LangGraph Crash Course #5 - Drawbacks of React Agents

Harish Neel | AI
7.67K subscribers

Subscribe

137 Share Clip Save ...

6,150 views Feb 12, 2025 #python #artificialintelligence #langgraph

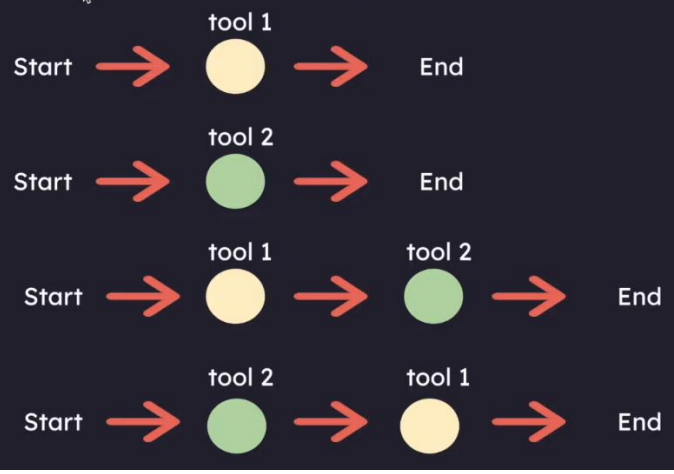
In this beginner-friendly playlist, I'll show you how to use LangGraph to build powerful AI-agents from scratch

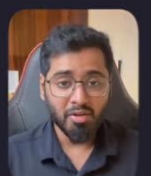
Let's Jump Into The Code

1. Build a Re-act Agent Using LangChain
2. What are its drawbacks and where does LangGraph come into the picture?

ReAct Agents

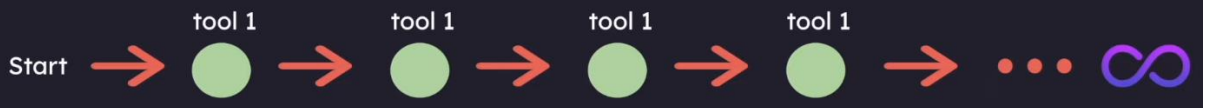
ReAct Agents Are Flexible. i.e., Any state is possible





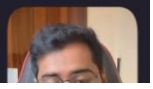
ReAct Agents

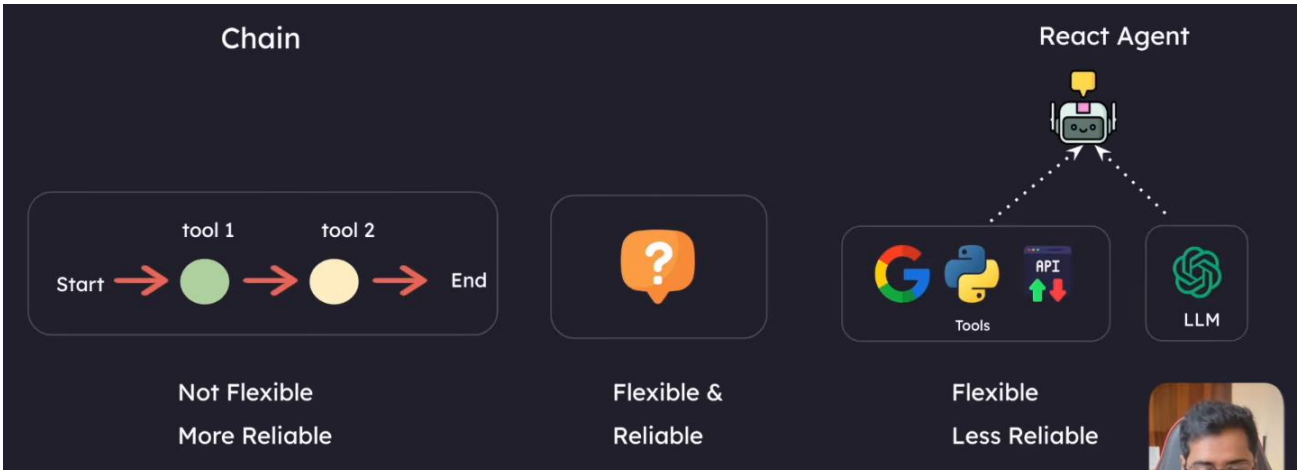
But high flexibility can also mean less reliability

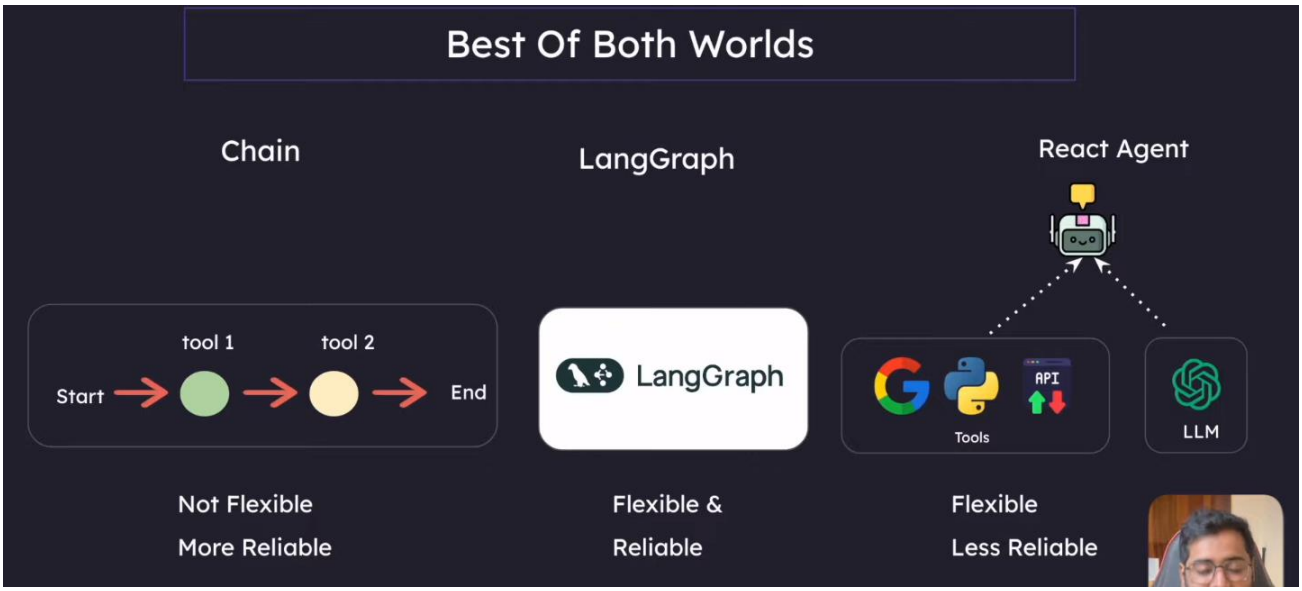


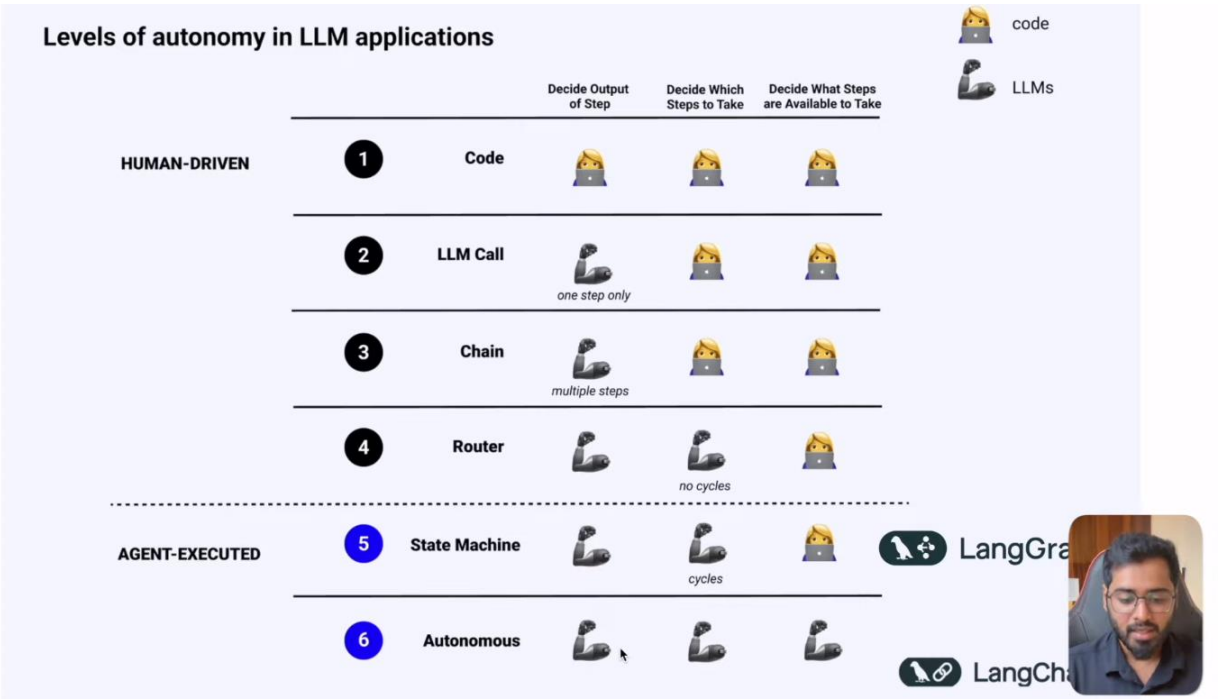
Infinite Loop causes:

1. We did not define the tools correctly
2. The LLM is not capable enough
3. The prompting doesn't define a clear end condition




Chain
React Agent
tool 1
tool 2
Start
End
API
Tools
LLM
Not Flexible
More Reliable
Flexible &
Reliable
Flexible
Less Reliable

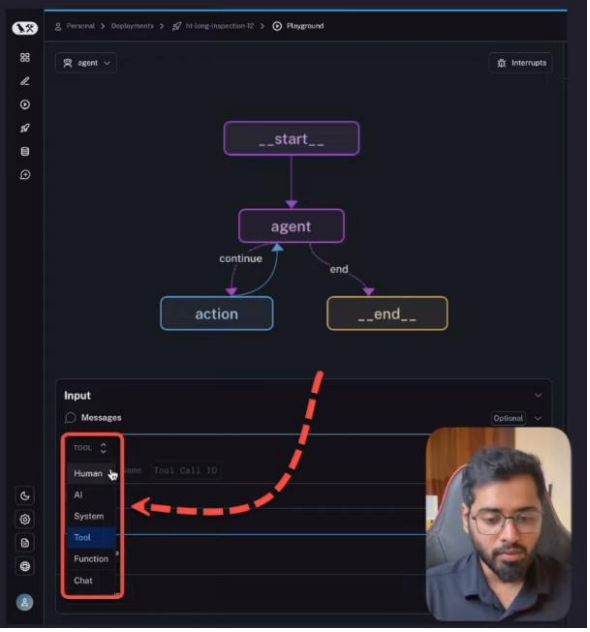

Best Of Both Worlds
Chain
LangGraph
React Agent
tool 1
tool 2
Start
End
LangGraph
API
Tools
LLM
Not Flexible
More Reliable
Flexible &
Reliable
Flexible
Less Reliable


Levels of autonomy in LLM applications
code
LLMs
Decide Output of Step
Decide Which Steps to Take
Decide What Steps are Available to Take
HUMAN-DRIVEN
1
Code
2
LLM Call
one step only
3
Chain
multiple steps
4
Router
no cycles
AGENT-EXECUTED
5
State Machine
cycles
6
Autonomous
LangGra
LangCha

What is LangGraph?

A framework for building controllable, persistent agent workflows with built-in support for human interaction, streaming, and state management.

It uses the Graph Data Structure to achieve this



Key Features Of LangGraph

1. Looping and Branching Capabilities:

Supports conditional statements and loop structures, allowing dynamic execution paths based on state.

2. State Persistence:

Automatically saves and manages state, supporting pause and resume for long-running conversations.

3. Human-Machine Interaction Support:

Allows inserting human review during execution, supporting state editing and modification with flexible interaction control mechanisms.

Key Features Of LangGraph

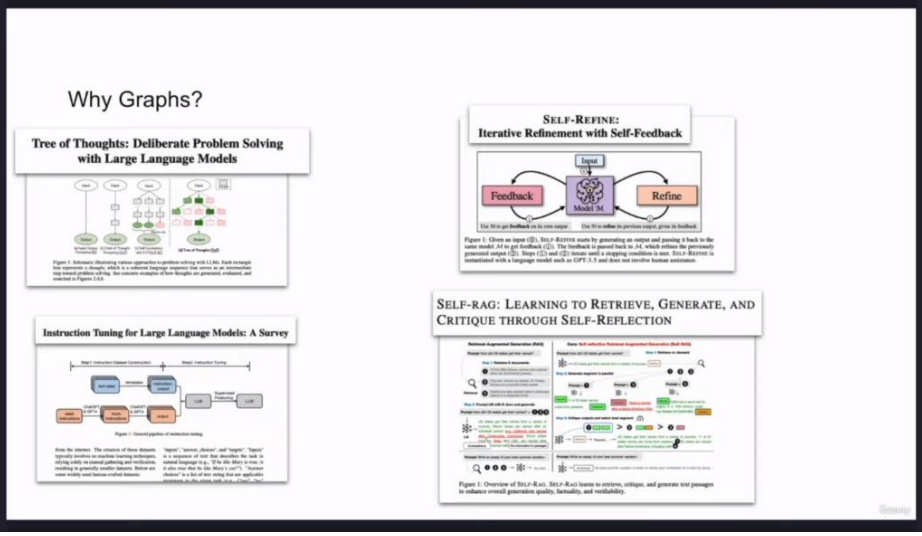
4. Streaming Processing:

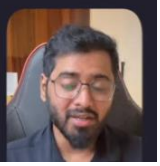
Supports streaming output and real-time feedback on execution status to enhance user experience.

5. Seamless Integration with LangChain:

Reuses existing LangChain components, supports LCEL expressions, and offers rich tool and model support.

Why use the Graph Data Structure?

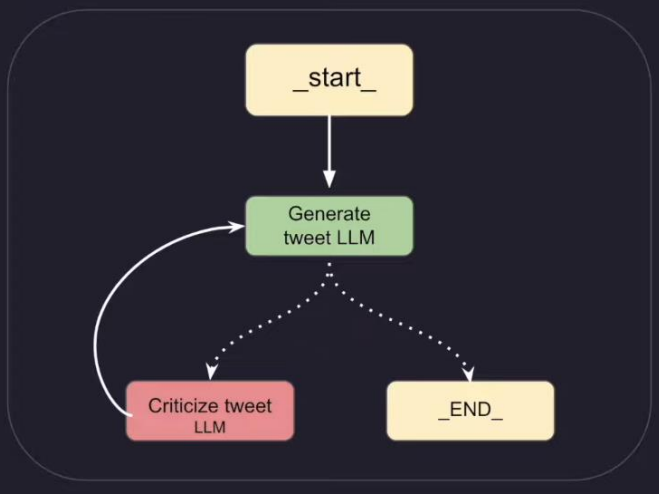




Core Components of LangGraph

1. Nodes
2. Edges
3. Conditional Edges
4. State

Example: Reflection Agent pattern



LangGraph Crash Course #6 - Reflection Agent - Introduction

Harish Neel | AI
7.67K subscribers

Subscribe

121 Share Clip Save ...

5,600 views Feb 12, 2025 #python #artificialintelligence #langgraph

In this beginner-friendly playlist, I'll show you how to use LangGraph to build powerful AI-agents from scratch

Reflection Agents in LangGraph

1. What is a Reflection Agent System?
2. Three types of Reflection Agent Systems
3. Setup & Installations
4. Implement a reflection Agent System

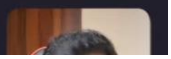
Reflection Agent pattern in LangGraph

But what does the English word "reflection" mean?

Like how you're looking at your reflection in the mirror, reflection means looking at yourself or your actions

For example:

- After giving a presentation, thinking about how it went
- After writing an email, reading it again to check if it's clear
- After making a decision, considering if it was the right choice



Reflection Agent pattern in LangGraph

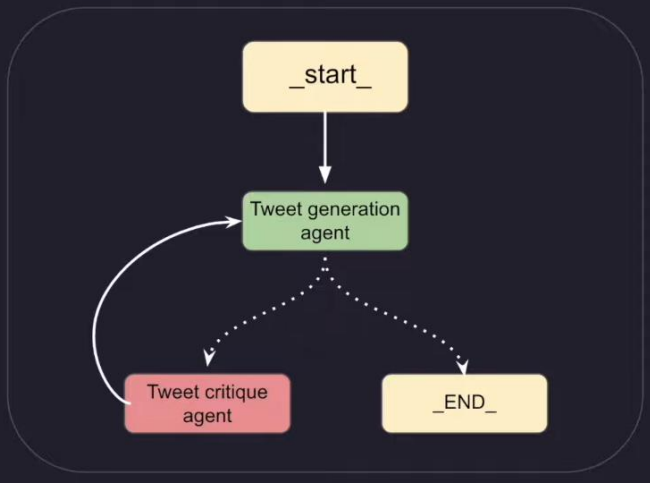
So what is a reflection-agent pattern?

