

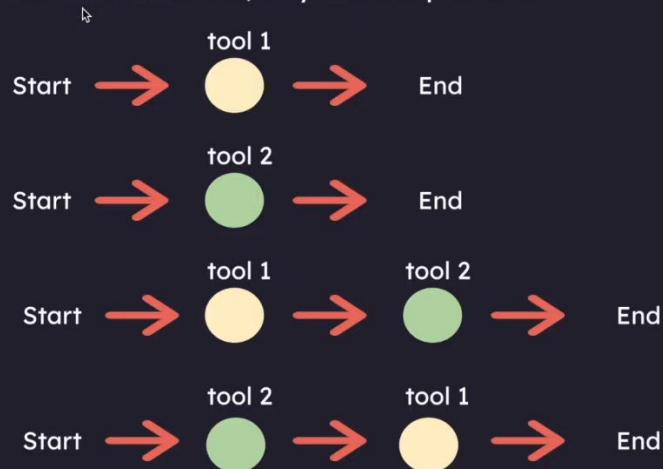
6,150 views Feb 12, 2025 #python #artificialintelligence #langgraph  
In this beginner-friendly playlist, I'll show you how to use LangGraph to build powerful AI-agents from scratch

## 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?

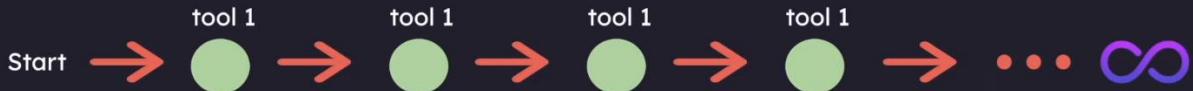
## ReAct Agents

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



## ReAct Agents

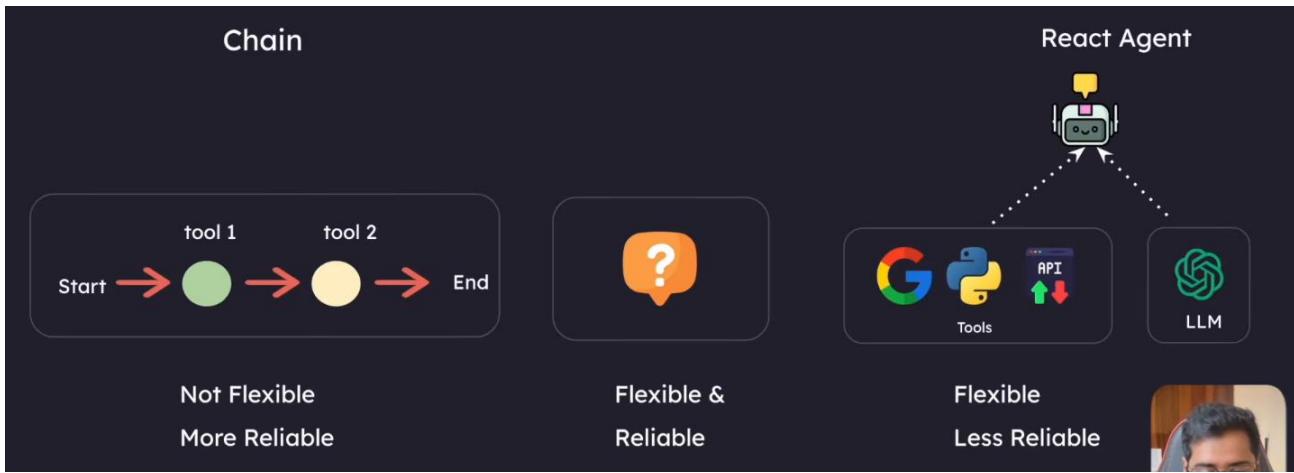
But high flexibility can also mean less reliability



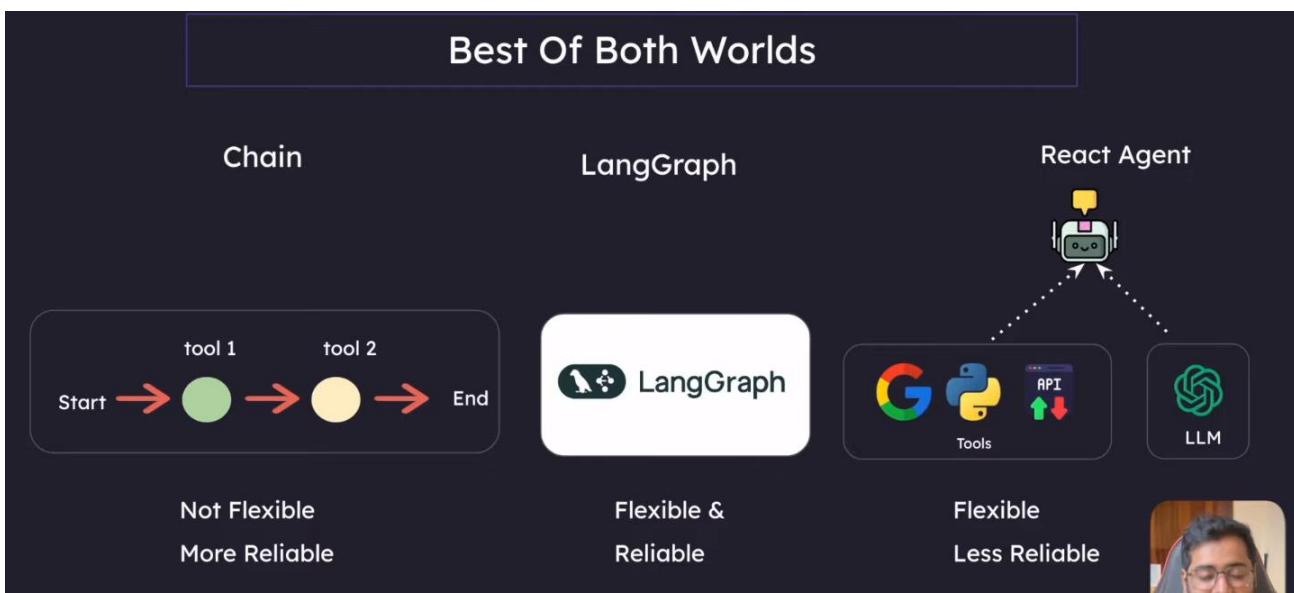
Infinite Loop causes:

