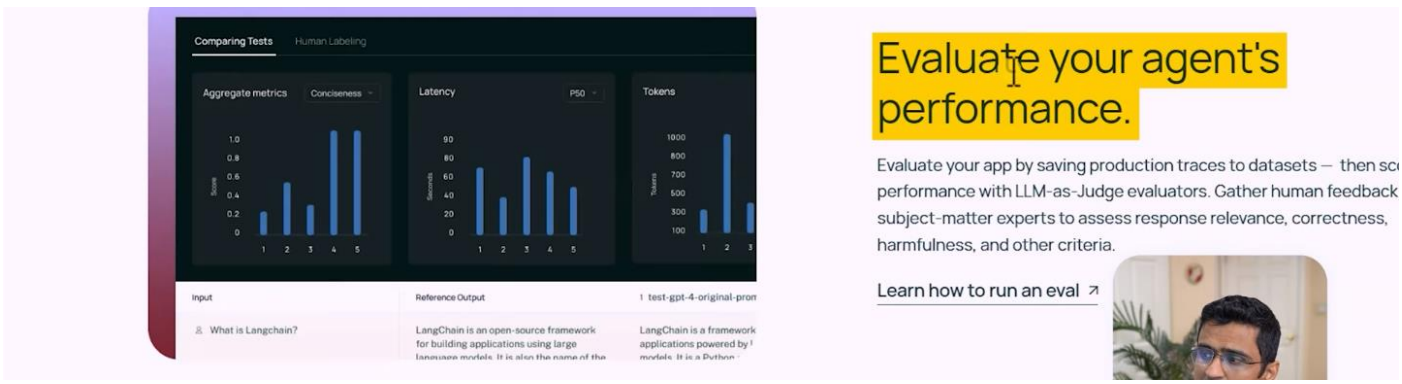

Waterfall
Most relevant
call_graph
1.76s
284
chatbot 0.82s
ChatGoogleGenera... gemini-2.0-f... 0.82s
tools_condition 0.00s
tools 0.01s
get_stock_price 0.00s
get_stock_price 0.00s
0.88s
call_graph
ID
Playground
Add to
Run
Feedback
Metadata
Input
Query
I want to buy 20 AMZN stocks using current price. 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