A reflection agent pattern is an AI system pattern that can look at its own outputs and think about them/make it better - just like how we look at ourselves in a mirror and self-reflect, make ourselves better

A basic reflection agent system typically consists of:

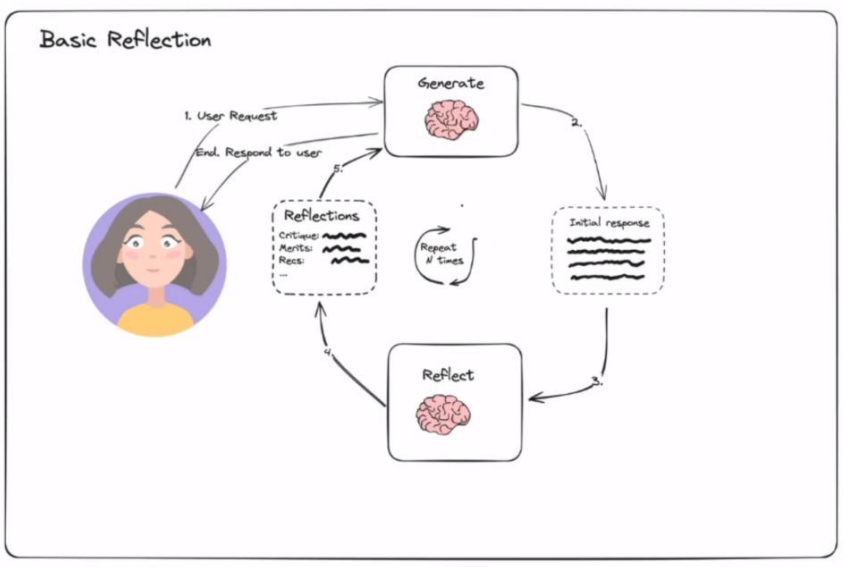
1. A generator agent
2. A reflector agent

Example: Basic Reflection Agent pattern

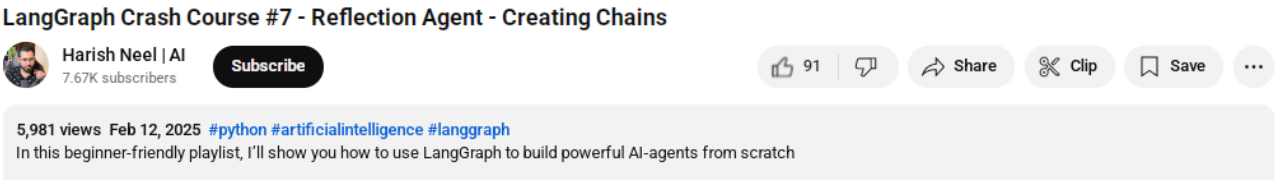


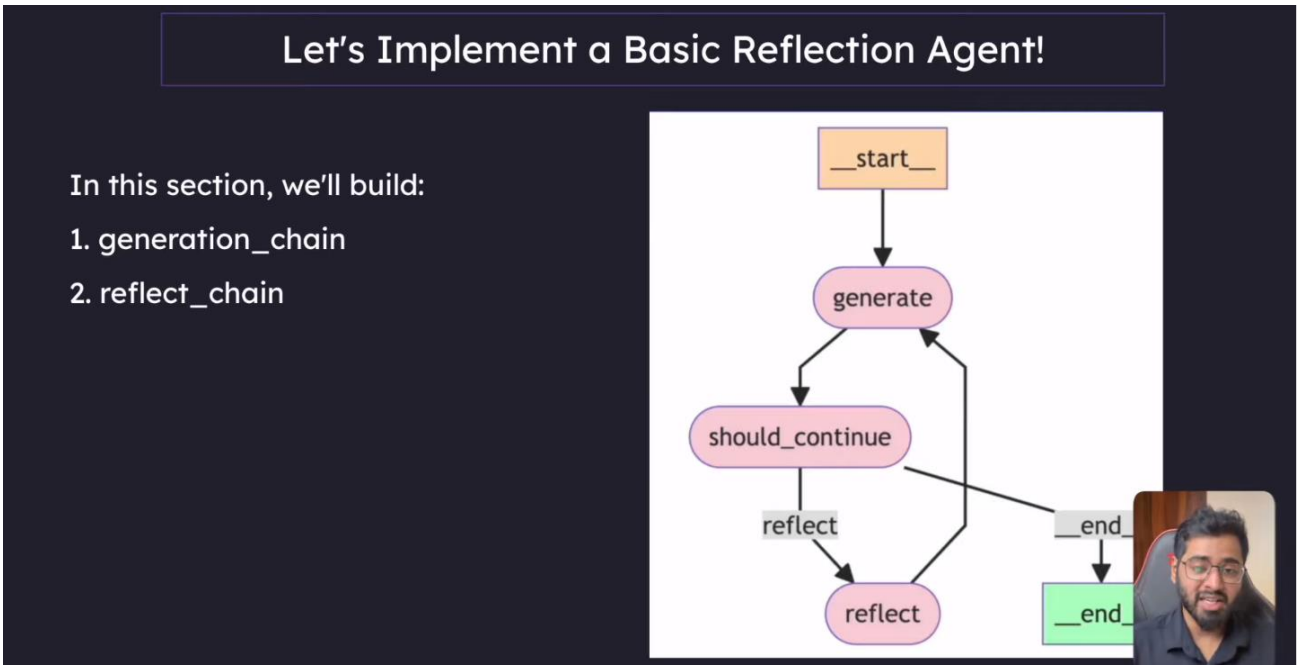
Reflection Agent pattern in LangGraph

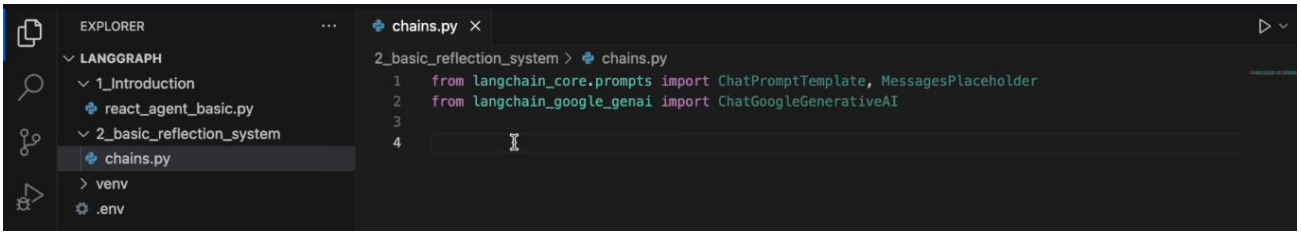
System 1	System 2
Fast	Slow
Subconscious	Conscious
Automatic	Effortful
Everyday Decisions	Complex Decisions
Error Prone	Reliable



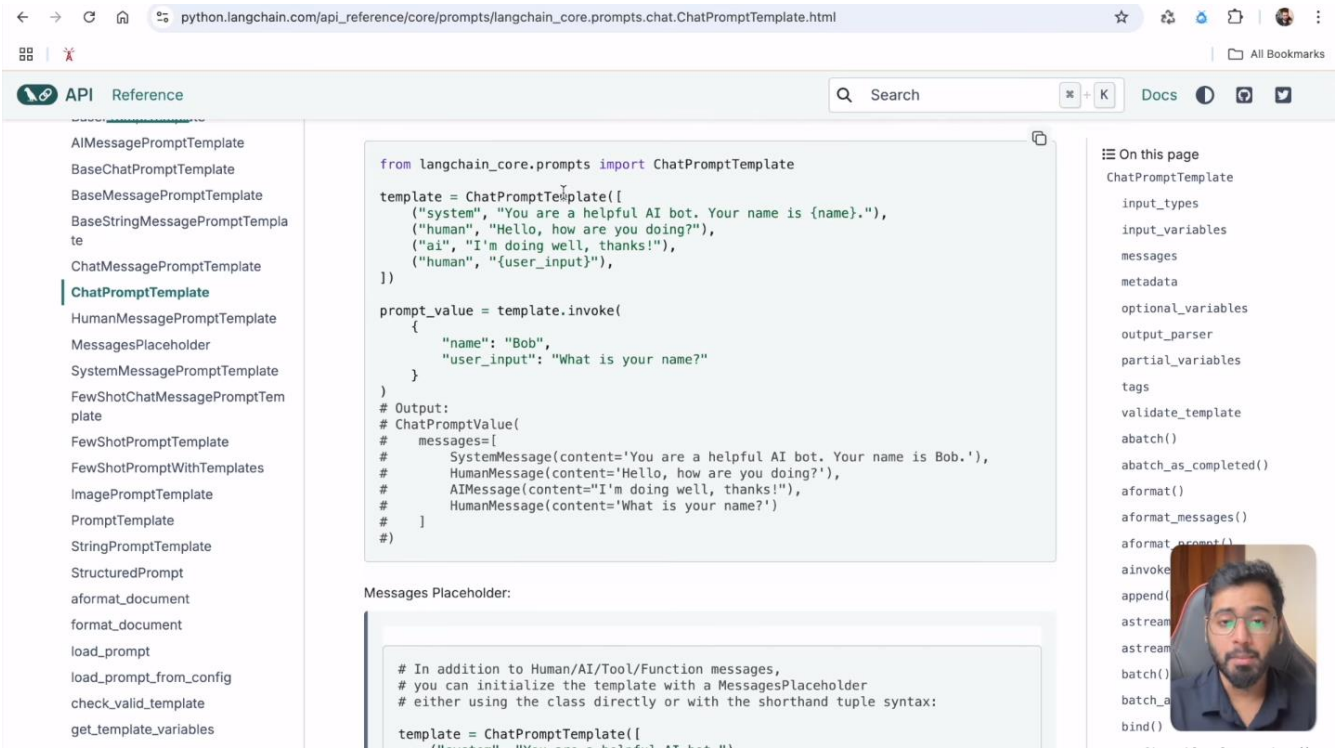
Simple Reflection Loop

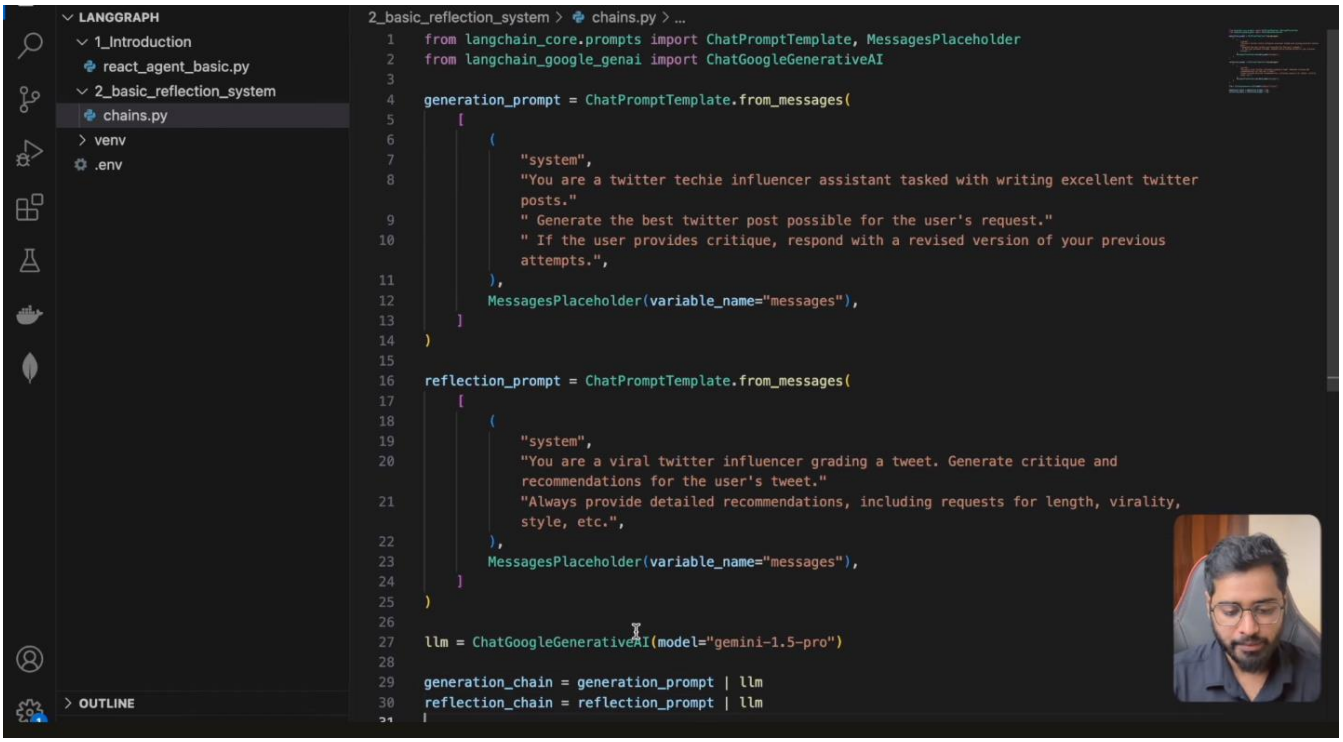




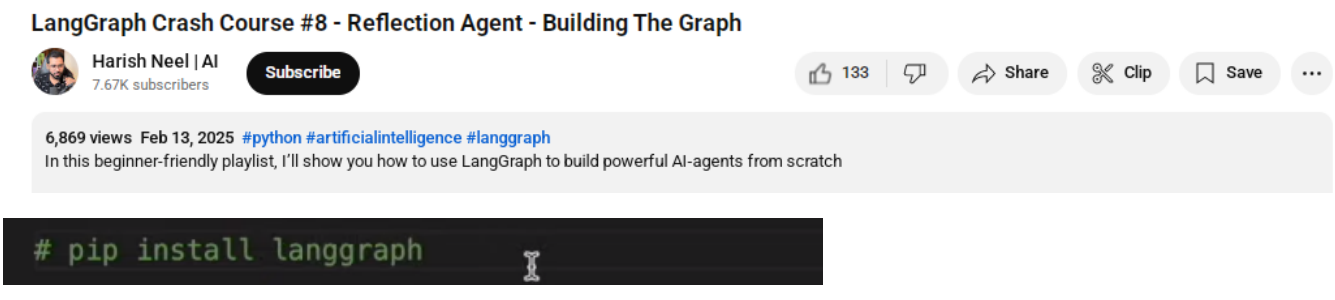


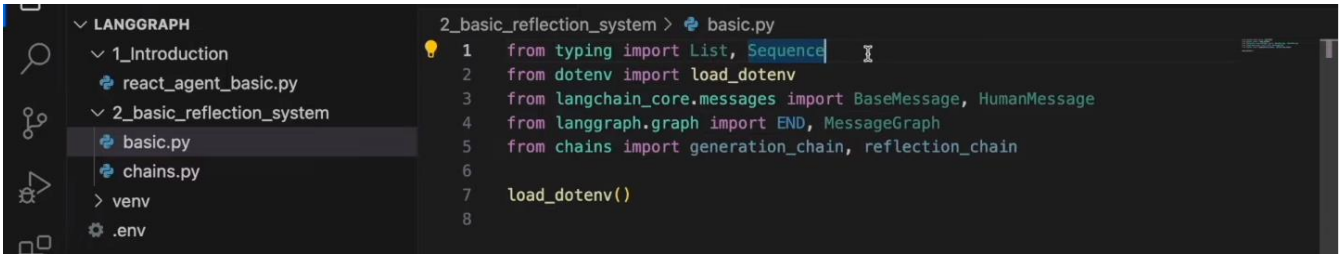
This is the file where we will put all our chains, it is not the file for the actual graph that will import the chains for use. We will have 2 chains, for the reflector chain and the generator chain.





We have now created our 2 chains and import them into our LangGraph graph as below





Basic Reflection Agent!

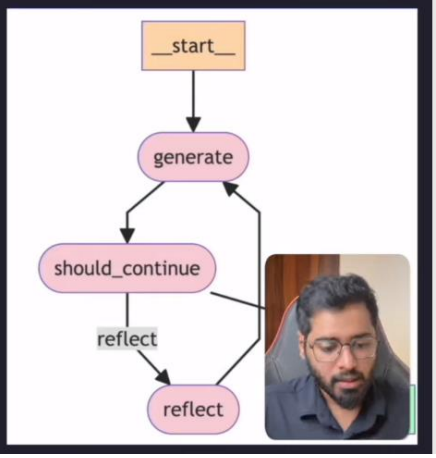
What is a MessageGraph?