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

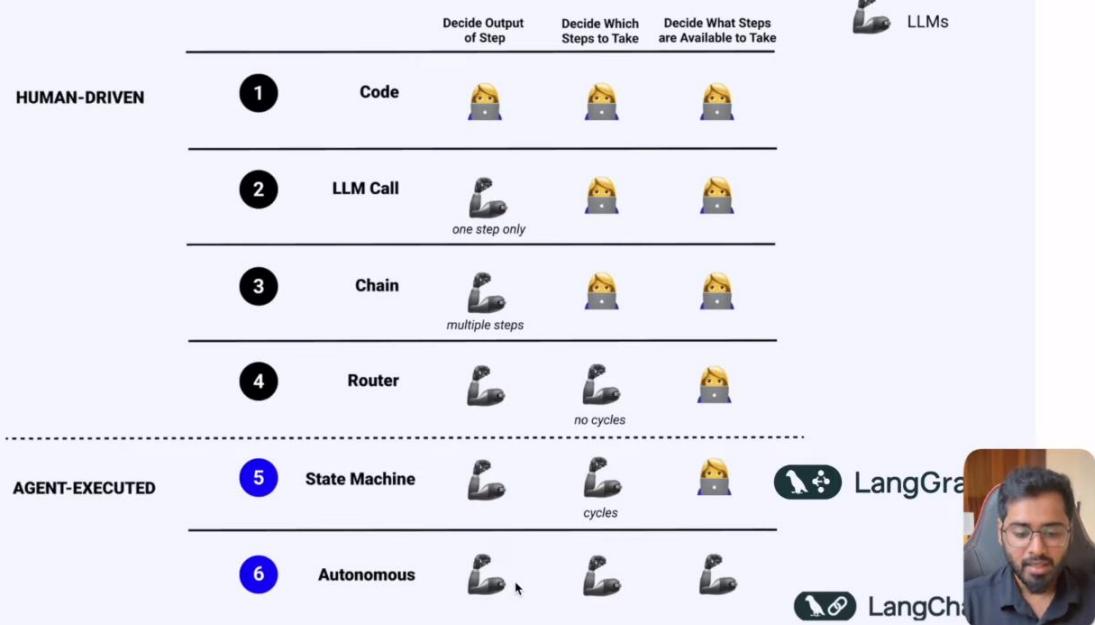




## Best Of Both Worlds



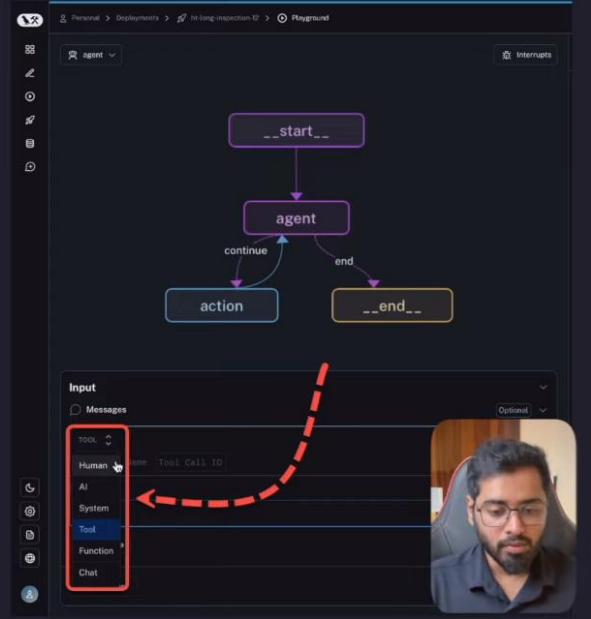
## Levels of autonomy in LLM applications



# 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

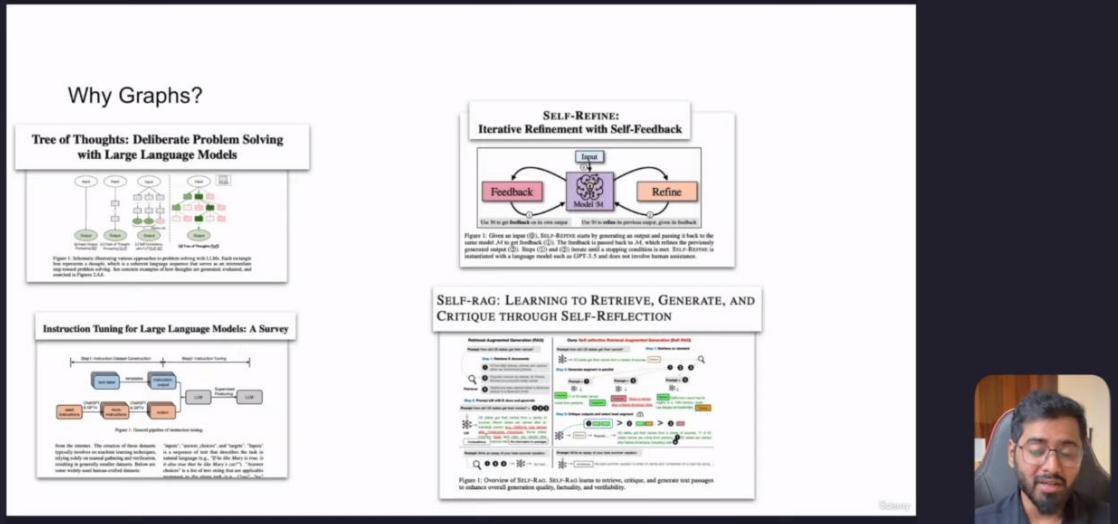
### 4. Streaming Processing:

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.

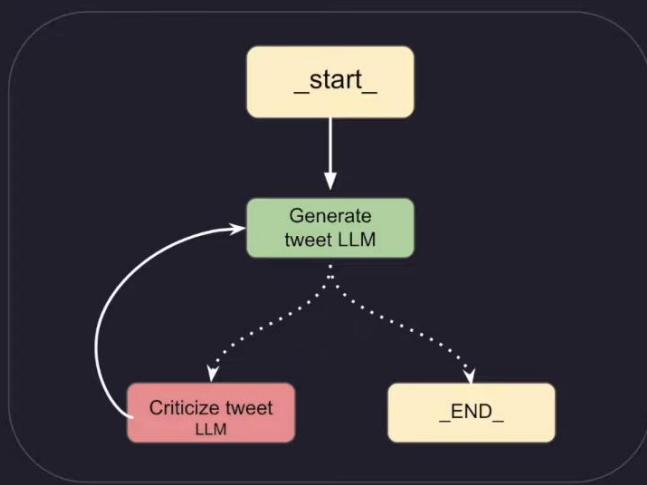
# Why use the Graph Data Structure?



## Core Components of LangGraph

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

## Example: Reflection Agent pattern





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

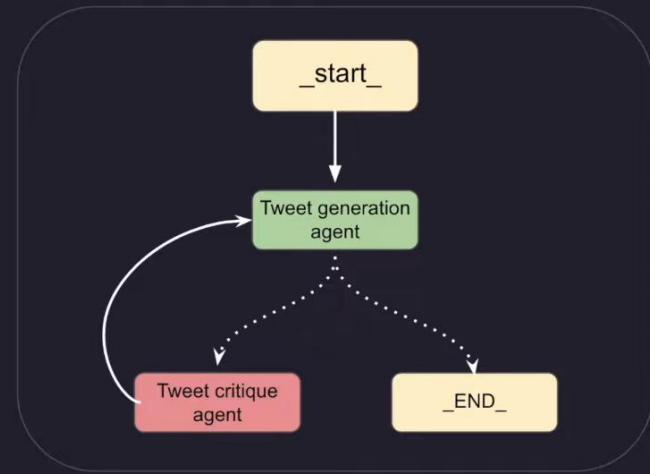
### So what is a reflection-agent pattern?