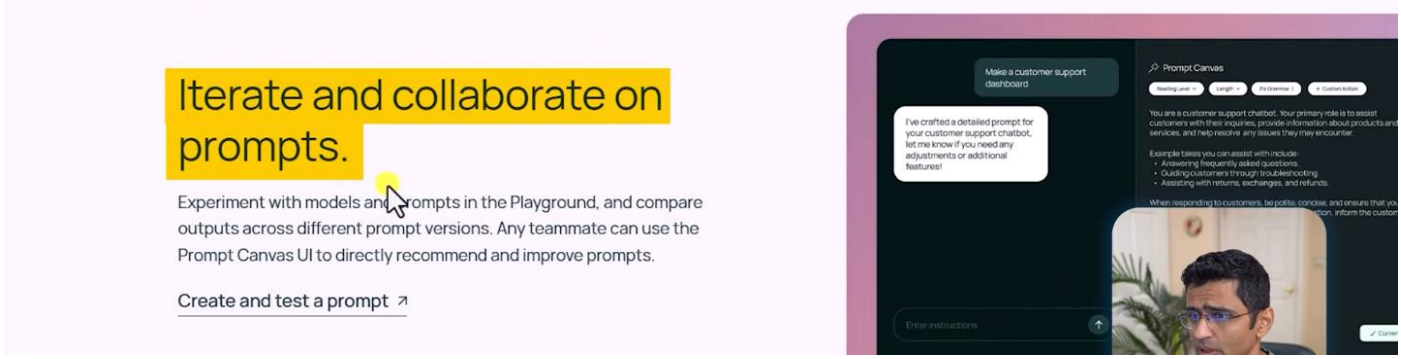

smith.langchain.com/o/2bffc779-a6b1-4c29-91f4-7262e9a696b5/projects/p/6675c7c1-2160-4698-a8ac-bd0805d3194e?timeModel=%7B"duration"%3A"7d"%7D&peek=ddece27d-d9a2-4072-a7e3-80fcb8d3592e&peeked_trace=e37a6...
Personal
Compare
TRACE
Waterfall
Most relevant
call_graph
1.76s
284
chatbot 0.82s
ChatGoogleGenera... gemini-2.0-f... 0.82s
tools_condition 0.00s
tools 0.01s
get_stock_price 0.00s
get_stock_price 0.00s
chatbot 0.88s
ChatGoogleGenera... gemini-2.0-f... 0.88s
tools_condition 0.00s
Some runs have been hidden. Show 1 hidden run
get_stock_price
ID
Playground
Add to
Run
Feedback
Metadata
Input
Input
{'symbol': 'AMZN'}
Output
TOOL
get_stock_price d724efe2-d903-4adf-9631-bb8c0420a0a7
150
YAML
START TIME
06/22/2025, 03:37:57 PM
END TIME
06/22/2025, 03:37:57 PM
TIME TO FIRST TOKEN