It is a class that LangGraph provides that we can use to orchestrate the flow of messages between different nodes

Example use cases: Simple routing decisions, simple chatbot conversation flow

If you just want to pass messages along between nodes, then go for MessageGraph

If the app requires complex state management, we have StateGraph (more on this later)



Basic Reflection Agent!

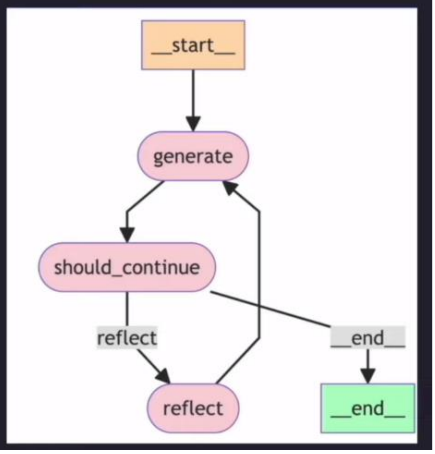
What is a MessageGraph?

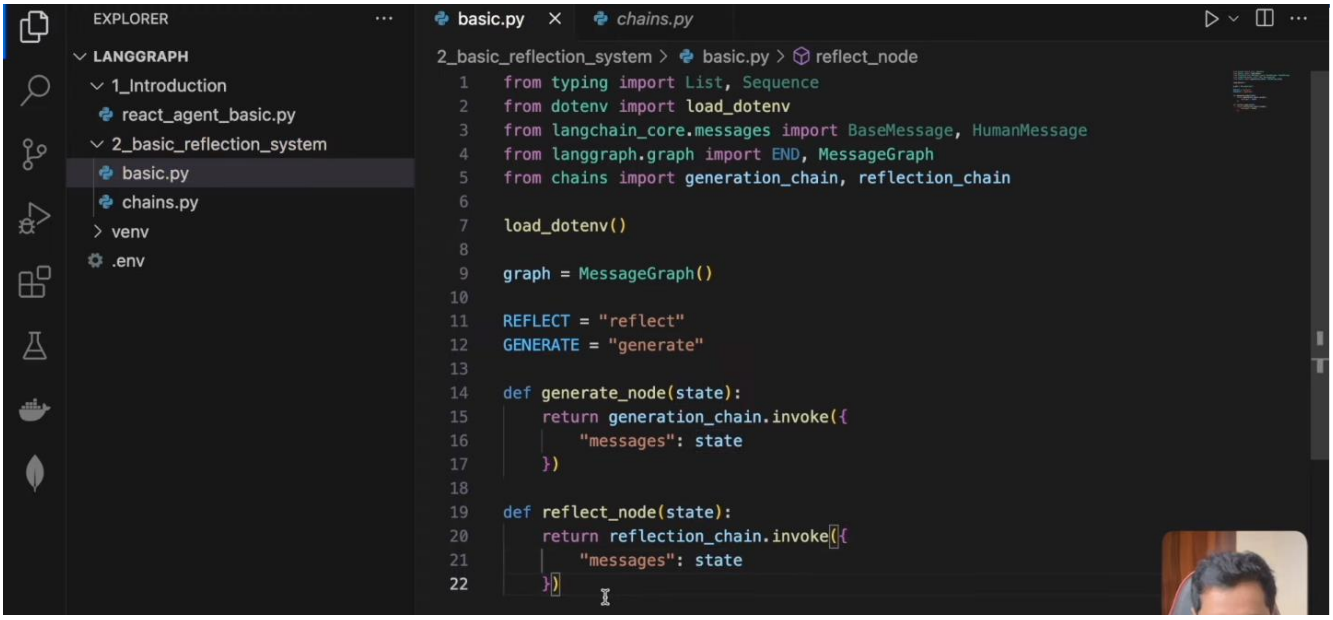
To put it simply, MessageGraph maintains a list of messages and decides the flow of those messages between nodes

Every node in MessageGraph receives the full list of previous messages as input

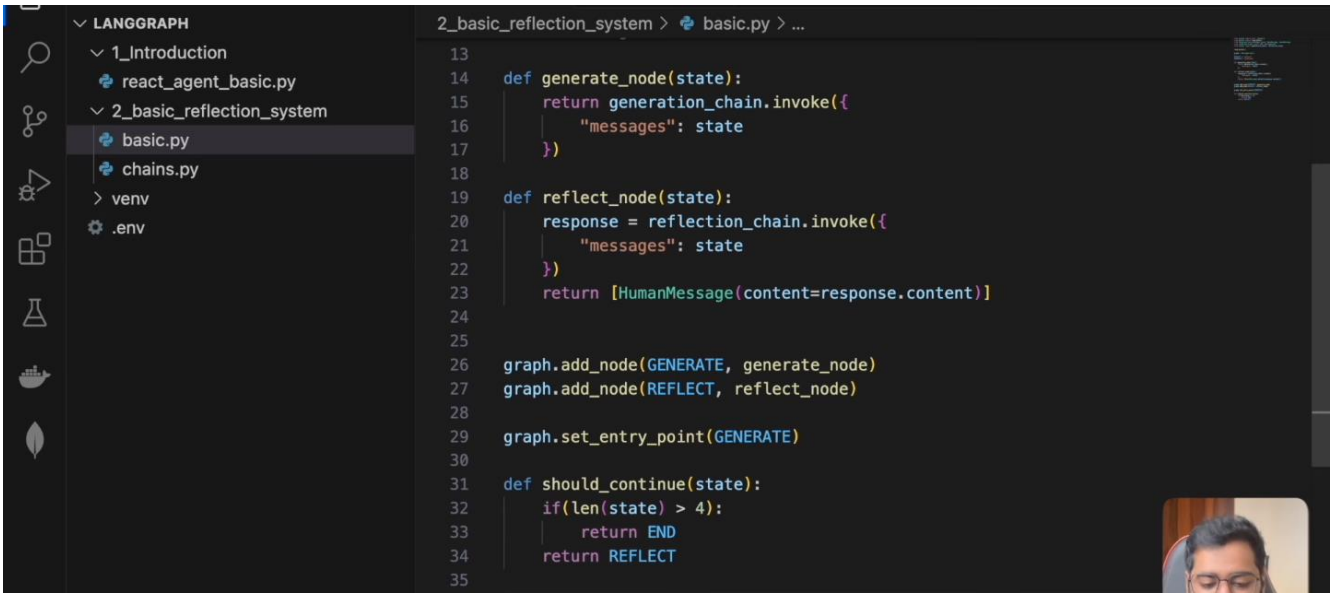
Each node can append new messages to the list and return it

The updated message list is then passed to the next node





We then can add the nodes and we can also trick the system into thinking the reflection is done by a human as below



Next, we created the **should_continue** function that should stop the loop after N iterations as below

Basic Reflection Agent!

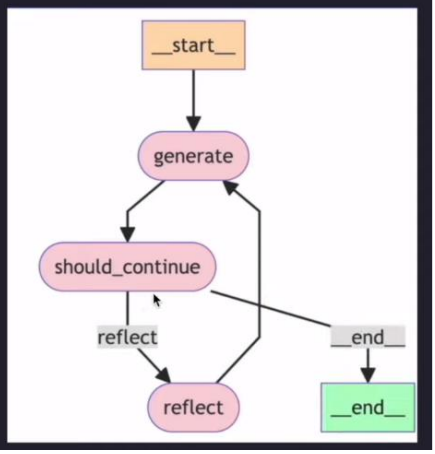
What is a MessageGraph?

To put it simply, MessageGraph maintains a list of messages and decides the flow of those messages between nodes

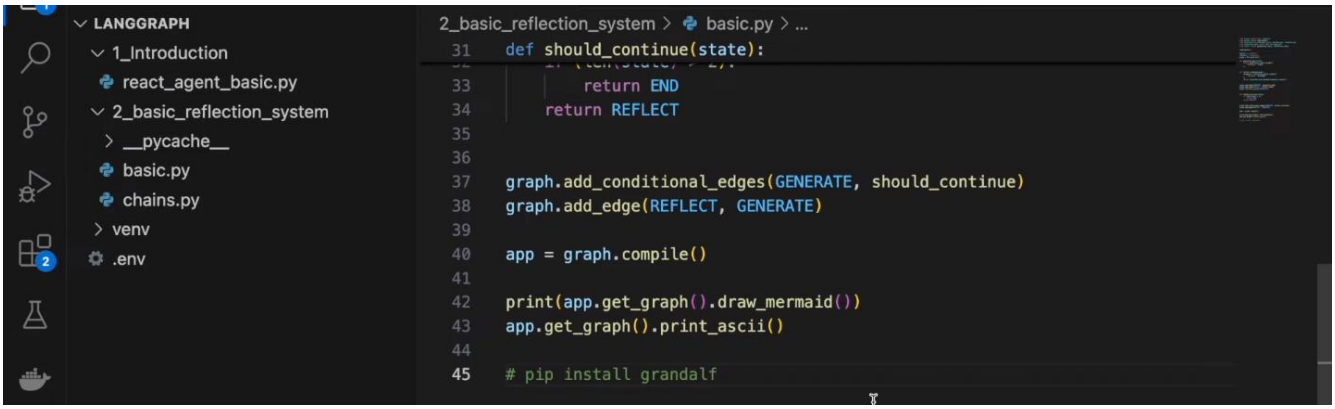
Every node in MessageGraph receives the full list of previous messages as input

Each node can append new messages to the list and return it

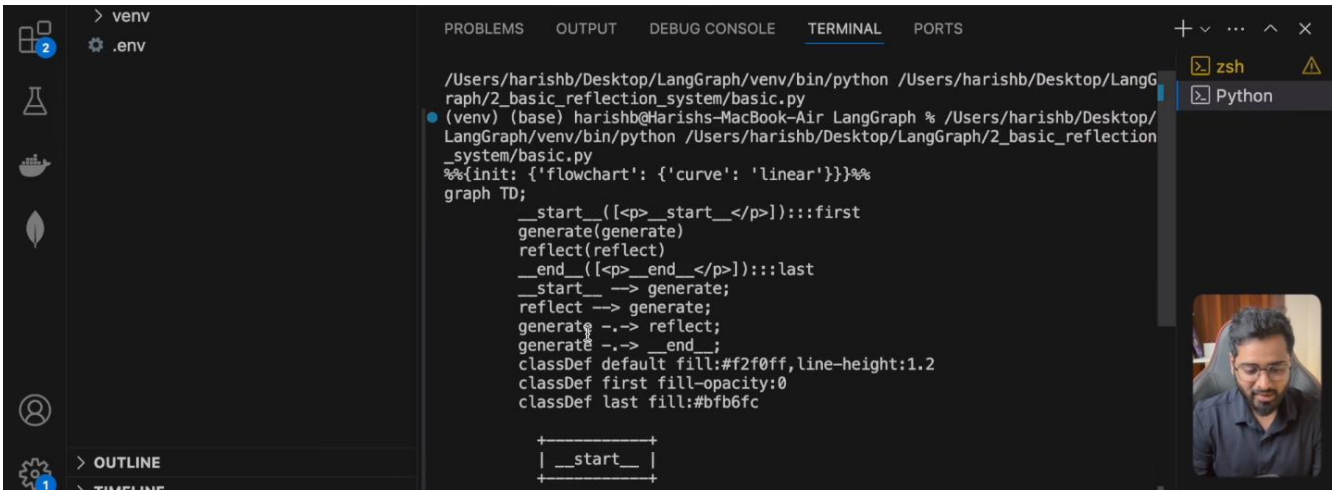
The updated message list is then passed to the next node

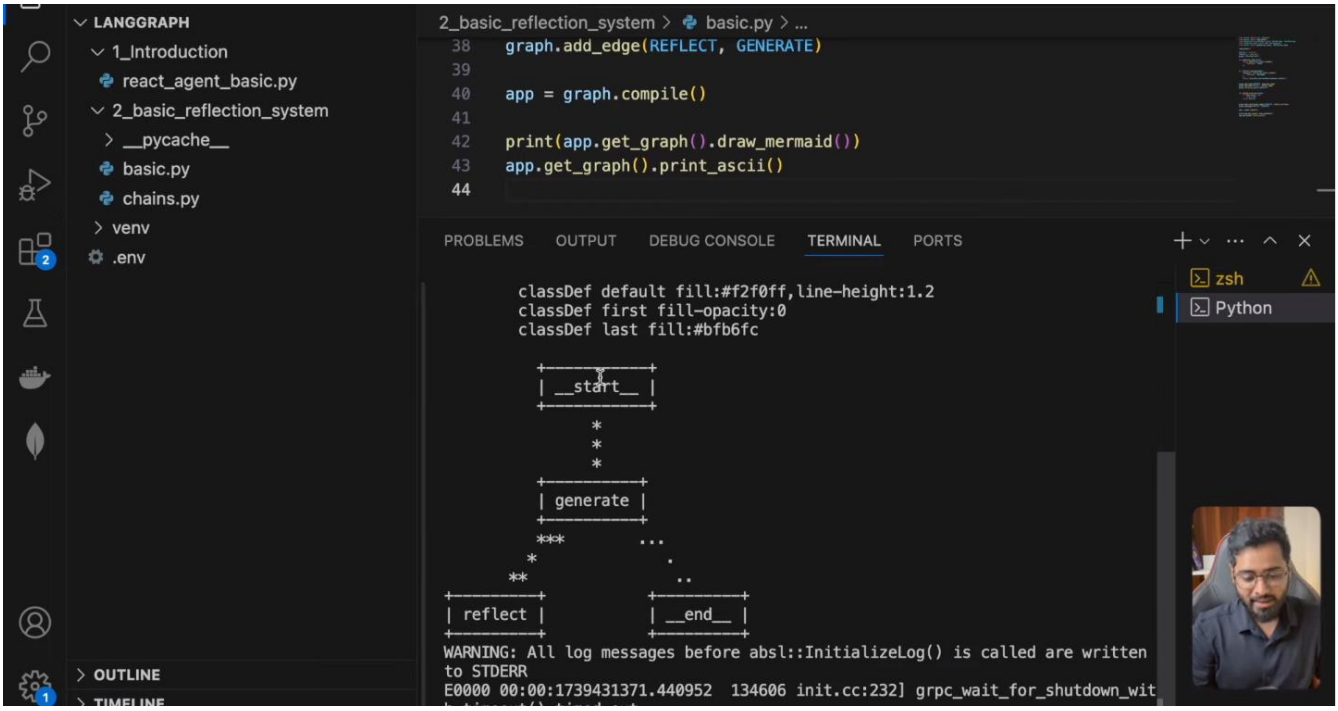


We can now start to connect the nodes using edges as below

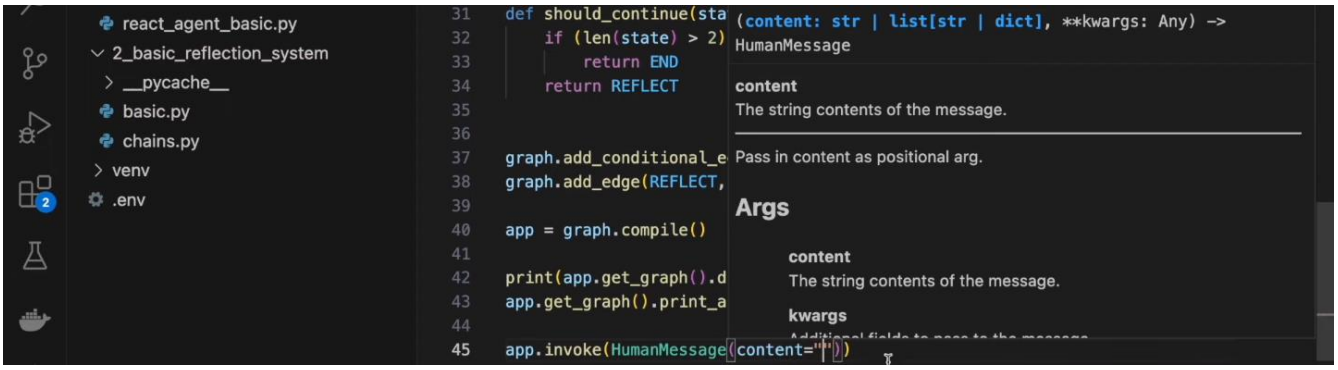


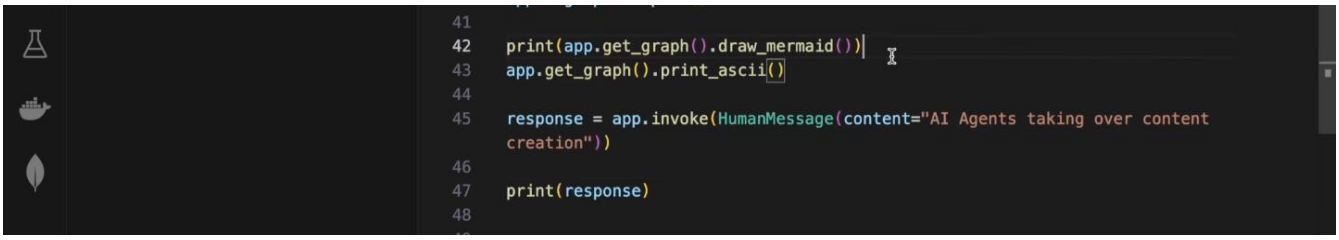
We then draw out a simple visualization of the graph, we might need to install the grandalf package, **pip install grandalf**