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

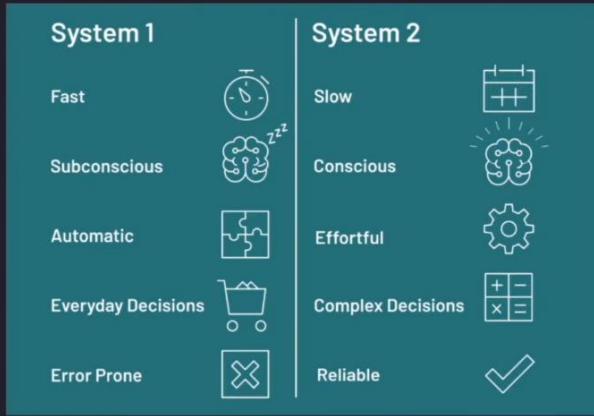
A basic reflection agent system typically consists of:

1. A generator agent
2. A reflector agent

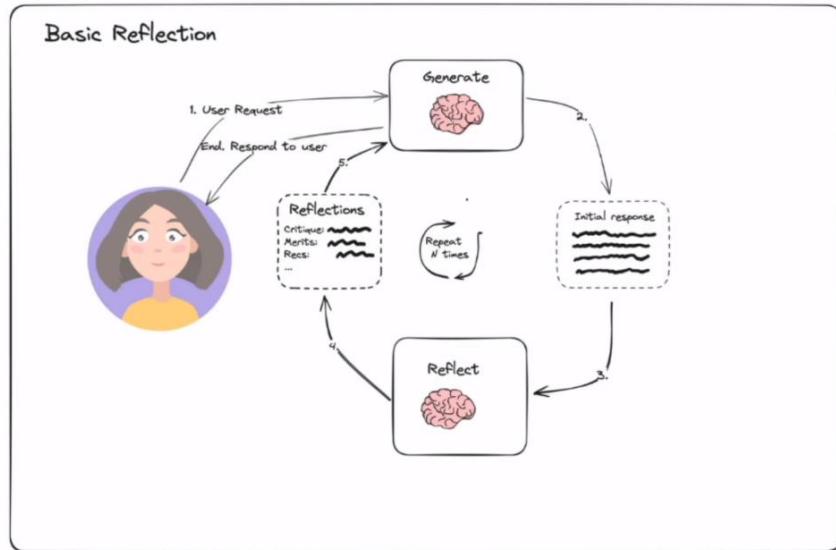
## Example: Basic Reflection Agent pattern



## Reflection Agent pattern in LangGraph



### Basic Reflection



Simple Reflection Loop



## Types of Reflection Agents in LangGraph

There are 3 types:

1. Basic Reflection Agents
2. Reflexion Agents
3. Language Agent Tree Search (LATS)

### LangGraph Crash Course #7 - Reflection Agent - Creating Chains

Harish Neel | AI  
7.67K subscribers

Subscribe

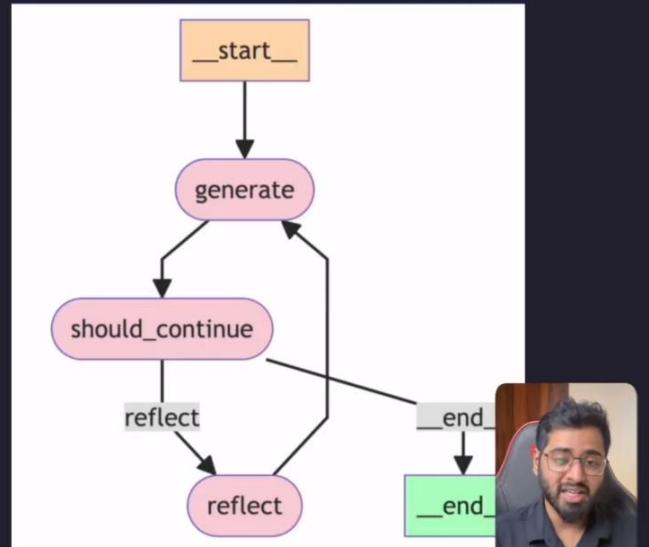
91 Share Clip Save ...

5,981 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 Implement a Basic Reflection Agent!

In this section, we'll build:

1. generation\_chain
2. reflect\_chain



The screenshot shows a code editor interface with a sidebar labeled 'EXPLORER'. In the sidebar, there are two main folders: 'LANGGRAPH' and '2\_basic\_reflection\_system'. Under '2\_basic\_reflection\_system', there is a file named 'chains.py'. The main panel displays the contents of 'chains.py':

```
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_google_genai import ChatGoogleGenerativeAI
```

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.

← → ⌛ ⌂ python.langchain.com/api\_reference/core/prompts/langchain\_core.prompts.chat.ChatPromptTemplate.html

All Bookmarks

API Reference

Search

Docs

On this page

**ChatPromptTemplate**

- input\_types
- input\_variables
- messages
- metadata
- optional\_variables
- output\_parser
- partial\_variables
- tags
- validate\_template
- abatch()
- abatch\_as\_completed()
- aformat()
- aformat\_messages()
- aformat\_prompt()
- ainvoke()
- append()
- astream()
- astream\_as\_completed()
- batch()
- batch\_as\_completed()
- bind()



Messages Placeholder:

```
# In addition to Human/AI/Tool/Function messages,
# you can initialize the template with a MessagesPlaceholder
# either using the class directly or with the shorthand tuple syntax:
template = ChatPromptTemplate([
    ("system", "You are a helpful AI bot. Your name is {name}."),
    ("human", "Hello, how are you doing?"),
    ("ai", "I'm doing well, thanks!"),
    ("human", "{user_input}"),
])

prompt_value = template.invoke(
    {
        "name": "Bob",
        "user_input": "What is your name?"
    }
)
```

LANGGRAPH

- 1\_Introduction
- react\_agent\_basic.py
- 2\_basic\_reflection\_system
- chains.py
- venv
- .env

OUTLINE

```
2_basic_reflection_system > chains.py > ...
1 from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
2 from langchain_google_genai import ChatGoogleGenerativeAI
3
4 generation_prompt = ChatPromptTemplate.from_messages(
5     [
6         (
7             "system",
8             "You are a twitter techie influencer assistant tasked with writing excellent twitter
9             posts."
10            " Generate the best twitter post possible for the user's request."
11            " If the user provides critique, respond with a revised version of your previous
12            attempts.",
13        ),
14        MessagesPlaceholder(variable_name="messages"),
15    ]
16)
17
18 reflection_prompt = ChatPromptTemplate.from_messages(
19     [
20         (
21             "system",
22             "You are a viral twitter influencer grading a tweet. Generate critique and
23             recommendations for the user's tweet."
24             "Always provide detailed recommendations, including requests for length, virality,
25             style, etc.",
26        ),
27        MessagesPlaceholder(variable_name="messages"),
28    ]
29)
30 llm = ChatGoogleGenerativeAI(model="gemini-1.5-pro")
31
32 generation_chain = generation_prompt | llm
33 reflection_chain = reflection_prompt | llm
34
```



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

### LangGraph Crash Course #8 - Reflection Agent - Building The Graph



Harish Neel | AI  
7.67K subscribers

Subscribe

133

Dislike

Share

Clip

Save

...