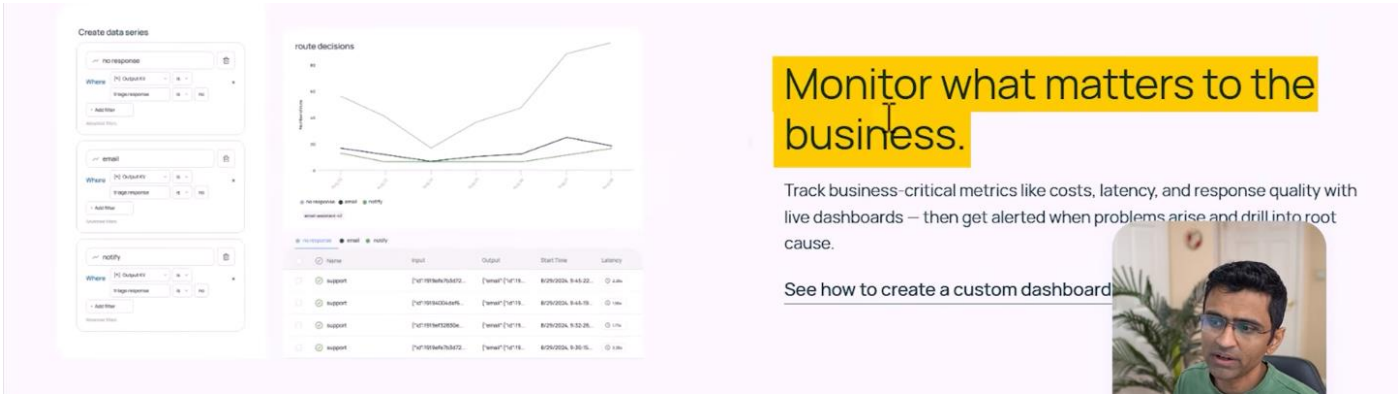




We can now monitor cost and latency, output checks can also be performed easily

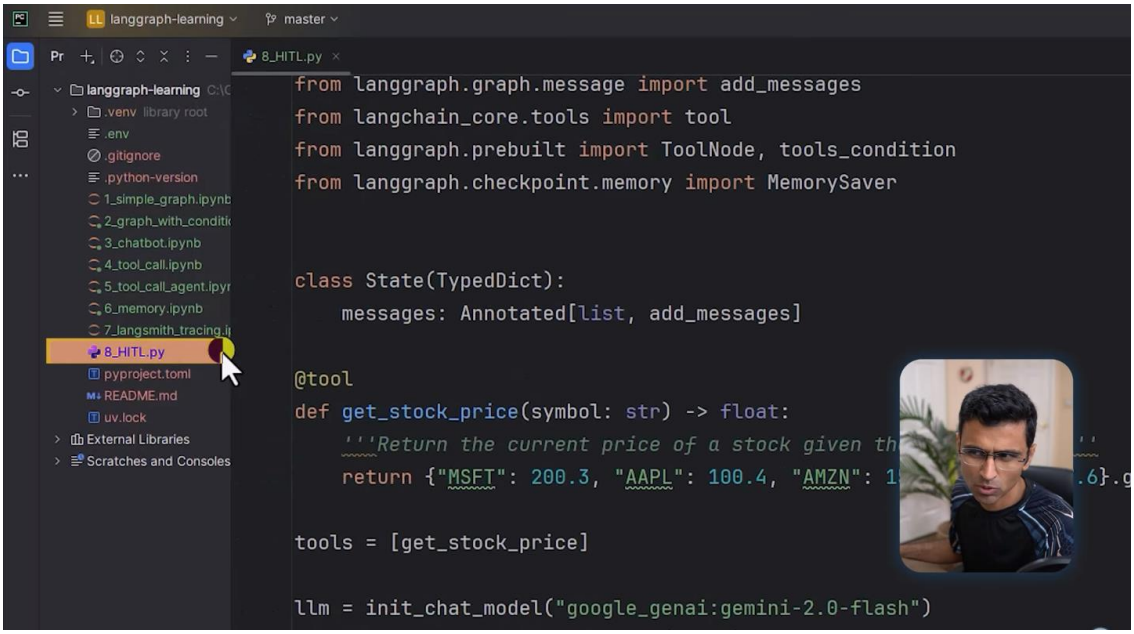


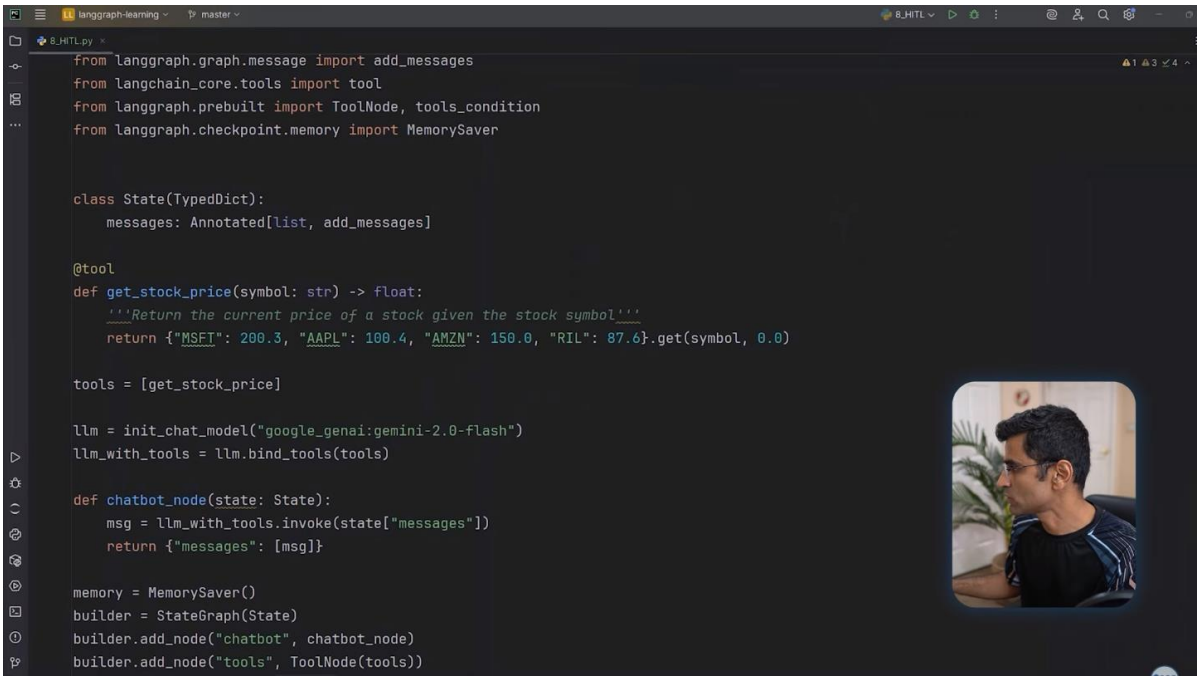




Human in the loop - HITL