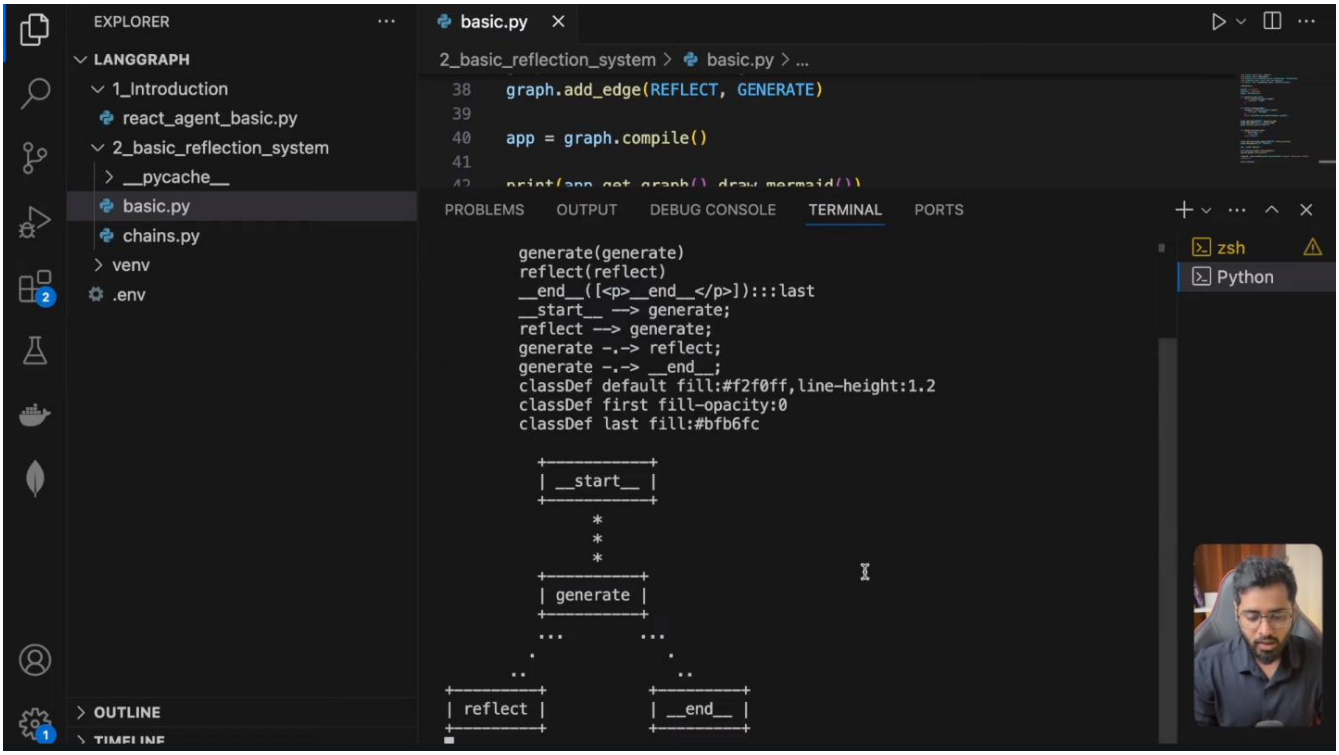


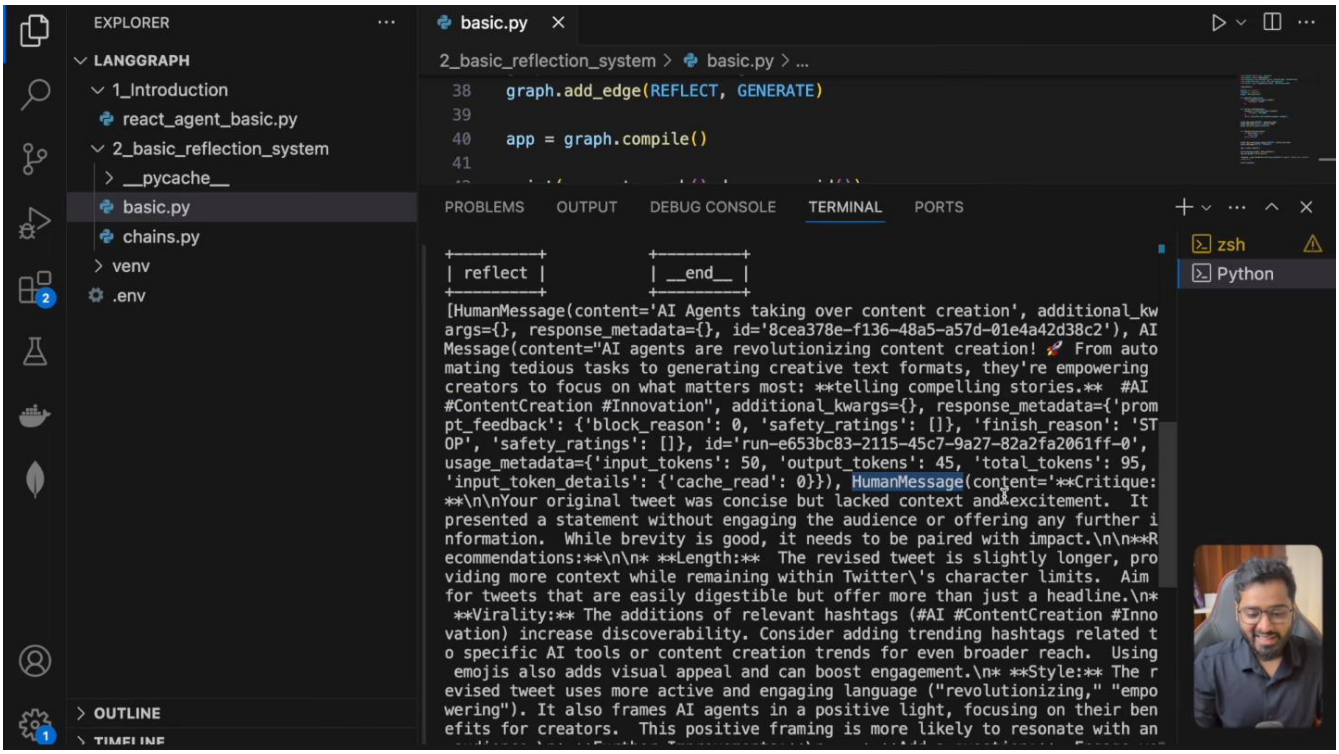


Next, we can now get the agent some HumanMessage context so that it can generate the tweet for us

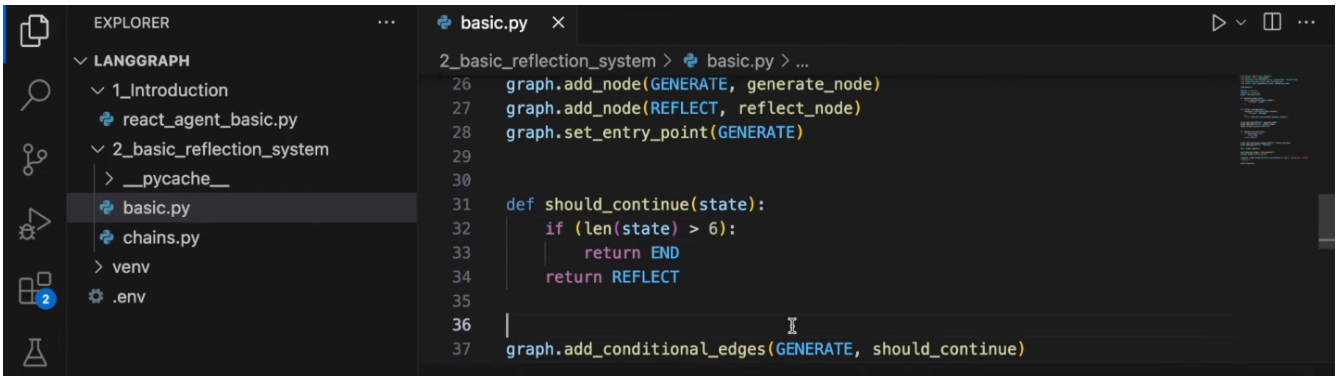




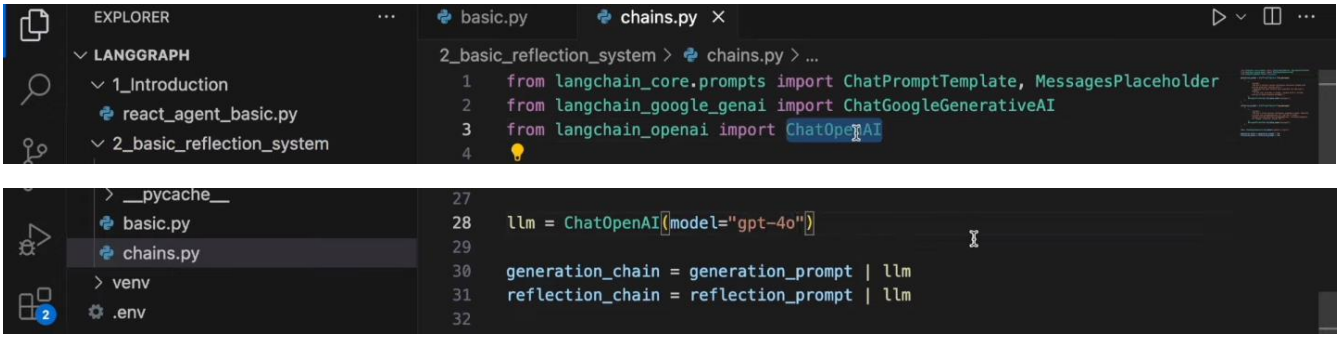


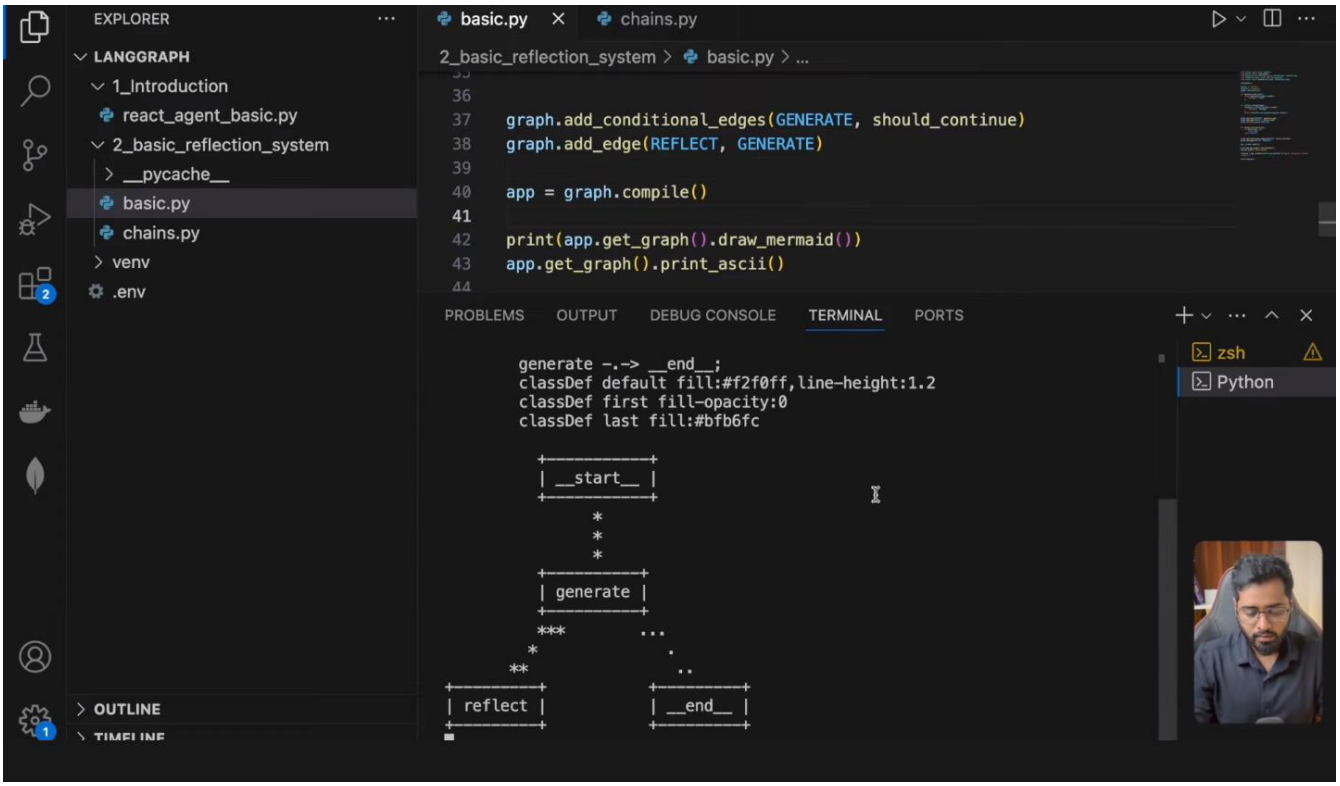


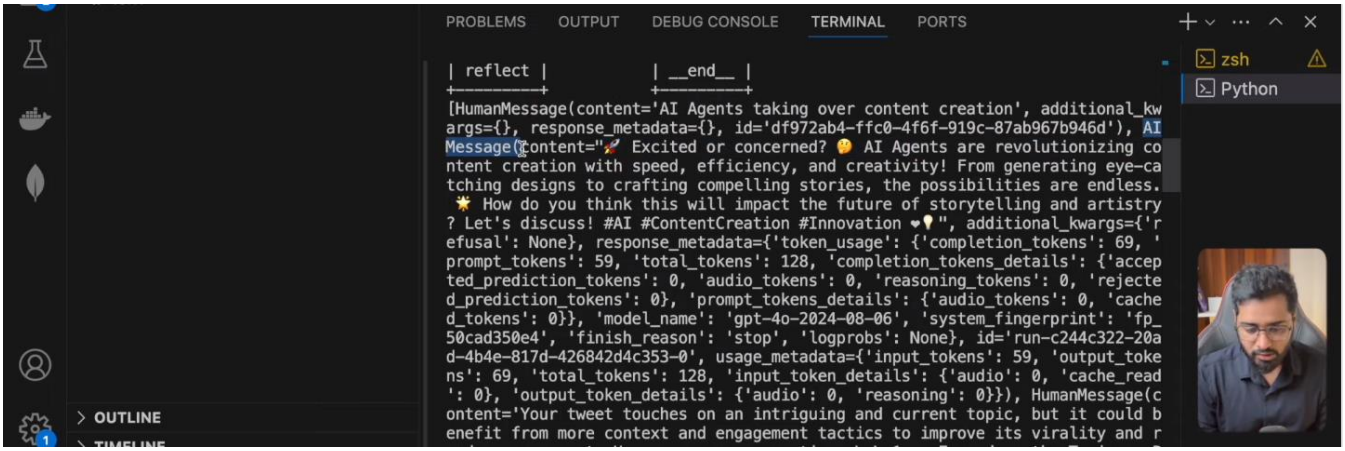
We can see the flow between **AIMessage** and **HumanMessage** that critiques the AIMessage for refinement.