6,869 views Feb 13, 2025 #python #artificialintelligence #langgraph

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

```
# pip install langgraph
```

The screenshot shows a code editor interface with a sidebar containing project files:

- LANGGRAPH**
  - 1\_Introduction
  - react\_agent\_basic.py
  - 2\_basic\_reflection\_system
  - basic.py (selected)
  - chains.py
  - venv
  - .env

The main pane displays the content of `basic.py`:

```

1 from typing import List, Sequence
2 from dotenv import load_dotenv
3 from langchain_core.messages import BaseMessage, HumanMessage
4 from langgraph.graph import END, MessageGraph
5 from chains import generation_chain, reflection_chain
6
7 load_dotenv()

```

## Basic Reflection Agent!

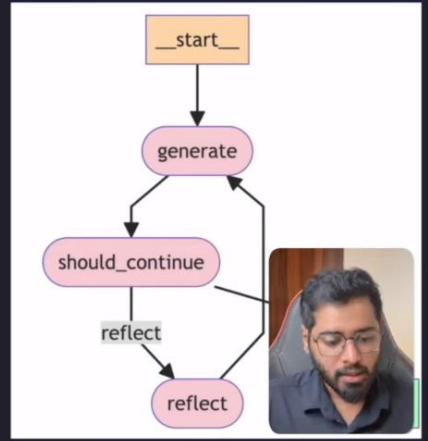
### What is a MessageGraph?

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!

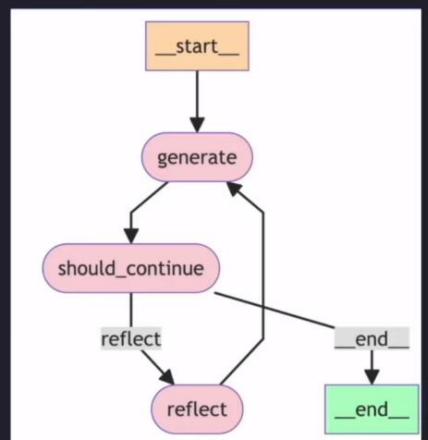
### What is a MessageGraph?

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





EXPLORER

LANGGRAPH

- 1\_Introduction
- react\_agent\_basic.py
- 2\_basic\_reflection\_system
- basic.py
- chains.py

basic.py

```
2_basic_reflection_system > basic.py > reflect_node
1   from typing import List, Sequence
2   from dotenv import load_dotenv
3   from langchain_core.messages import BaseMessage, HumanMessage
4   from langgraph.graph import END, MessageGraph
5   from chains import generation_chain, reflection_chain
6
7   load_dotenv()
8
9   graph = MessageGraph()
10
11  REFLECT = "reflect"
12  GENERATE = "generate"
13
14  def generate_node(state):
15      return generation_chain.invoke({
16          "messages": state
17      })
18
19  def reflect_node(state):
20      return reflection_chain.invoke([
21          "messages": state
22      ])
```

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



EXPLORER

LANGGRAPH

- 1\_Introduction
- react\_agent\_basic.py
- 2\_basic\_reflection\_system
- basic.py
- chains.py

basic.py

```
2_basic_reflection_system > basic.py > ...
13
14  def generate_node(state):
15      return generation_chain.invoke({
16          "messages": state
17      })
18
19  def reflect_node(state):
20      response = reflection_chain.invoke({
21          "messages": state
22      })
23      return [HumanMessage(content=response.content)]
24
25
26  graph.add_node(GENERATE, generate_node)
27  graph.add_node(REFLECT, reflect_node)
28
29  graph.set_entry_point(GENERATE)
30
31  def should_continue(state):
32      if(len(state) > 4):
33          return END
34      return REFLECT
35
```

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

# Basic Reflection Agent!

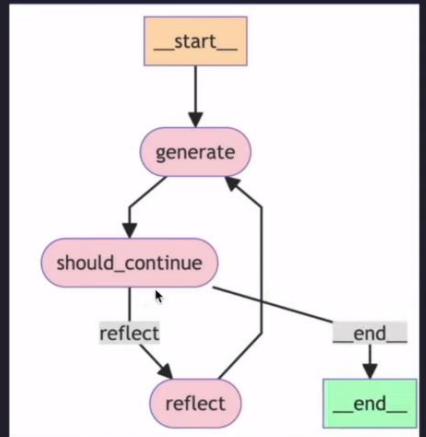
## What is a MessageGraph?

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

```
2_basic_reflection_system > basic.py > ...
31 def should_continue(state):
32     if state == 'END':
33         return END
34     return REFLECT
35
36
37 graph.add_conditional_edges(GENERATE, should_continue)
38 graph.add_edge(REFLECT, GENERATE)
39
40 app = graph.compile()
41
42 print(app.get_graph().draw_mermaid())
43 app.get_graph().print_ascii()
44
45 # pip install grandalf
```

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

```
/Users/harishb/Desktop/LangGraph/venv/bin/python /Users/harishb/Desktop/LangGraph/2_basic_reflection_system/basic.py
(venv) harishb@Harishb-MacBook-Air LangGraph % /Users/harishb/Desktop/LangGraph/venv/bin/python /Users/harishb/Desktop/LangGraph/2_basic_reflection_system/basic.py
%%init: {'flowchart': {'curve': 'linear'}}}%%
graph TD;
    _start_([<p>_start_</p>]):::first
    generate((generate))
    reflect((reflect))
    _end_([<p>_end_</p>]):::last
    _start_ --> generate;
    reflect --> generate;
    generate --> reflect;
    generate --> _end_;
    classDef default fill:#f2f0ff,line-height:1.2
    classDef first fill-opacity:0
    classDef last fill:#fbfbfc
```