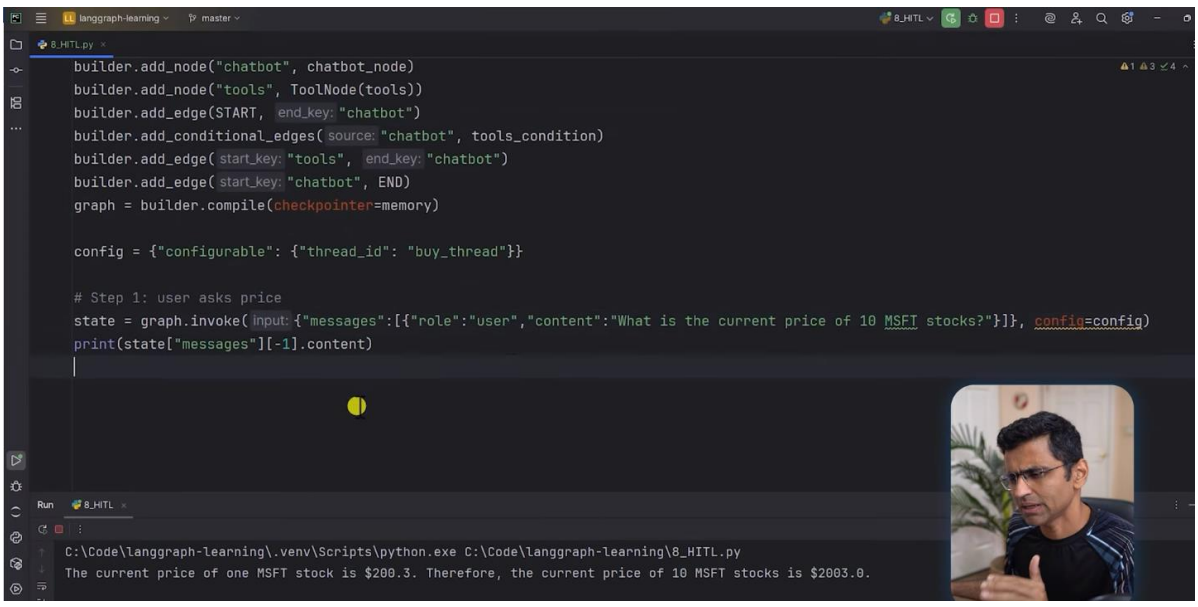
You might want human approvals within some agentic workflows like below





```python
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))
```

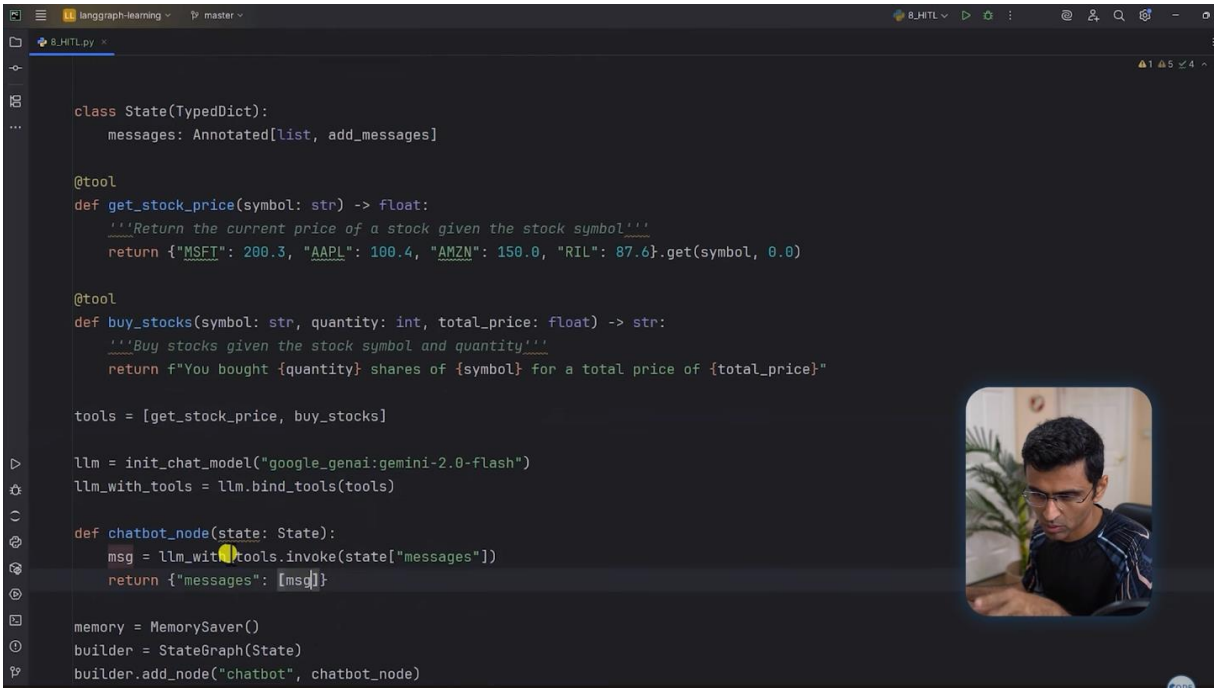
```python
def chatbot_node(state: State):
    msg = llm_with_tools.invoke(state["messages"])
    return {"messages": [msg]}

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 sto
print(state["messages"][-1].content)
```