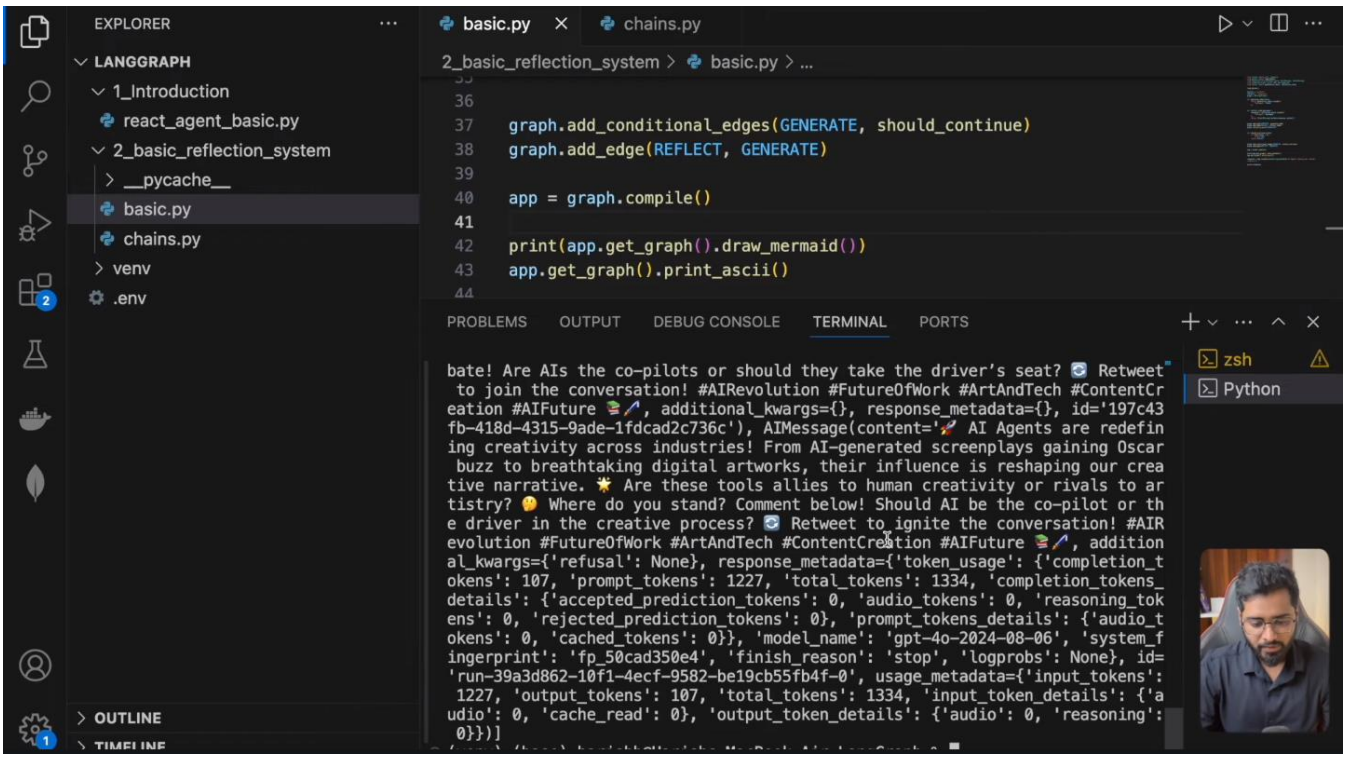


Increase the loops to 6 as above and try again but use the OpenAI LLM by updating the LLM in chains.py as below. Do a **pip install langchain_openai** command

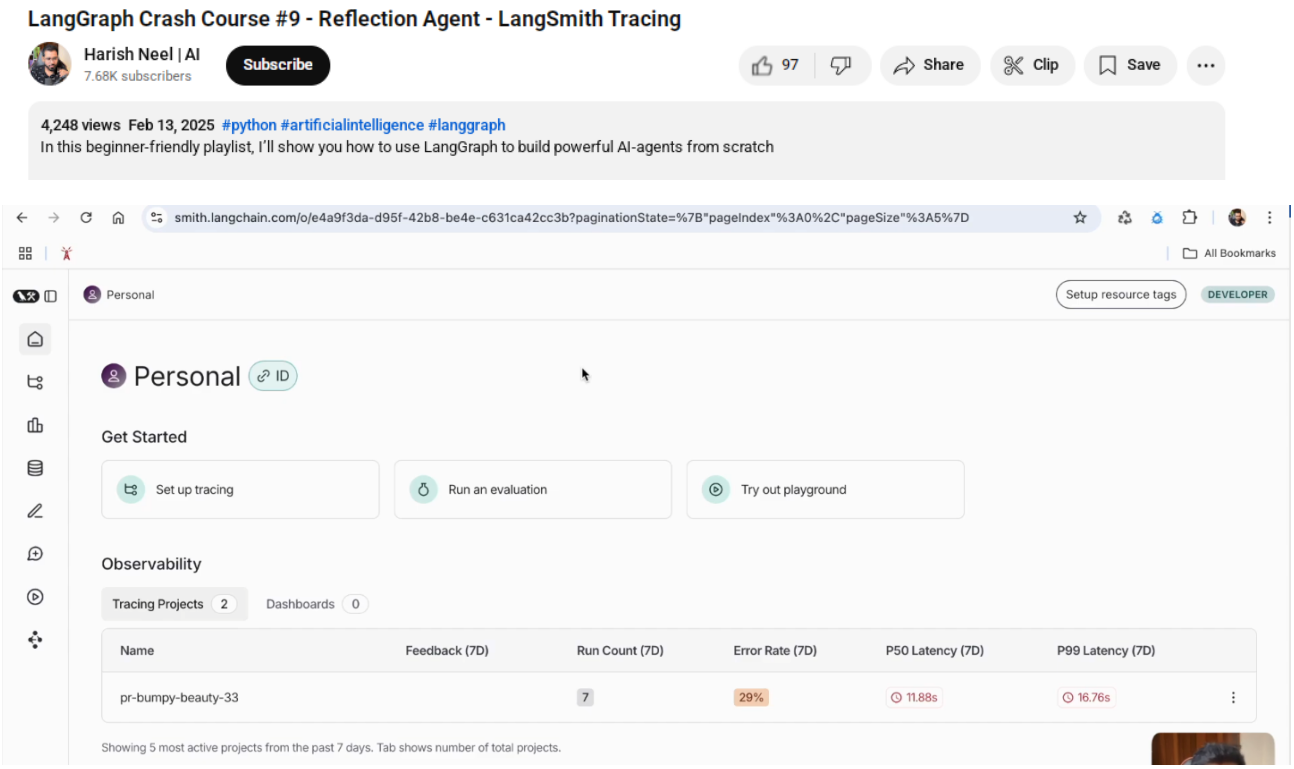




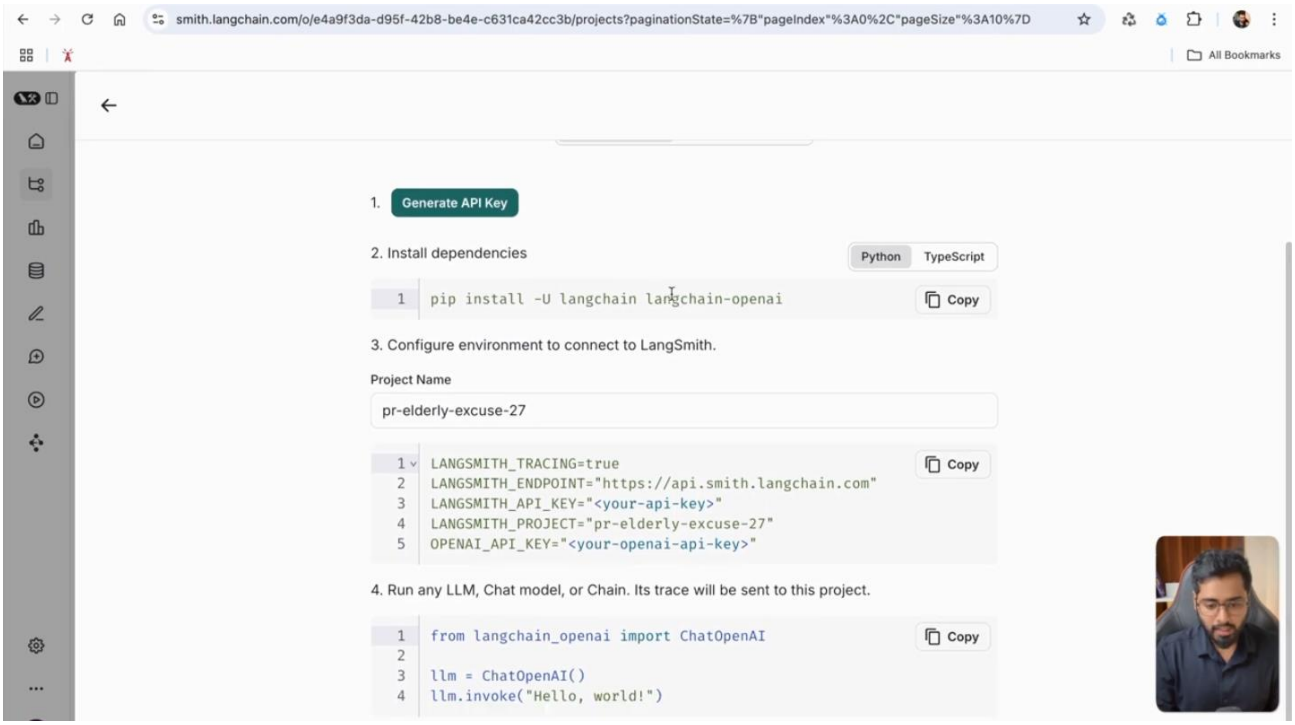


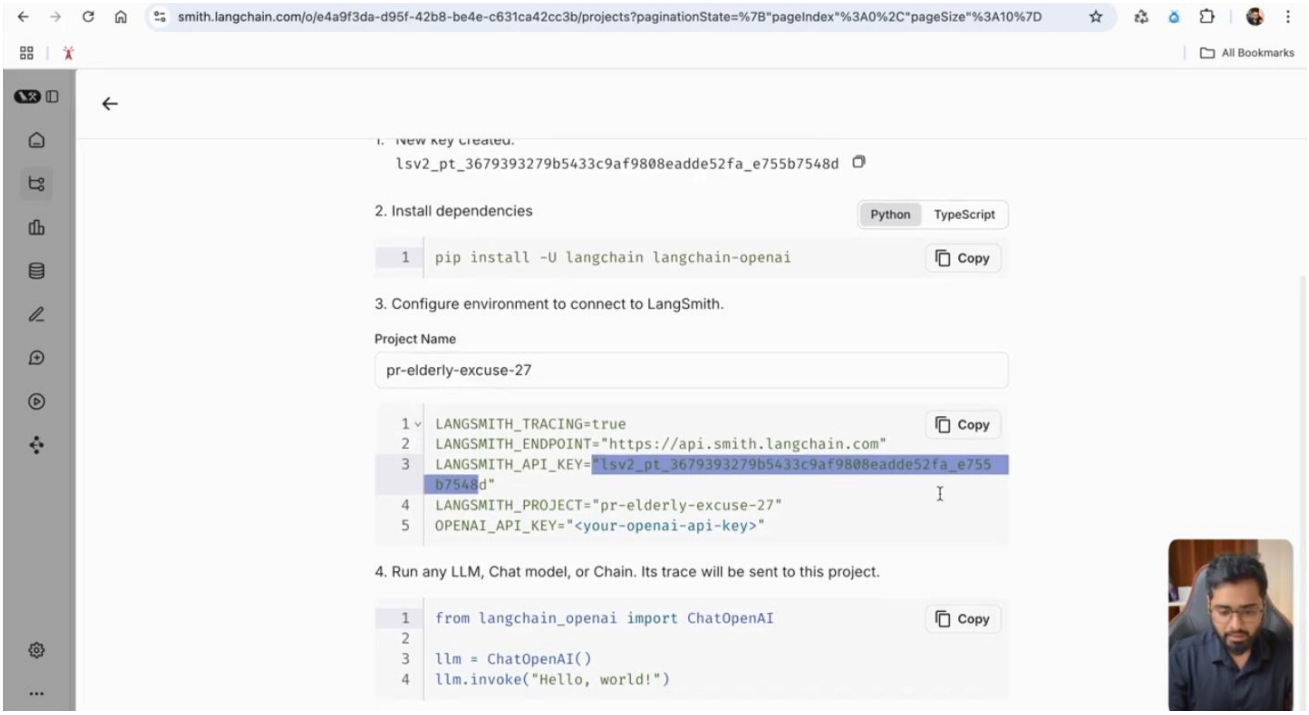


Next, we will be parsing and tracing the entire messages list using LangSmith.

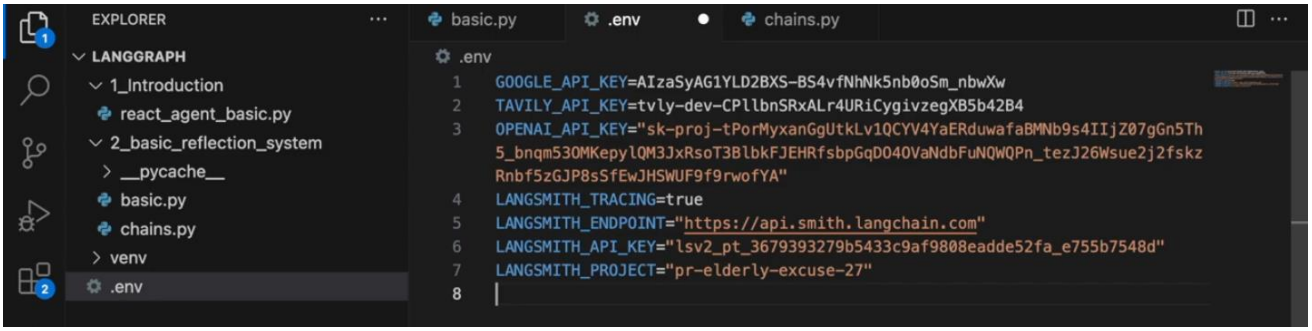


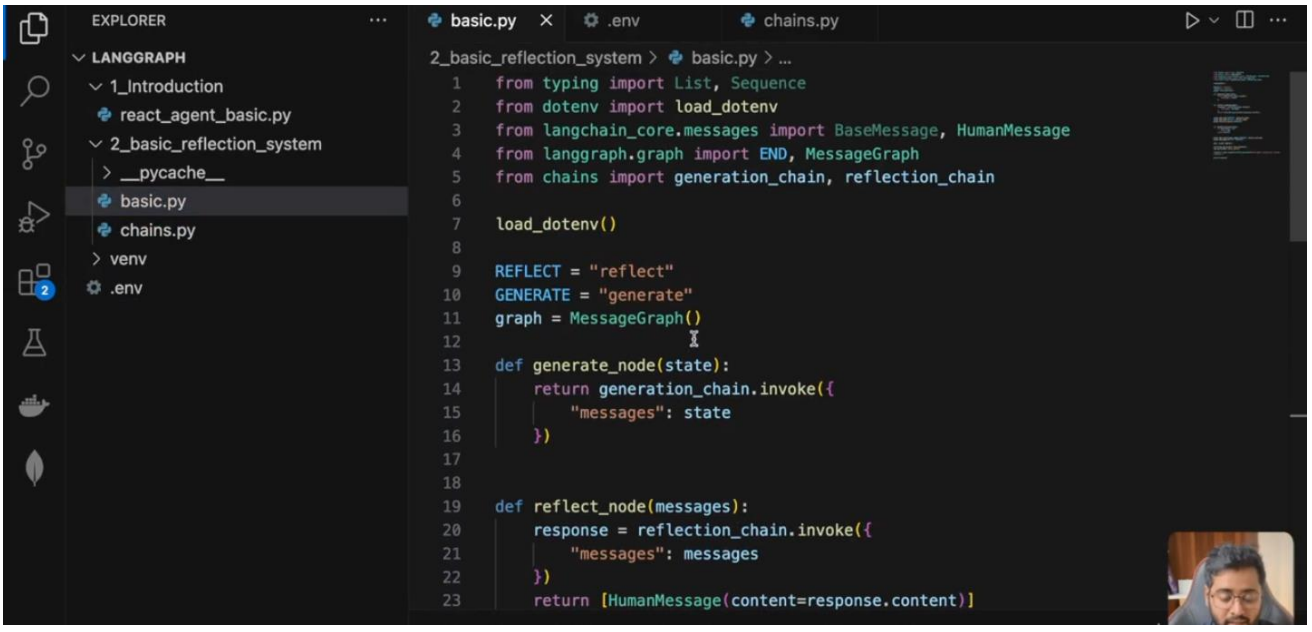
Let us now trace the Reflection agent that we built earlier to better understand how the agents work. Create an account at **smith.langchain.com**