The screenshot shows a dark-themed IDE interface. On the left is a sidebar with icons for search, file navigation, and project structure. The main area has tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and PORTS. The TERMINAL tab is active, displaying a Mermaid diagram and some log messages. The right side features a video call window with a person's face.

```

38     graph.add_edge(REFLECT, GENERATE)
39
40     app = graph.compile()
41
42     print(app.get_graph().draw_mermaid())
43     app.get_graph().print_ascii()
44
classDef default fill:#f2f0ff,line-height:1.2
classDef first fill-opacity:0
classDef last fill:#fbfbfc

+--+
| __start__ |
+--+
|   *   |
|   *   |
|   *   |
+--+
| generate |
+--+
| ***   ... |
|   *   .   |
|   **  .. |
+--+
| reflect | +--+
|           | | __end__ |
+--+

```

WARNING: All log messages before absl::InitializeLog() is called are written to STDERR  
E0000 00:00:1739431371.440952 134606 init.cc:232] grpc\_wait\_for\_shutdown\_wit

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

The screenshot shows a dark-themed IDE interface. On the left is a sidebar with icons for search, file navigation, and project structure. The main area has tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and PORTS. The TERMINAL tab is active, displaying a Mermaid diagram and some log messages. A tooltip is visible over the 'content' parameter in the code, providing documentation: 'The string contents of the message.' Below the code editor, there is a preview of the generated Mermaid diagram.

```

31     def should_continue(state: str | list[str | dict], **kwargs: Any) ->
32         if (len(state)) > 2
33             return END
34         return REFLECT
35
36
37     graph.add_conditional_edge(state, REFLECT)
38
39     app = graph.compile()
40
41     print(app.get_graph().draw_mermaid())
42     app.get_graph().print_ascii()
43
44
45     response = app.invoke(HumanMessage(content="AI Agents taking over content creation"))
46
47     print(response)
48

```

**content**  
The string contents of the message.

**Args**

- content**  
The string contents of the message.
- kwargs**  
Additional keyword arguments to pass to the message.

EXPLORER

- LANGGRAPH
  - 1\_Introduction
  - react\_agent\_basic.py
  - 2\_basic\_reflection\_system
    - > \_\_pycache\_\_
    - basic.py
    - chains.py
    - > venv
    - .env

**basic.py**

```

2_basic_reflection_system > basic.py > ...
38     graph.add_edge(REFLECT, GENERATE)
39
40     app = graph.compile()
41
42     print(app.get_graph().draw('graph TD'))

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

graph TD
    start --> generate
    generate --> reflect
    reflect --> end
    generate --> generate

```

TERMINAL

zsh Python

OUTLINE TIMELINE

EXPLORER

- LANGGRAPH
  - 1\_Introduction
  - react\_agent\_basic.py
  - 2\_basic\_reflection\_system
    - > \_\_pycache\_\_
    - basic.py
    - chains.py
    - > venv
    - .env

**basic.py**

```

2_basic_reflection_system > basic.py > ...
38     graph.add_edge(REFLECT, GENERATE)
39
40     app = graph.compile()
41
42     print(app.get_graph().draw('graph TD'))

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

graph TD
    reflect --> end

```

TERMINAL

zsh Python

OUTLINE TIMELINE

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

```
basic.py
26 graph.add_node(GENERATE, generate_node)
27 graph.add_node(REFLECT, reflect_node)
28 graph.set_entry_point(GENERATE)
29
30
31 def should_continue(state):
32     if (len(state) > 6):
33         return END
34     return REFLECT
35
36
37 graph.add_conditional_edges(GENERATE, should_continue)
```

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

```
chains.py
1 from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
2 from langchain_google_genai import ChatGoogleGenerativeAI
3 from langchain_openai import ChatOpenAI
4
5
6 llm = ChatOpenAI(model="gpt-4o")
7
8 generation_chain = generation_prompt | llm
9 reflection_chain = reflection_prompt | llm
10
```

```
basic.py
26 graph.add_node(GENERATE, generate_node)
27 graph.add_node(REFLECT, reflect_node)
28 graph.set_entry_point(GENERATE)
29
30
31 def should_continue(state):
32     if (len(state) > 6):
33         return END
34     return REFLECT
35
36
37 graph.add_conditional_edges(GENERATE, should_continue)
38 graph.add_edge(REFLECT, GENERATE)
39
40 app = graph.compile()
41
42 print(app.get_graph().draw_mermaid())
43 app.get_graph().print_ascii()
44
```

TERMINAL

```
generate -.--> __end__;
classDef default fill:#f2f0ff,line-height:1.2
classDef first fill=opacity:0
classDef last fill:#fbfb6fc

+-----+
| __start__ |
+-----+
*
*
*
+-----+
| generate |
+-----+
***      ...
*          ..
**        __end__ |
```

zsh

Python

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS + ⌂ zsh Python

```
| reflect |      | __end__ |
+-----+      +-----+
[HumanMessage(content='AI Agents taking over content creation', additional_kw
args={}, response_metadata={}, id='df972ab4-ffc0-4f6f-919c-87ab967b946d'), AI
Message(content=' Excited or concerned? 😊 AI Agents are revolutionizing co
ntent creation with speed, efficiency, and creativity! From generating eye-ca
tching designs to crafting compelling stories, the possibilities are endless.
★ How do you think this will impact the future of storytelling and artistry
? Let's discuss! #AI #ContentCreation #Innovation ❤️", additional_kwarg
s={'refusal': None}, response_metadata={'token_usage': {'completion_tokens': 69,
'prompt_tokens': 59, 'total_tokens': 128}, 'completion_tokens_details': {'accep
ted_prediction_tokens': 0, 'audio_tokens': 0, 'reasoning_tokens': 0, 'rejecte
d_prediction_tokens': 0}, 'prompt_tokens_details': {'audio_tokens': 0, 'cache
d_tokens': 0}, 'model_name': 'gpt-4o-2024-08-06', 'system_fingerprint': 'fp_
50cad350e4', 'finish_reason': 'stop', 'logprobs': None}, id='run-c244c322-20a
d-4b4e-817d-426842d4c353-0', usage_metadata={'input_tokens': 59, 'output_toke
ns': 69, 'total_tokens': 128, 'input_token_details': {'audio': 0, 'cache_read
': 0}, 'output_token_details': {'audio': 0, 'reasoning': 0}}}, HumanMessage(c
ontent='Your tweet touches on an intriguing and current topic, but it could b
e benefici
```



PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS + ⌂ zsh Python

```
| reflect |      | __end__ |
+-----+      +-----+
[HumanMessage(content='AI Agents taking over content creation', additional_kw
args={}, response_metadata={}, id='df972ab4-ffc0-4f6f-919c-87ab967b946d'), AI
Message(content=' Excited or concerned? 😊 AI Agents are revolutionizing co
ntent creation with speed, efficiency, and creativity! From generating eye-ca
tching designs to crafting compelling stories, the possibilities are endless.
★ How do you think this will impact the future of storytelling and artistry
? Let's discuss! #AI #ContentCreation #Innovation ❤️", additional_kwarg
s={'refusal': None}, response_metadata={'token_usage': {'completion_tokens': 69,
'prompt_tokens': 59, 'total_tokens': 128}, 'completion_tokens_details': {'accep
ted_prediction_tokens': 0, 'audio_tokens': 0, 'reasoning_tokens': 0, 'rejecte
d_prediction_tokens': 0}, 'prompt_tokens_details': {'audio_tokens': 0, 'cache
d_tokens': 0}, 'model_name': 'gpt-4o-2024-08-06', 'system_fingerprint': 'fp_
50cad350e4', 'finish_reason': 'stop', 'logprobs': None}, id='run-c244c322-20a
d-4b4e-817d-426842d4c353-0', usage_metadata={'input_tokens': 59, 'output_toke
ns': 69, 'total_tokens': 128, 'input_token_details': {'audio': 0, 'cache_read
': 0}, 'output_token_details': {'audio': 0, 'reasoning': 0}}}, HumanMessage(c
ontent='Your tweet touches on an intriguing and current topic, but it could b
e benefici
```



EXPLORER LANGGRAPH ... basic.py x chains.py

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()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS + ⌂ zsh Python

```
be! Are AIs the co-pilots or should they take the driver's seat? 🚗 Retweet
to join the conversation! #AIRevolution #FutureOfWork #ArtAndTech #ContentCr
eation #AIFuture 🚗, additional_kwarg
s={}, response_metadata={}, id='197c43
fb-418d-4315-9ade-1fdcad2c736c', AIMessage(content=' AI Agents are redefin
ing creativity across industries! From AI-generated screenplays gaining Oscar
buzz to breathtaking digital artworks, their influence is reshaping our crea
tive narrative. ★ Are these tools allies to human creativity or rivals to ar
tistry? 😊 Where do you stand? Comment below! Should AI be the co-pilot or th
e driver in the creative process? 🚗 Retweet to ignite the conversation! #AIR
evolution #FutureOfWork #ArtAndTech #ContentCreation #AIFuture 🚗, addition
al_kwarg
s={'refusal': None}, response_metadata={'token_usage': {'completion_t
okens': 107, 'prompt_tokens': 1227, 'total_tokens': 1334}, 'completion_tokens_
details': {'accepted_prediction_tokens': 0, 'audio_tokens': 0, 'reasoning_tok
ens': 0, 'rejected_prediction_tokens': 0}, 'prompt_tokens_details': {'audio_t
okens': 0, 'cached_tokens': 0}, 'model_name': 'gpt-4o-2024-08-06', 'system_f
ingerprint': 'fp_50cad350e4', 'finish_reason': 'stop', 'logprobs': None}, id=
'run-39a3d862-10f1-4ecf-9582-be19cb55fb4f-0', usage_metadata={'input_tokens':
1227, 'output_tokens': 107, 'total_tokens': 1334, 'input_token_details': {'a
udio': 0, 'cache_read': 0}, 'output_token_details': {'audio': 0, 'reasoning':
0}})]
```



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

## LangGraph Crash Course #9 - Reflection Agent - LangSmith Tracing



Harish Neel | AI  
7.68K subscribers

Subscribe

97 | Share | Clip | Save | ...

4,248 views Feb 13, 2025 #python #artificialintelligence #langgraph

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

The screenshot shows the LangSmith Tracing interface. At the top, there's a navigation bar with icons for back, forward, search, and refresh, followed by the URL 'smith.langchain.com/o/e4a9f3da-d95f-42b8-be4e-c631ca42cc3b?paginationState=%7B"pageIndex"%3A0%2C"pageSize"%3A5%7D'. To the right are star, folder, and other bookmarking options. Below the URL is a sidebar with icons for home, personal, and developer mode. The main area is titled 'Personal' with an 'ID' link. It has a 'Get Started' section with three buttons: 'Set up tracing', 'Run an evaluation', and 'Try out playground'. Under 'Observability', there's a table showing project metrics for 'pr-bumpy-beauty-33': Name (pr-bumpy-beauty-33), Feedback (7D) (7), Run Count (7D) (29%), Error Rate (7D) (11.88s), P50 Latency (7D) (16.76s), and P99 Latency (7D). A note below says 'Showing 5 most active projects from the past 7 days. Tab shows number of total projects.' On the right, there's a small video thumbnail of a person.

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](https://smith.langchain.com)

The screenshot shows the LangSmith Tracing interface. At the top, there's a navigation bar with icons for back, forward, search, and refresh, followed by the URL 'smith.langchain.com/o/e4a9f3da-d95f-42b8-be4e-c631ca42cc3b/projects?paginationState=%7B"pageIndex"%3A0%2C"pageSize"%3A10%7D'. To the right are star, folder, and other bookmarking options. Below the URL is a sidebar with icons for home, personal, and developer mode. The main area shows a list of steps: 1. Generate API Key (button), 2. Install dependencies (Python/TypeScript tabs, code: 'pip install -U langchain langchain-openai'), 3. Configure environment to connect to LangSmith (Project Name: 'pr-elderly-excuse-27', code: 'LANGSMITH\_TRACING=true', 'LANGSMITH\_ENDPOINT="https://api.smith.langchain.com"', 'LANGSMITH\_API\_KEY="<your-api-key>"', 'LANGSMITH\_PROJECT="pr-elderly-excuse-27"', 'OPENAI\_API\_KEY="<your-openai-api-key>"'), and 4. Run any LLM, Chat model, or Chain. Its trace will be sent to this project. (code: 'from langchain\_openai import ChatOpenAI', 'llm = ChatOpenAI()', 'llm.invoke("Hello, world!")'). On the right, there's a small video thumbnail of a person.

smith.langchain.com/o/e4a9f3da-d95f-42b8-be4e-c631ca42cc3b/projects?paginationState=%7B"pageIndex":%3A0%2C"pageSize":%3A10%7D

1. New Key Created.  
lsv2\_pt\_3679393279b5433c9af9808eadde52fa\_e755b7548d

2. Install dependencies  
1 pip install -U langchain langchain-openai 

3. Configure environment to connect to LangSmith.

Project Name  
pr-elderly-excuse-27

```
1 LANGSMITH_TRACING=true
2 LANGSMITH_ENDPOINT="https://api.smith.langchain.com"
3 LANGSMITH_API_KEY="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"
4 LANGSMITH_PROJECT="pr-elderly-excuse-27"
5 OPENAI_API_KEY=<your-openai-api-key>
```