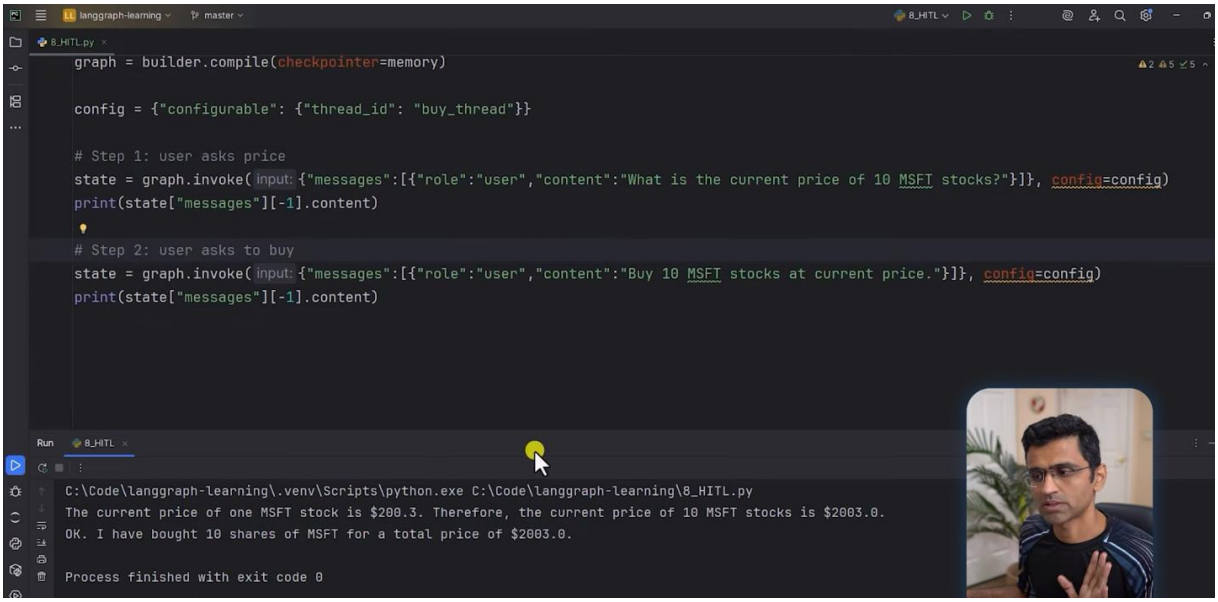
```python
builder.add_node("chatbot", chatbot_node)
builder.add_node("tools", ToolNode(tools))
builder.add_edge(START, end_key="chatbot")
builder.add_conditional_edges(source="chatbot", tools_condition)
builder.add_edge(start_key="tools", end_key="chatbot")
builder.add_edge(start_key="chatbot", END)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

```
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.
```

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



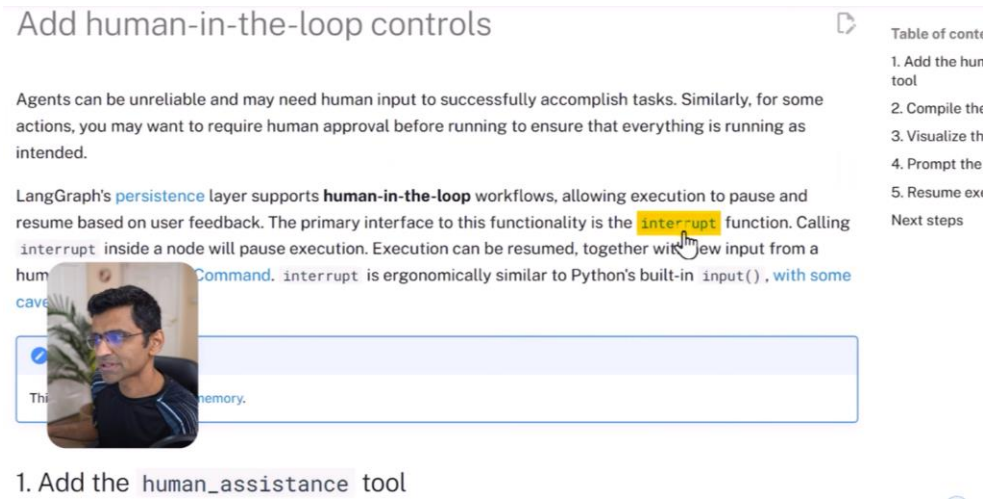
```python
class State(TypedDict):
    messages: Annotated[list, add_messages]