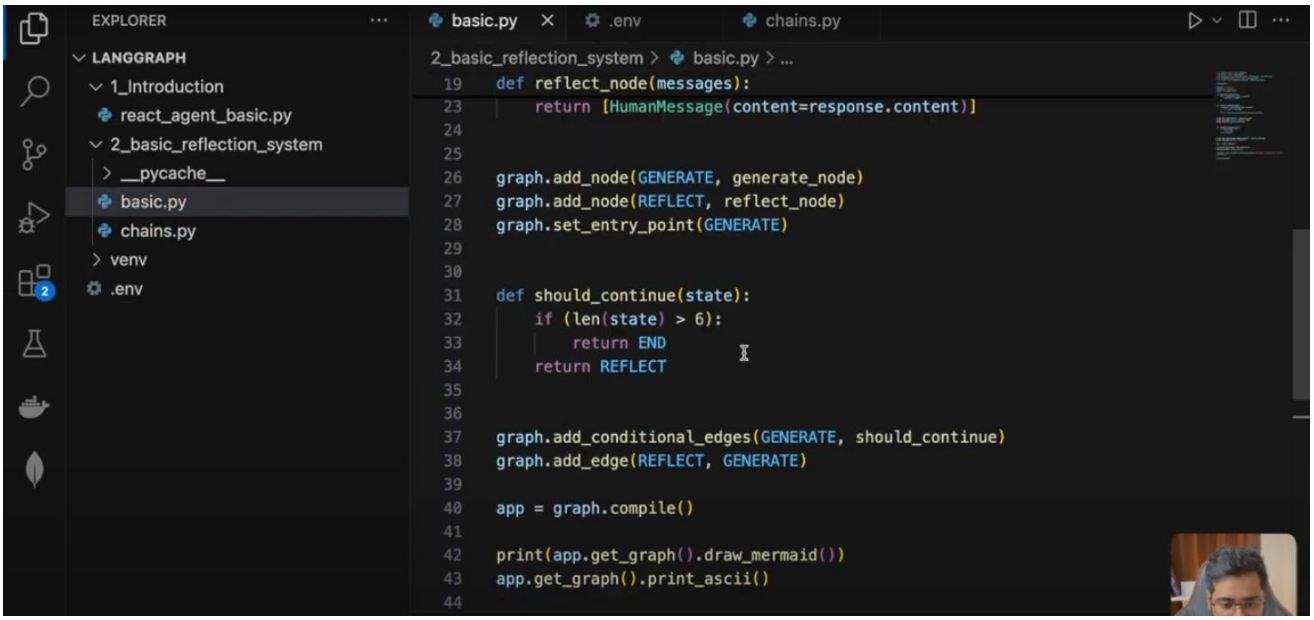


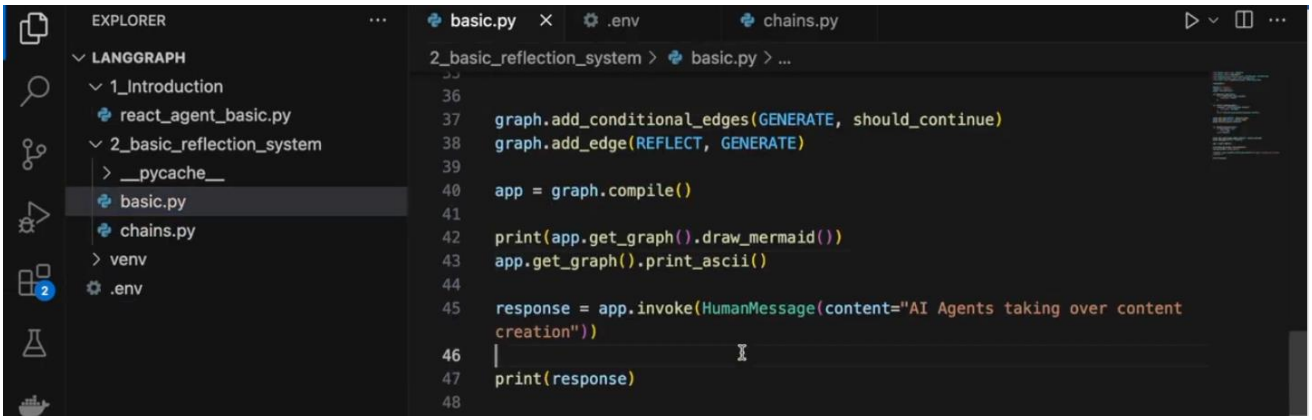


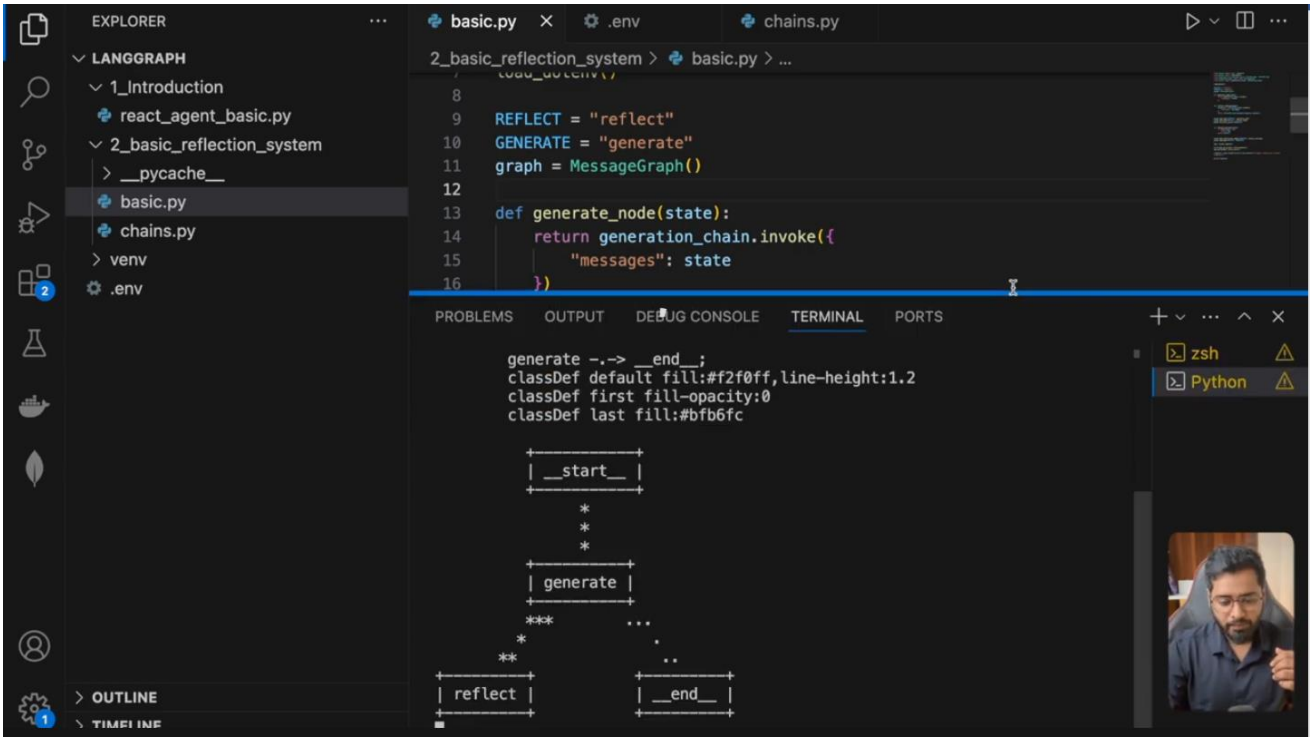
Also make sure you install the dependencies as **pip install -U langchain langchain-openai**

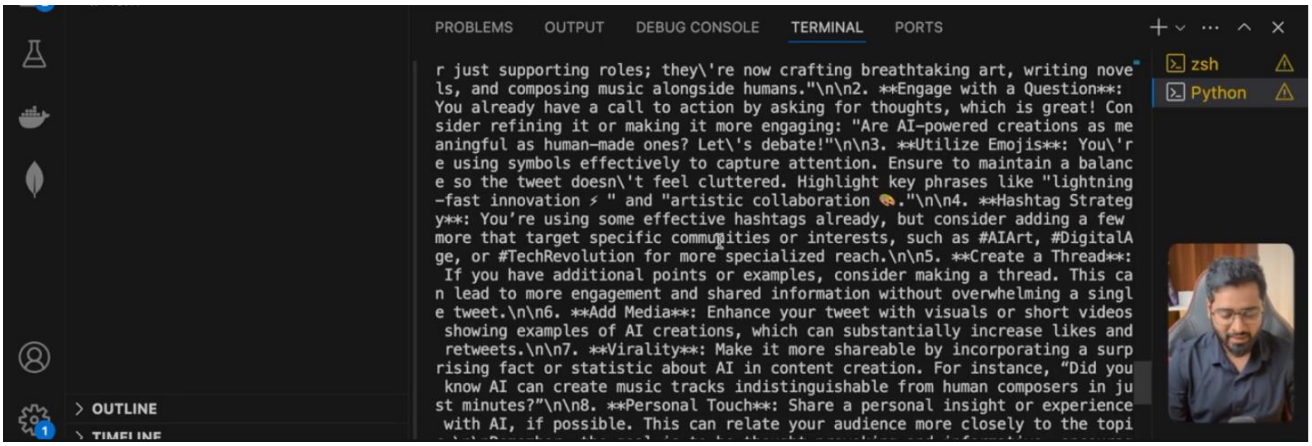


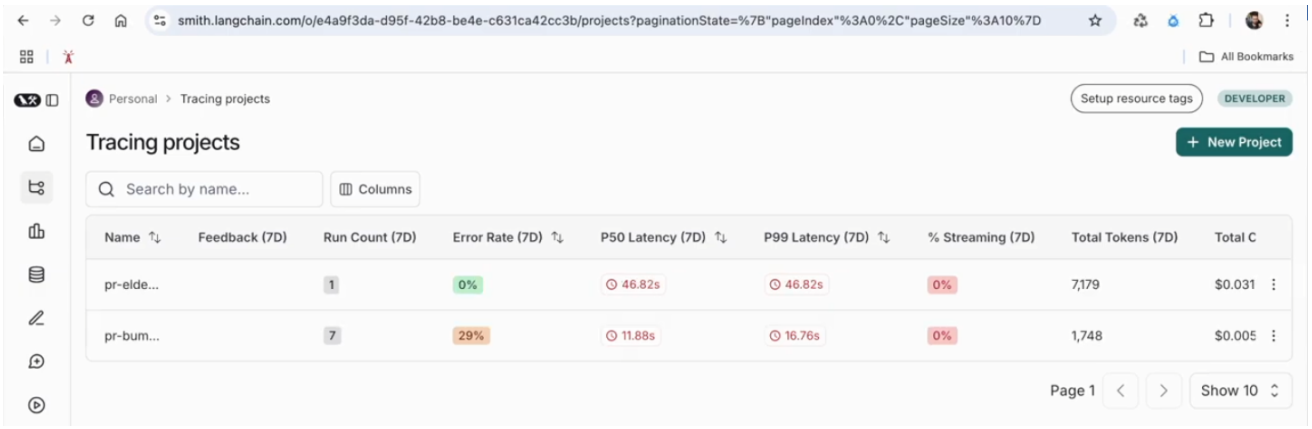


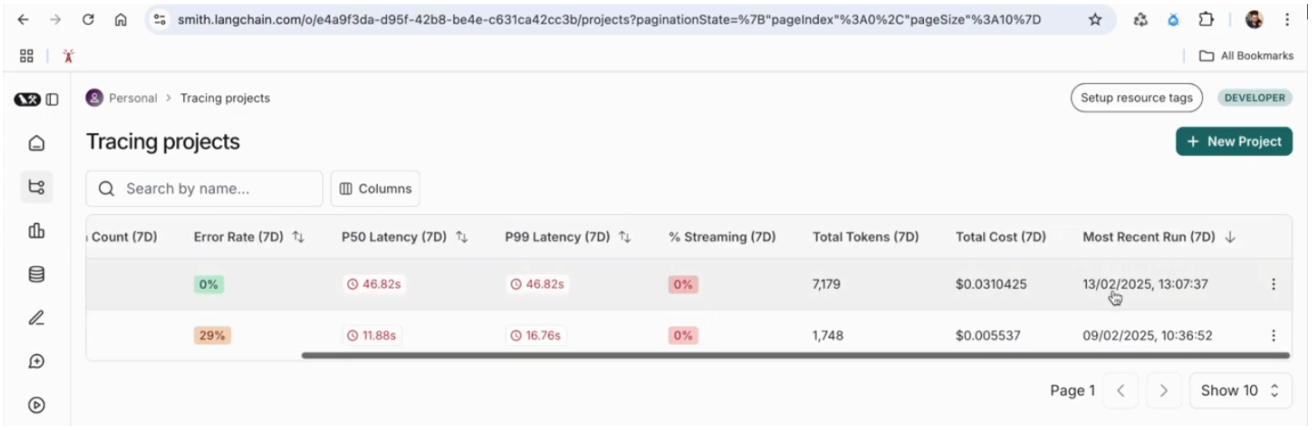

EXPLORER
LANGGRAPH
1_Introduction
react_agent_basic.py
2_basic_reflection_system
__pycache__
basic.py
chains.py
venv
.env
basic.py
.env
chains.py
2_basic_reflection_system > basic.py > ...
def reflect_node(messages):
return [HumanMessage(content=response.content)]
graph.add_node(GENERATE, generate_node)
graph.add_node(REFLECT, reflect_node)
graph.set_entry_point(GENERATE)
def should_continue(state):
if (len(state) > 6):
return END
return REFLECT
graph.add_conditional_edges(GENERATE, should_continue)
graph.add_edge(REFLECT, GENERATE)
app = graph.compile()
print(app.get_graph().draw_mermaid())
app.get_graph().print_ascii()

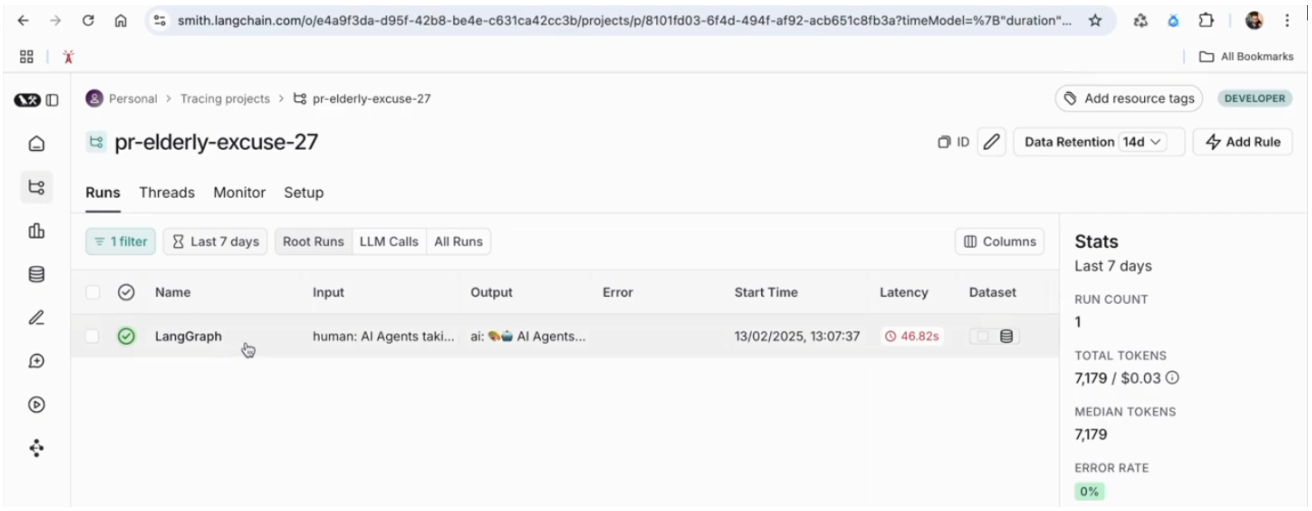

EXPLORER
LANGGRAPH
1_Introduction
react_agent_basic.py
2_basic_reflection_system
__pycache__
basic.py
chains.py
venv
.env
basic.py
.env
chains.py
2_basic_reflection_system > basic.py > ...
graph.add_conditional_edges(GENERATE, should_continue)
graph.add_edge(REFLECT, GENERATE)
app = graph.compile()
print(app.get_graph().draw_mermaid())
app.get_graph().print_ascii()
response = app.invoke(HumanMessage(content="AI Agents taking over content creation"))
print(response)