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

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



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

```
.env
1 GOOGLE_API_KEY=AIZaSyAG1YLD2BX-SB4vfNhNk5nb0oSm_nbwXw
2 TAVILY_API_KEY=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
3 OPENAI_API_KEY="sk-proj-tPorMyxanGyUtkLv1QCYV4YaERduwafaBMNb9s4IIjZ07gGn5Th
5_bnqm530MKepylQM3JxRsoT3BlbkFJEHRfsbpGqD040VaNdbFuNQWQPn_tezJ26Wsue2j2fskz
Rnbf5zGJP8sSfEwJHSWUF9f9rwofYA"
4 LANGSMITH_TRACING=true
5 LANGSMITH_ENDPOINT="https://api.smith.langchain.com"
6 LANGSMITH_API_KEY="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"
7 LANGSMITH_PROJECT="pr-elderly-excuse-27"
```

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 ...
1 from typing import List, Sequence
2 from dotenv import load_dotenv
3 from langchain_core.messages import BaseMessage, HumanMessage
4 from langgraph.graph import END, MessageGraph
5 from chains import generation_chain, reflection_chain
6
7 load_dotenv()
8
9 REFLECT = "reflect"
10 GENERATE = "generate"
11 graph = MessageGraph()
12
13 def generate_node(state):
14     return generation_chain.invoke({
15         "messages": state
16     })
17
18 def reflect_node(messages):
19     response = reflection_chain.invoke({
20         "messages": messages
21     })
22     return [HumanMessage(content=response.content)]
```



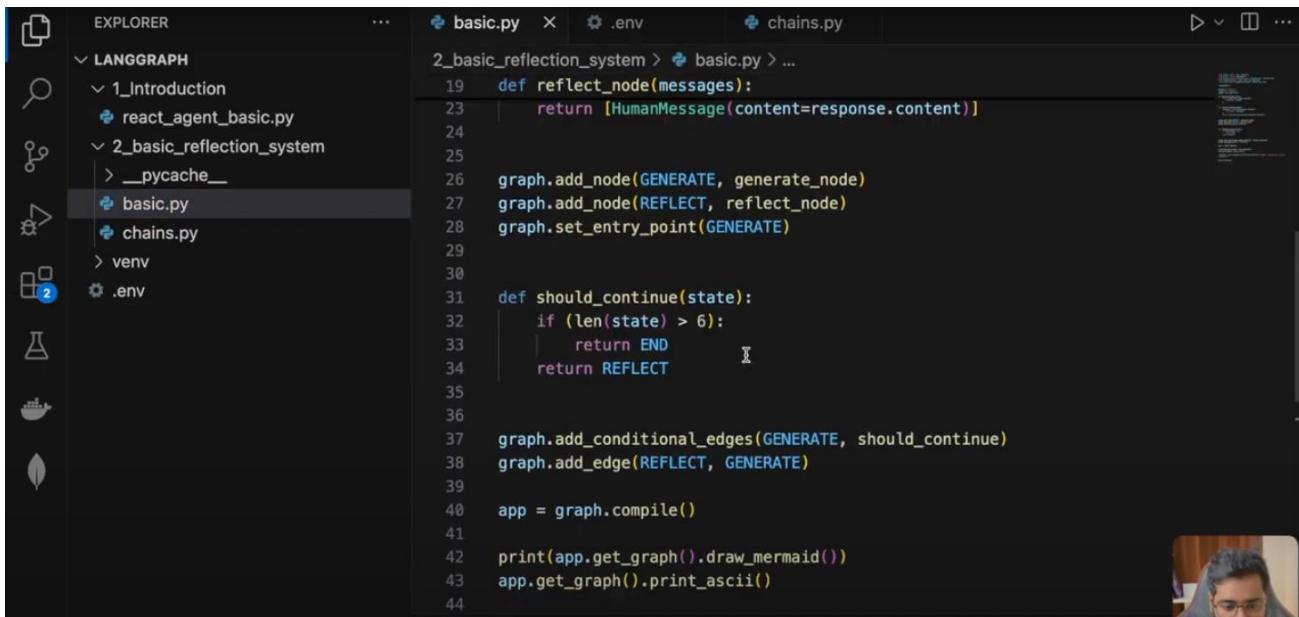
EXPLORER

LANGGRAPH

- 1\_Introduction
  - react\_agent\_basic.py
- 2\_basic\_reflection\_system
  - basic.py
  - chains.py
- venv
- .env

basic.py    .env    chains.py

```
2_basic_reflection_system > basic.py > ...
19 def reflect_node(messages):
20     return [HumanMessage(content=response.content)]
21
22 graph.add_node(GENERATE, generate_node)
23 graph.add_node(REFLECT, reflect_node)
24 graph.set_entry_point(GENERATE)
25
26 def should_continue(state):
27     if (len(state) > 6):
28         return END
29     return REFLECT
30
31 graph.add_conditional_edges(GENERATE, should_continue)
32 graph.add_edge(REFLECT, GENERATE)
33
34 app = graph.compile()
35
36 print(app.get_graph().draw_mermaid())
37 app.get_graph().print_ascii()
38
39
40
41
42
43
44
```



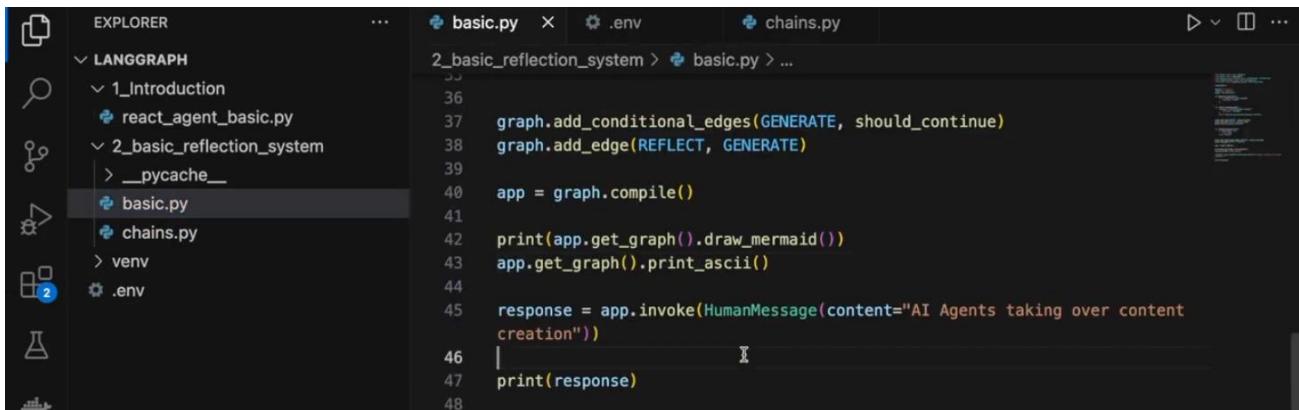
EXPLORER

LANGGRAPH

- 1\_Introduction
  - react\_agent\_basic.py
- 2\_basic\_reflection\_system
  - basic.py
  - chains.py
- venv
- .env

basic.py    .env    chains.py