@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)
```



```python
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)
```

```
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.

Process finished with exit code 0
```

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 hum Command. `interrupt` is ergonomically similar to Python's built-in `input()`, with some cave

Table of cont
1. Add the hun tool
2. Compile the
3. Visualize th
4. Prompt the
5. Resume exe
Next steps

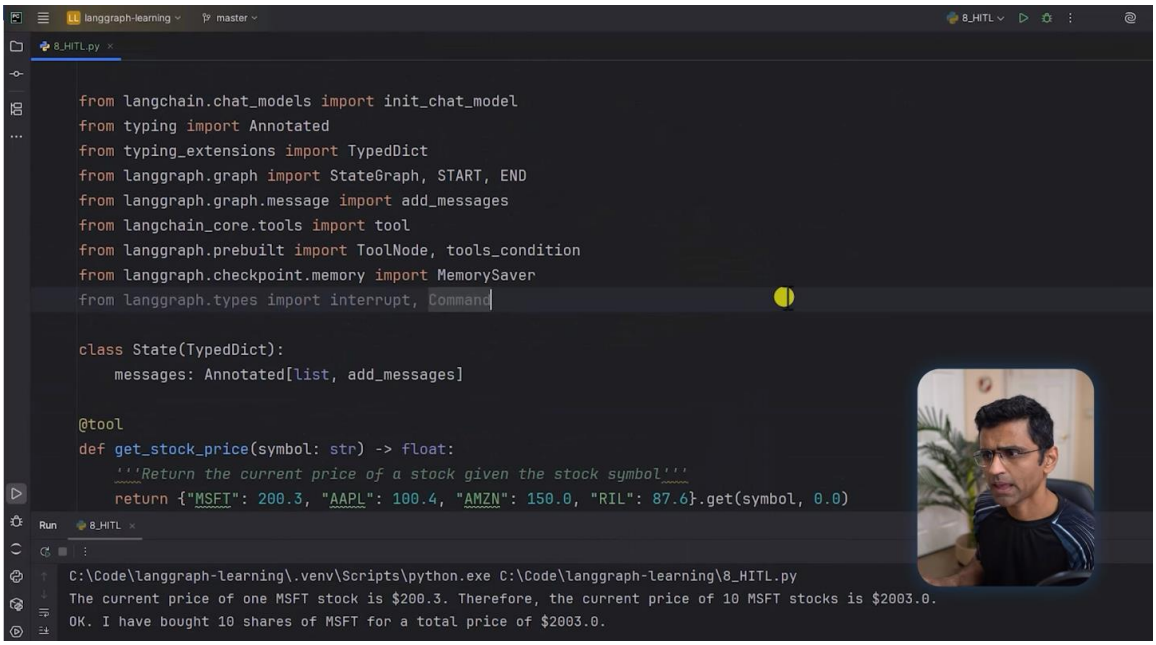
1. Add the `human_assistance` tool

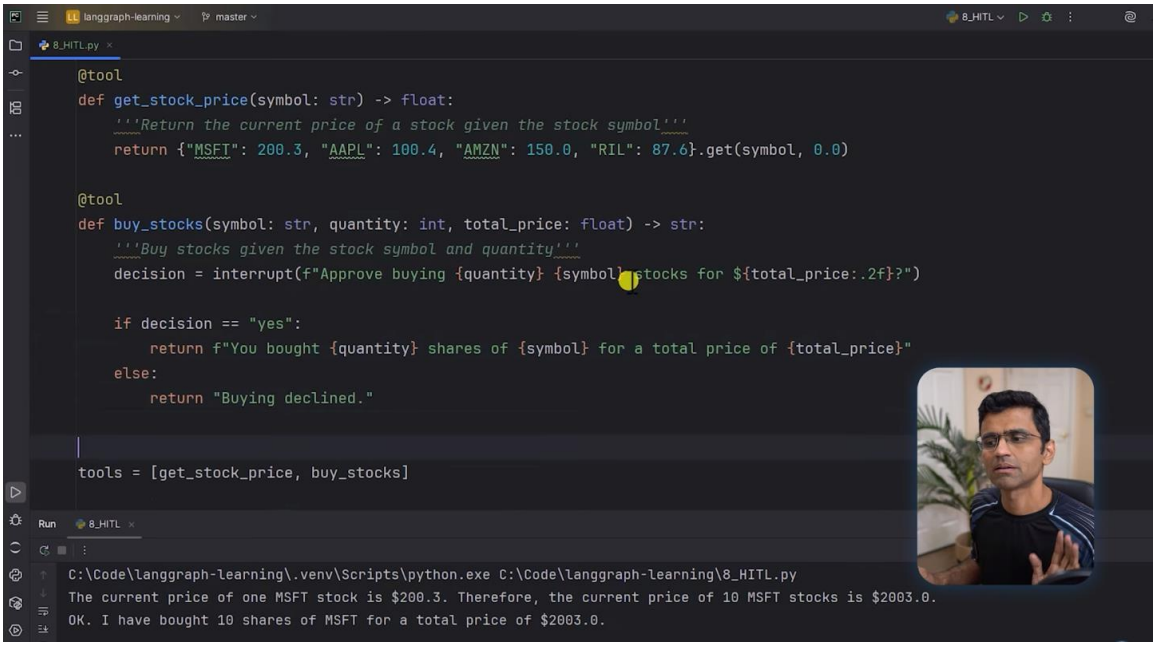
like approvals, edits, or gathering additional context.

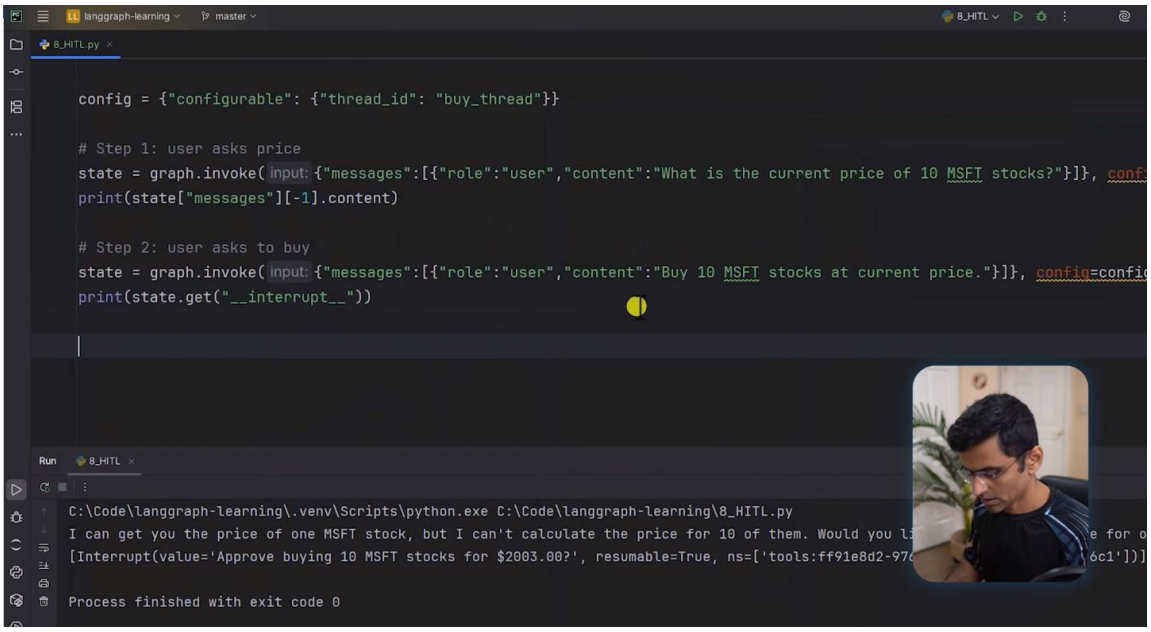
The graph is resumed using a `Command` object that provides the human's response.

API Reference: interrupt | Command