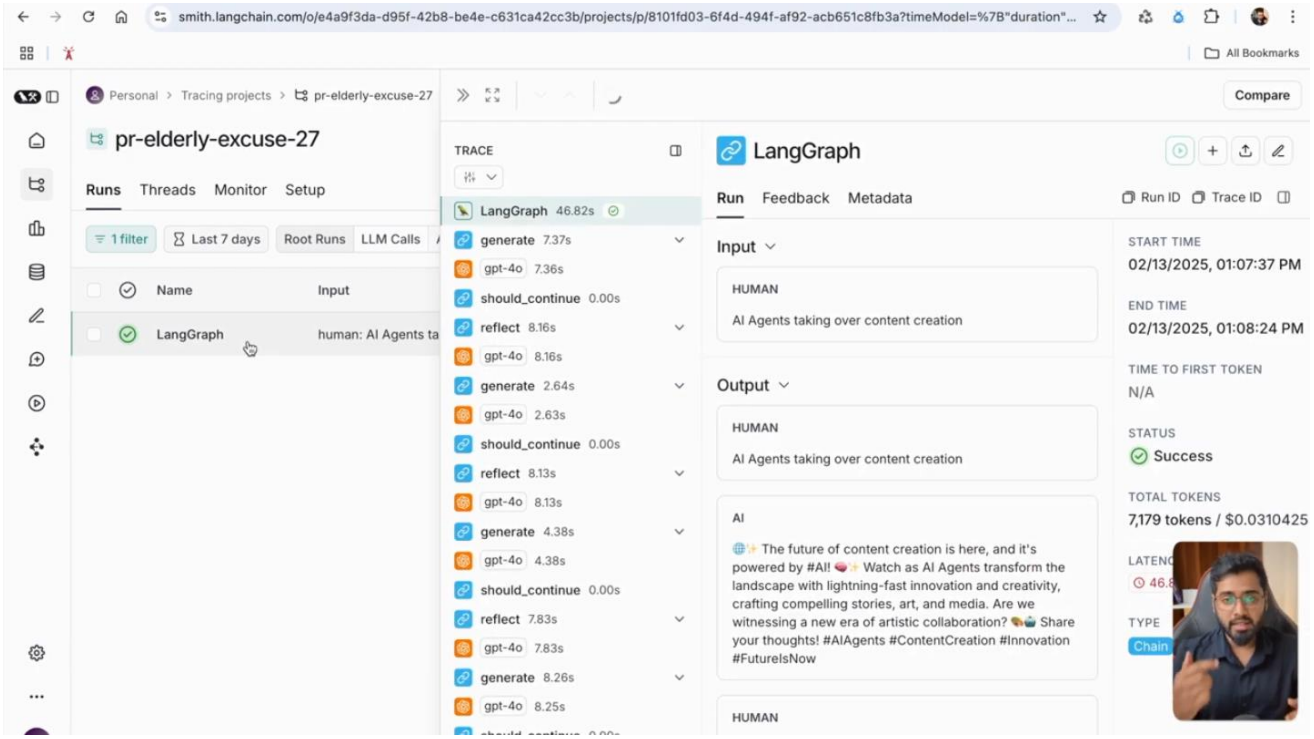

EXPLORER
LANGGRAPH
1_Introduction
react_agent_basic.py
2_basic_reflection_system
__pycache__
basic.py
chains.py
venv
.env
OUTLINE
basic.py
.env
chains.py
2_basic_reflection_system > basic.py > ...
REFLECT = "reflect"
GENERATE = "generate"
graph = MessageGraph()
def generate_node(state):
return generation_chain.invoke({
"messages": state
})
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
zsh
Python
generate -.-> __end__;
classDef default fill:#f2f0ff,line-height:1.2
classDef first fill-opacity:0
classDef last fill:#bfb6fc
__start__
generate
reflect
__end__

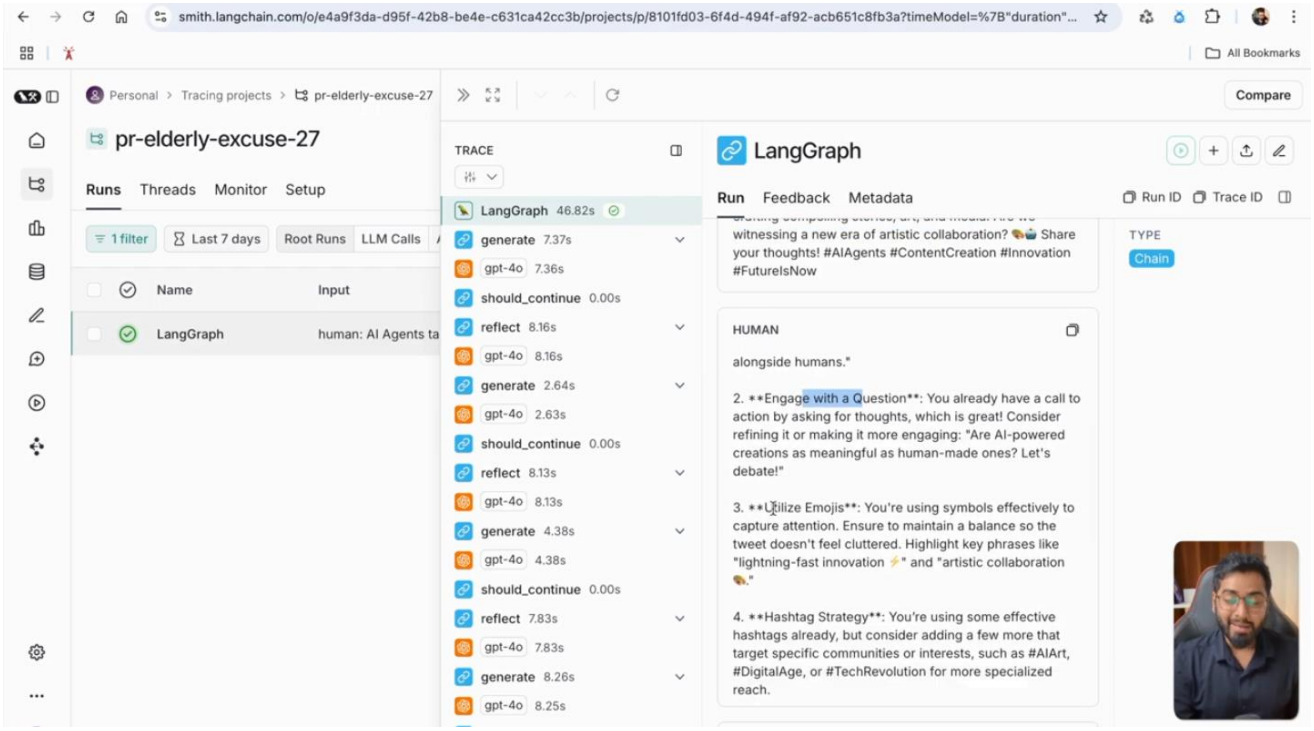

PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
zsh
Python
r just supporting roles; they\'re now crafting breathtaking art, writing nove
ls, and composing music alongside humans."\n\n2. **Engage with a Question**:
You already have a call to action by asking for thoughts, which is great! Con
sider refining it or making it more engaging: "Are AI-powered creations as me
aningful as human-made ones? Let\'s debate!"\n\n3. **Utilize Emojis**: You\'r
e using symbols effectively to capture attention. Ensure to maintain a balanc
e so the tweet doesn\'t feel cluttered. Highlight key phrases like "lightning
-fast innovation ⚡" and "artistic collaboration 🎨."\n\n4. **Hashtag Strateg
y**: You're using some effective hashtags already, but consider adding a few
more that target specific communities or interests, such as #AIArt, #DigitalA
ge, or #TechRevolution for more specialized reach.\n\n5. **Create a Thread**:
If you have additional points or examples, consider making a thread. This ca
n lead to more engagement and shared information without overwhelming a singl
e tweet.\n\n6. **Add Media**: Enhance your tweet with visuals or short videos
showing examples of AI creations, which can substantially increase likes and
retweets.\n\n7. **Virality**: Make it more shareable by incorporating a surp
rising fact or statistic about AI in content creation. For instance, "Did you
know AI can create music tracks indistinguishable from human composers in ju
st minutes?"\n\n8. **Personal Touch**: Share a personal insight or experience
with AI, if possible. This can relate your audience more closely to the topi
OUTLINE
TIMELINE


smith.langchain.com/o/e4a9f3da-d95f-42b8-be4e-c631ca42cc3b/projects?paginationState=%7B"pageIndex"%3A0%2C"pageSize"%3A10%7D
All Bookmarks
Personal > Tracing projects
Setup resource tags
DEVELOPER
Tracing projects
New Project
Search by name...
Columns
Name
Feedback (7D)
Run Count (7D)
Error Rate (7D)
P50 Latency (7D)
P99 Latency (7D)
% Streaming (7D)
Total Tokens (7D)
Total C
pr-elde...
1
0%
46.82s
46.82s
0%
7,179
$0.031
pr-bum...
7
29%
11.88s
16.76s
0%
1,748
$0.005
Page 1
Show 10


smith.langchain.com/o/e4a9f3da-d95f-42b8-be4e-c631ca42cc3b/projects?paginationState=%7B"pageIndex"%3A0%2C"pageSize"%3A10%7D
All Bookmarks
Personal > Tracing projects
Setup resource tags
DEVELOPER
Tracing projects
New Project
Search by name...
Columns
Count (7D)
Error Rate (7D)
P50 Latency (7D)
P99 Latency (7D)
% Streaming (7D)
Total Tokens (7D)
Total Cost (7D)
Most Recent Run (7D)
0%
46.82s
46.82s
0%
7,179
$0.0310425
13/02/2025, 13:07:37
29%
11.88s
16.76s
0%
1,748
$0.005537
09/02/2025, 10:36:52
Page 1
Show 10


smith.langchain.com/o/e4a9f3da-d95f-42b8-be4e-c631ca42cc3b/projects/p/8101fd03-6f4d-494f-af92-acb651c8fb3a?timeModel=%7B"duration"...
All Bookmarks
Personal > Tracing projects > pr-elderly-excuse-27
Add resource tags
DEVELOPER
pr-elderly-excuse-27
ID
Data Retention 14d
Add Rule
Runs
Threads
Monitor
Setup
1 filter
Last 7 days
Root Runs
LLM Calls
All Runs
Columns
Name
Input
Output
Error
Start Time
Latency
Dataset
LangGraph
human: AI Agents taki...
ai: AI Agents...
13/02/2025, 13:07:37
46.82s
Stats
Last 7 days
RUN COUNT
1
TOTAL TOKENS
7,179 / $0.03
MEDIAN TOKENS
7,179
ERROR RATE
0%





LangGraph Crash Course #9.5 - Structured LLM Outputs

4,036 views Apr 2, 2025 #python #langgraph #artificialintelligence
In this beginner-friendly playlist, I'll show you how to use LangGraph to build powerful AI-agents from scratch

Structured Outputs

It is often useful to have a model return output that matches a specific schema that we define

```
Input: "Tell me a joke about cats"

Output:

{
  'setup': 'Why was the cat sitting on the computer?',
  'punchline': 'Because it wanted to keep an eye on the mouse!',
  'rating': 7
 }
```

We have options to get outputs in formats such as - JSON, Dictionary, string, YAM

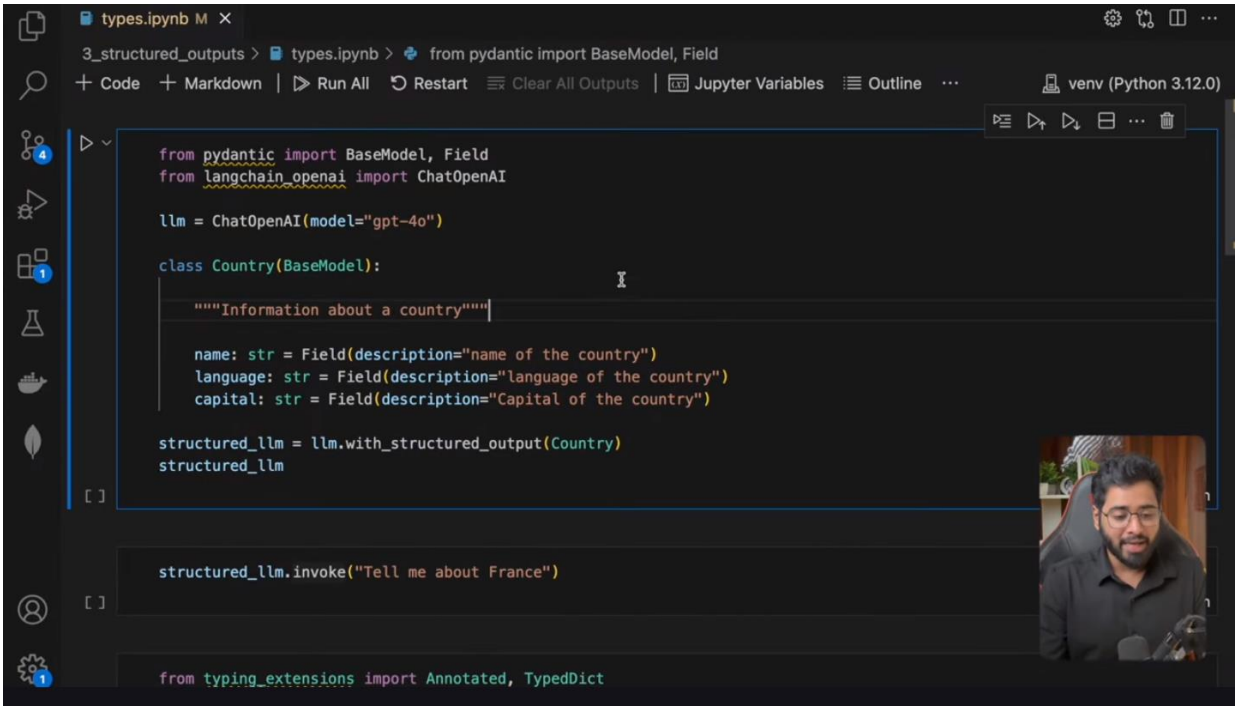
Structured Outputs

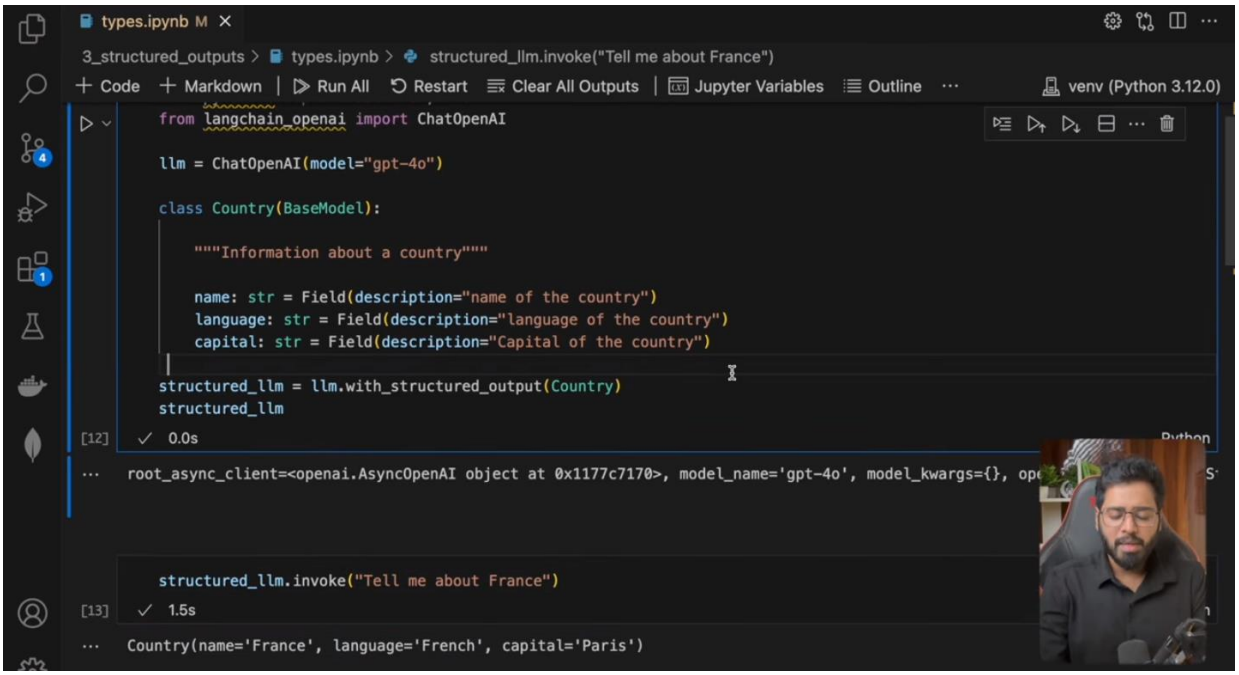
Pydantic Models for Structured Outputs:

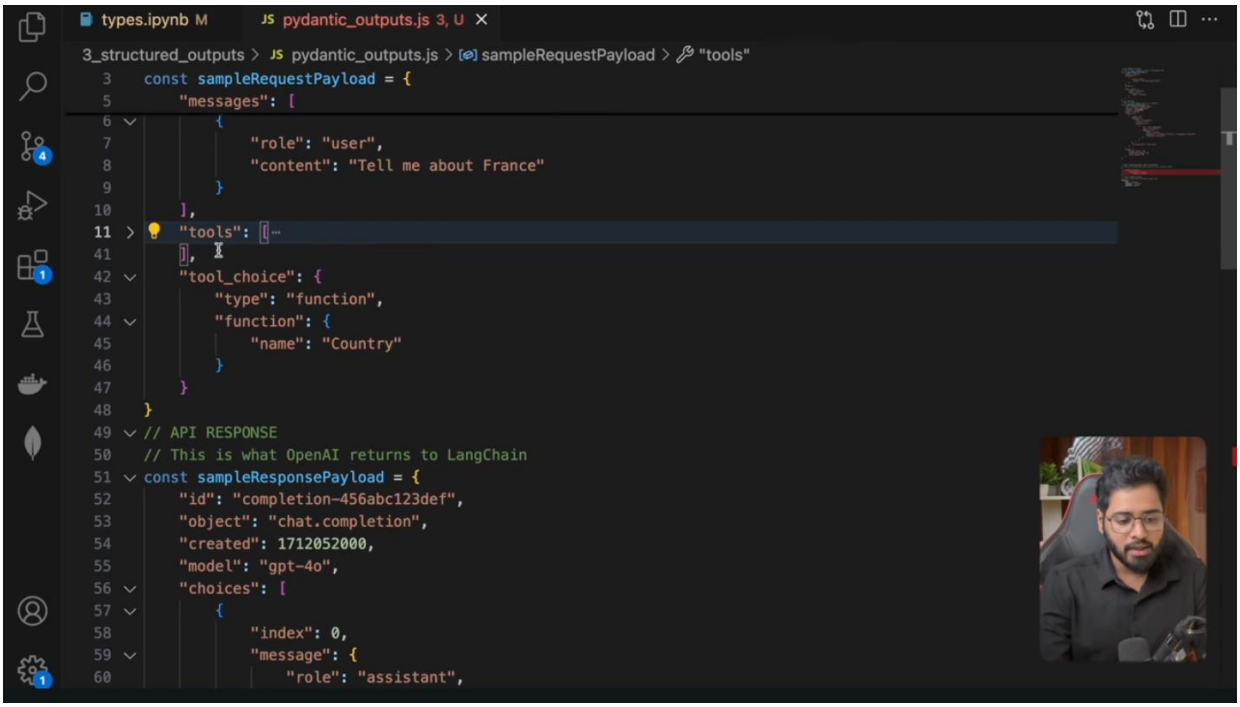
1. Pydantic is a Python library that helps define data structures
2. Acts like a "blueprint" for data
3. Uses Python's type hints (like str, int) to enforce correct data types