```python
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)
```

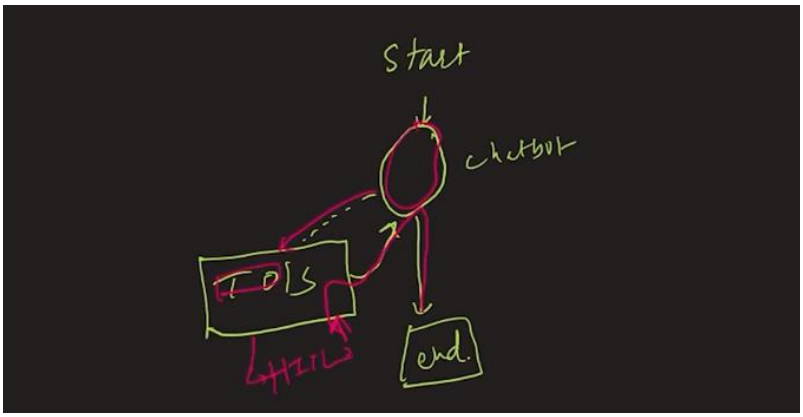


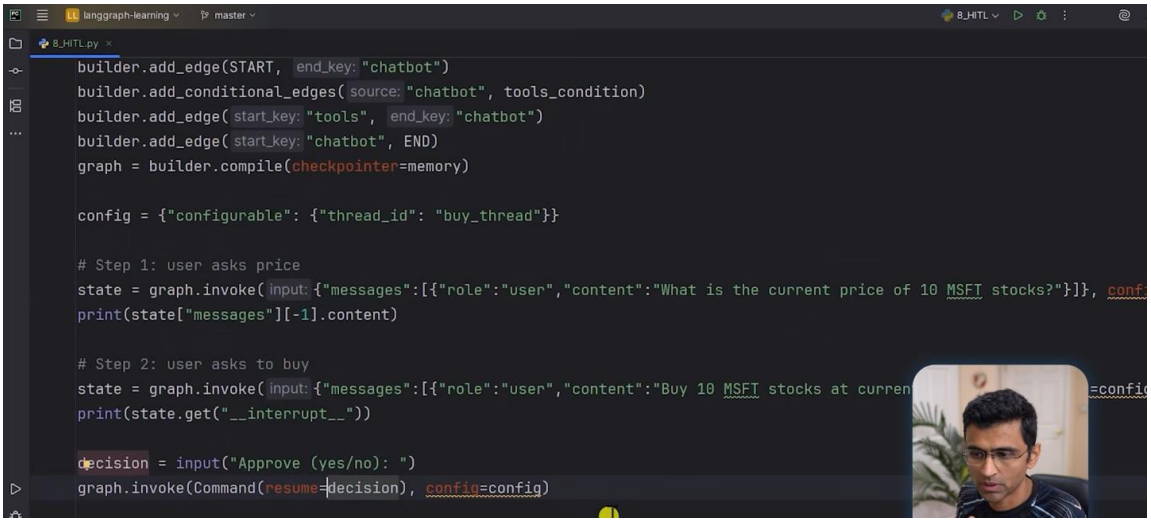




This tells us that the graph execution is correctly interrupted as intended

Human in the loop - HITL



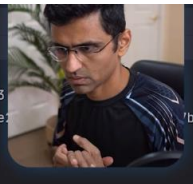


Command will take the HIL reply and feed it back as the decision output value in interrupt as below

```python
@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:.2f}?")

    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"])
```

```python
# Step 2: user asks to buy
state = graph.invoke( input: {"messages":[{"role":"user","content":"Buy 10 MSFT stocks at current price."}]}, config=confi
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
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
[Interrupt(value='Approve buying 10 MSFT stocks for $2003.00?', resumable=True, ns=['tools:def9b365-26
Approve (yes/no): yes
OK. You bought 10 shares of MSFT for a total price of $2003.0.

Process finfinished with exit code 0
```



```
Run  8_HITL
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
[Interrupt(value='Approve buying 10 MSFT stocks for $2003.00?', resumable=True, ns=['tools:e5342d26-6e
Approve (yes/no): no
I was not able to buy the stocks. Buying declined.
```