How it works in LangChain/LangGraph:

1. Define a class with the fields you need (name, capital, language)
2. Add descriptions to explain what each field means
3. Use with_structured_output() to tell the LLM to follow your format







The Pydantic **Country** model that we wrote will be available for the LLM to use/call when needed.

types.ipynb M pydantic_outputs.js 3, U

3_structured_outputs > pydantic_outputs.js > sampleRequestPayload

```
// API REQUEST PAYLOAD
// This is what LangChain sends to the OpenAI API
const sampleRequestPayload = {
    "model": "gpt-4o",
    "messages": [
        {
            "role": "user",
            "content": "Tell me about France"
        }
    ],
    "tools": [
        {
            "type": "function",
            "function": {
                "name": "Country",
                "description": "Information about a country",
                "parameters": {
                    "type": "object",
                    "properties": {
                        "name": {
                            "type": "string",
                            "description": "name of the country"
                        },
                        "language": {
                            "type": "string",
                            "description": "language of the country"
                        },
                        "capital": {
```

types.ipynb M pydantic_outputs.js 3, U

3_structured_outputs > pydantic_outputs.js > sampleRequestPayload > "tools" > "function" > "parameters"

```
const sampleRequestPayload = {
    ],
    "tools": [
        {
            "type": "function",
            "function": {
                "name": "Country",
                "description": "Information about a country",
                "parameters": {
                    "type": "object",
                    "properties": {
                        "name": {
                            "type": "string",
                            "description": "name of the country"
                        },
                        "language": {
                            "type": "string",
                            "description": "language of the country"
                        },
                        "capital": {
                            "type": "string",
                            "description": "Capital of the country"
                        }
                    },
                    "required": [
                        "name",
                        "language",
```

types.ipynb M pydantic_outputs.js 3, U

_structured_outputs > pydantic_outputs.js > sampleRequestPayload > "tools" > "function" > "parameters" > "properties" > "capital"

```
const sampleRequestPayload = {
    "tools": [
            "function": {
                "parameters": {
                    "type": "object",
                    "properties": {
                        "name": {
                            "type": "string",
                            "description": "name of the country"
                        },
                        "language": {
                            "type": "string",
                            "description": "language of the country"
                        },
                        "capital": {
                            "type": "string",
                            "description": "Capital of the country"
                        }
                    },
                    "required": [
                        "name",
                        "language",
                        "capital"
                    ]
                }
            }
        }
    ],
```

types.ipynb M pydantic_outputs.js 3, U

3_structured_outputs > pydantic_outputs.js > sampleRequestPayload

```
// API REQUEST PAYLOAD
// This is what LangChain sends to the OpenAI API
const sampleRequestPayload = {
    "model": "gpt-4o",
    "messages": [...
    ],
    "tools": [...
    ],
    "tool_choice": {
        "type": "function",
        "function": {
            "name": "Country"
        }
    }
}
// API RESPONSE
// This is what OpenAI returns to LangChain
const sampleResponsePayload = {
    "id": "completion-456abc123def",
    "object": "chat.completion",
    "created": 1712052000,
    "model": "gpt-4o",
    "choices": [
        {
            "index": 0,
            "message": {
                "role": "assistant",
                "content": null,
```

types.ipynb M pydantic_outputs.js 3, U

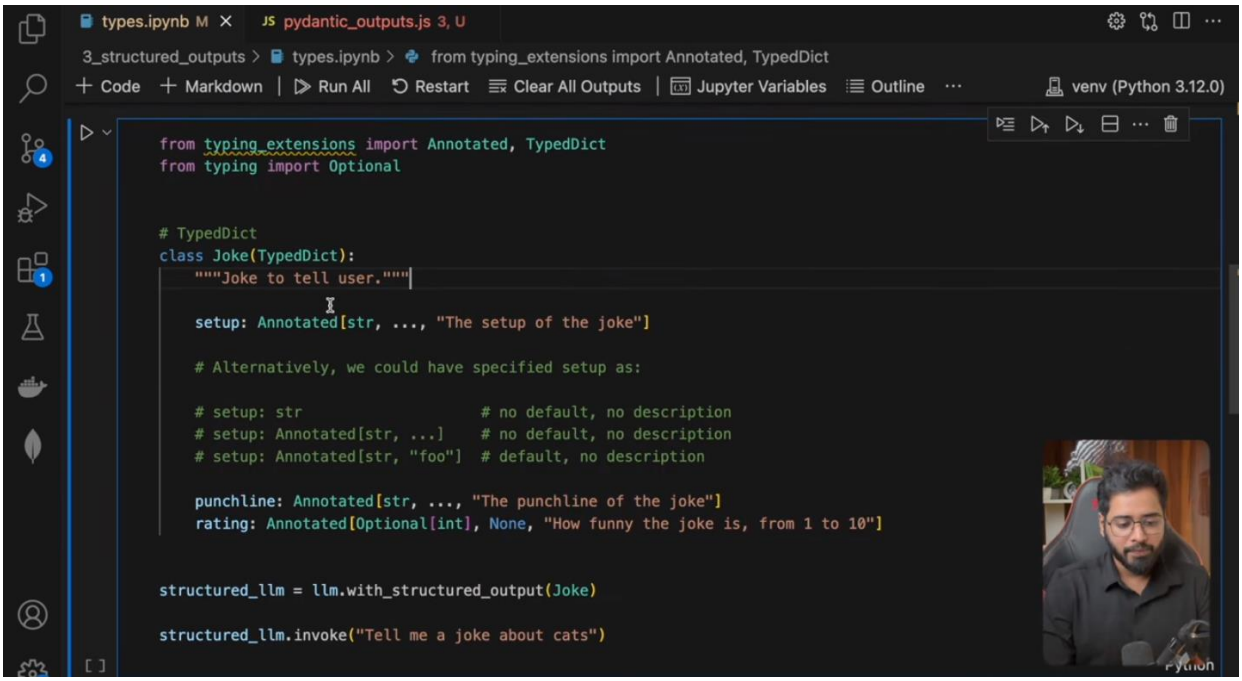
3_structured_outputs > pydantic_outputs.js > sampleRequestPayload

```
const sampleRequestPayload = {
    "tool_choice": {
        "function": {
            "name": "Country"
        }
    }
}
// API RESPONSE
// This is what OpenAI returns to LangChain
const sampleResponsePayload = {
    "id": "completion-456abc123def",
    "object": "chat.completion",
    "created": 1712052000,
    "model": "gpt-4o",
    "choices": [
        {
            "index": 0,
            "message": {
                "role": "assistant",
                "content": null,
                "tool_calls": [
                    {
                        "id": "call_789xyz456",
                        "type": "function",
                        "function": {
                            "name": "Country",
                            "arguments": "{\"name\":\"France\",\"language\":\"French\",
                            \"capital\":\"Paris\"}"
```

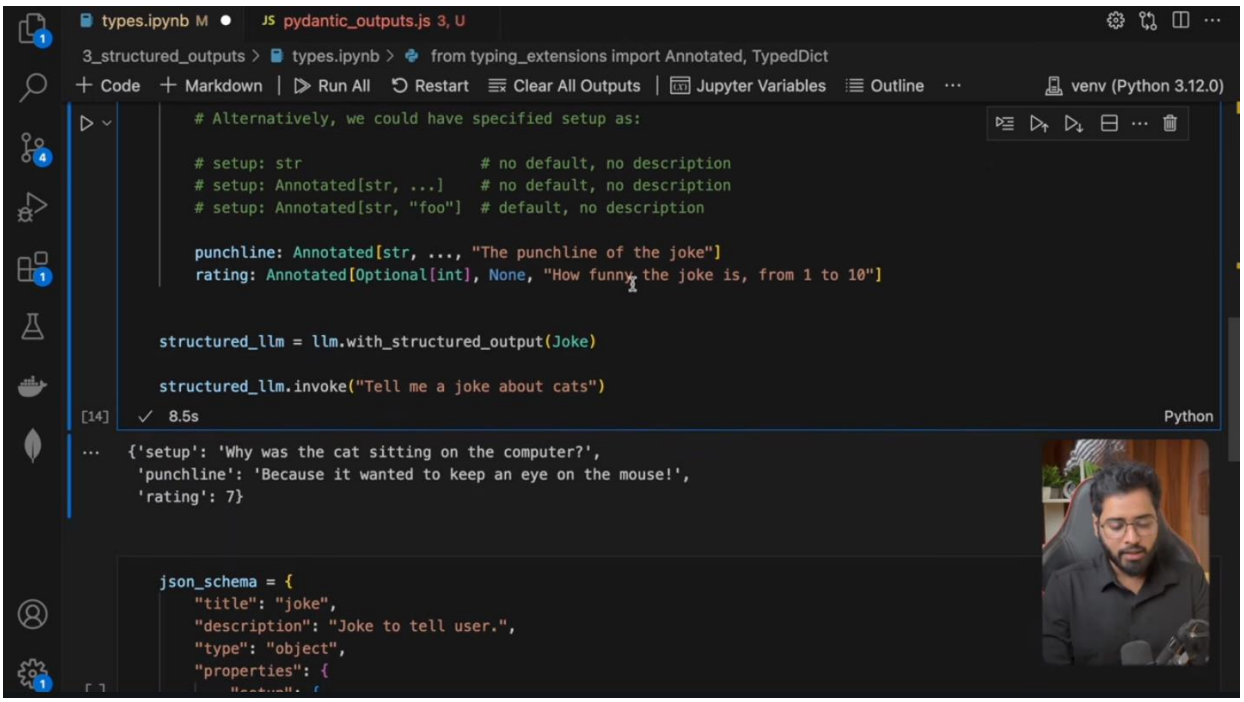
types.ipynb M pydantic_outputs.js 3, U

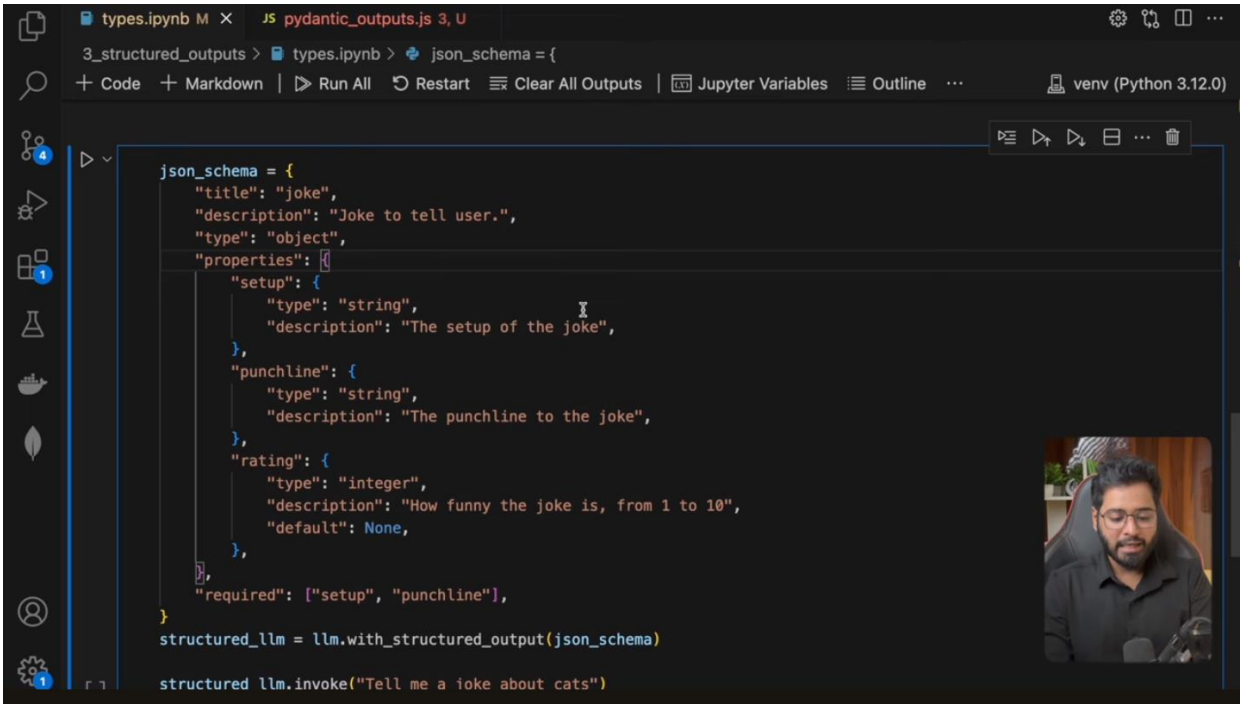
3_structured_outputs > pydantic_outputs.js > sampleResponsePayload > "choices" > "message" > "tool_calls" > "function" > "arg

```
const sampleResponsePayload = {
    "created": 1712052000,
    "model": "gpt-4o",
    "choices": [
        {
            "index": 0,
            "message": {
                "role": "assistant",
                "content": null,
                "tool_calls": [
                    {
                        "id": "call_789xyz456",
                        "type": "function",
                        "function": {
                            "name": "Country",
                            "arguments": {\"name\":\"France\",\"language\":\"French\",
                            \"capital\":\"Paris\"}"
                        }
                    }
                ]
            },
            "finish_reason": "tool_calls"
        }
    ],
    "usage": {
        "prompt_tokens": 75,
        "completion_tokens": 24,
        "total tokens": 99
```

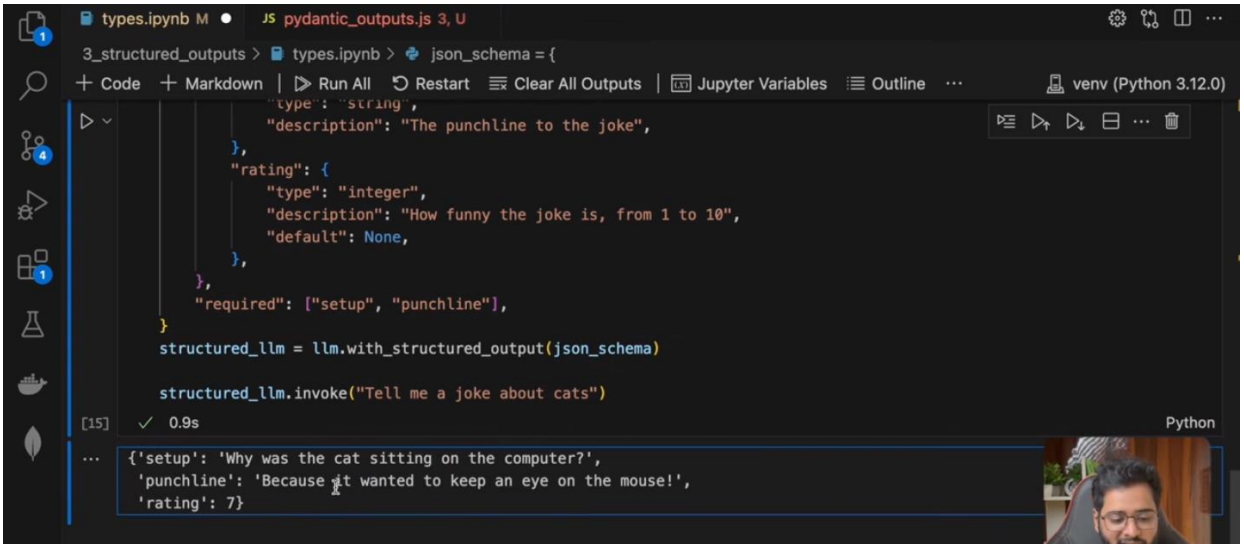


Instead of the **BaseModel** class example, the 3rd way of getting structured output from the LLM is that we can use the **TypedDict** class that offers lesser type validations.





The 3rd way of getting structured output from the LLM is by using a **json_schema** as above



We can now go ahead and get structured output from our **Reflexion agents** as below