```
2_basic_reflection_system > basic.py > ...
35
36 graph.add_conditional_edges(GENERATE, should_continue)
37 graph.add_edge(REFLECT, GENERATE)
38
39 app = graph.compile()
40
41 print(app.get_graph().draw_mermaid())
42 app.get_graph().print_ascii()
43
44 response = app.invoke(HumanMessage(content="AI Agents taking over content creation"))
45
46
47 print(response)
48
```



EXPLORER

LANGGRAPH

- 1\_Introduction
  - react\_agent\_basic.py
- 2\_basic\_reflection\_system
  - basic.py
  - chains.py
- venv
- .env

basic.py    .env    chains.py

```
2_basic_reflection_system > basic.py > ...
7
8
9
10
11
12
13
14
15
16
```

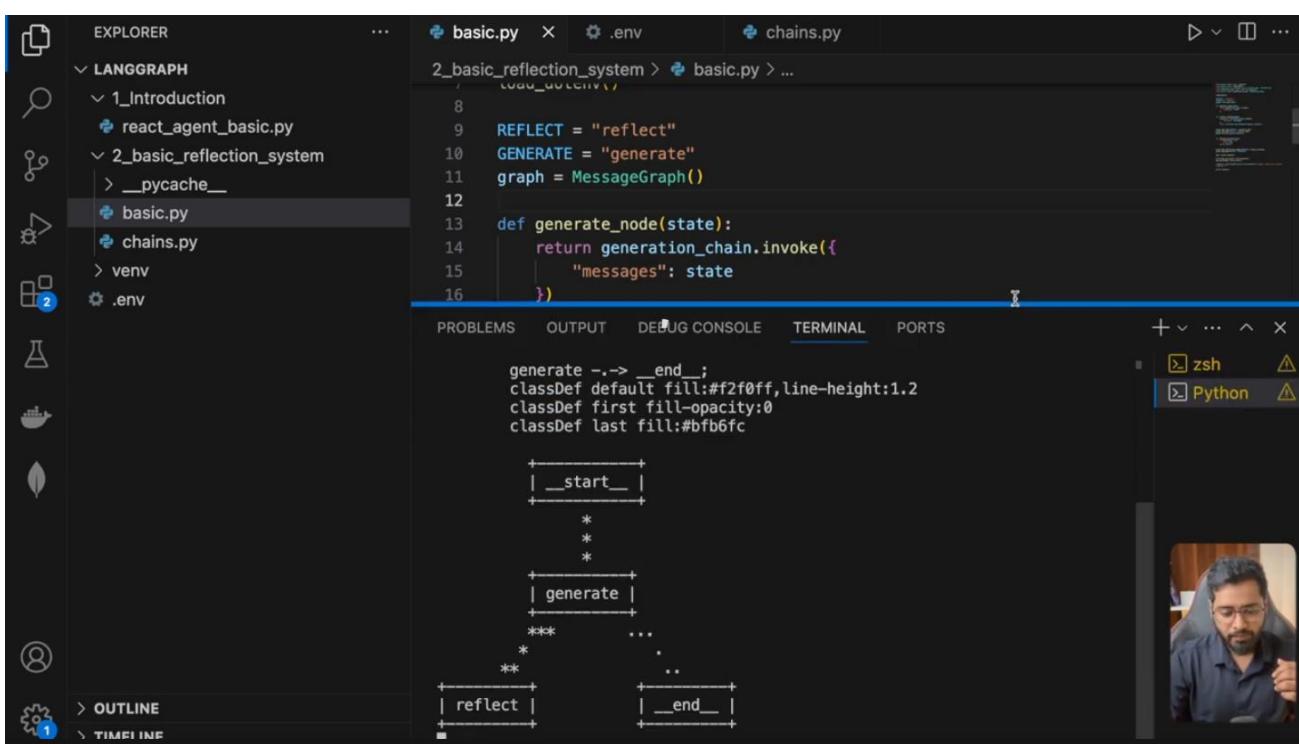
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

TERMINAL

```
generate --> __end__;
classDef default fill:#f2f0ff,line-height:1.2
classDef first fill-opacity:0
classDef last fill:#bfbbfc

+---+
| __start__ |
+---+
*
*
*
+---+
| generate |
+---+
***   ...
**    ..
+---+ +---+
| reflect | | __end__ |
+---+ +---+
```

zsh    Python



PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
r just supporting roles; they're now crafting breathtaking art, writing novels, 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! Consider refining it or making it more engaging: "Are AI-powered creations as meaningful as human-made ones? Let's debate!"\n\n3. **Utilize Emojis:** You're using symbols effectively to capture attention. Ensure to maintain a balance so the tweet doesn't feel cluttered. Highlight key phrases like "lightning-fast innovation" and "artistic collaboration".\n\n4. **Hashtag Strategy:** You're using some effective hashtags already, but consider adding a few more that target specific communities or interests, such as #AIArt, #DigitalAge, or #TechRevolution for more specialized reach.\n\n5. **Create a Thread:** If you have additional points or examples, consider making a thread. This can lead to more engagement and shared information without overwhelming a single 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 surprising fact or statistic about AI in content creation. For instance, "Did you know AI can create music tracks indistinguishable from human composers in just minutes?"\n\n8. **Personal Touch:** Share a personal insight or experience with AI, if possible. This can relate your audience more closely to the topic.
```

< > ⌂ 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

+ New Project

### Tracing projects

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-elder...	1	0%	46.82s	46.82s	0%	7,179	\$0.031	⋮
pr-burn...	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

+ New Project

### Tracing projects

Search by name... Columns

Run 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 taking over the world.	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%

smith.langchain.com/o/e4a9f3da-d95f-42b8-be4e-c631ca42cc3b/projects/p/8101fd03-6f4d-494f-af92-acb651c8fb3a?timeModel=%7B"duration"

**pr-elderly-excuse-27**

Runs Threads Monitor Setup

1 filter Last 7 days Root Runs LLM Calls

Name Input

LangGraph human: AI Agents ta

TRACE

LangGraph 46.82s

- generate 7.37s
- gpt-4o 7.36s
- should\_continue 0.00s
- reflect 8.16s
- gpt-4o 8.16s
- generate 2.64s
- gpt-4o 2.63s
- should\_continue 0.00s
- reflect 8.13s
- gpt-4o 8.13s
- generate 4.38s
- gpt-4o 4.38s
- should\_continue 0.00s
- reflect 7.83s
- gpt-4o 7.83s
- generate 8.26s
- gpt-4o 8.25s
- should\_continue 0.00s

**LangGraph**

Run Feedback Metadata

Input

HUMAN  
AI Agents taking over content creation

Output

HUMAN  
AI Agents taking over content creation

AI

The future of content creation is here, and it's powered by AI! 🎉 Watch as AI Agents transform the landscape with lightning-fast innovation and creativity, crafting compelling stories, art, and media. Are we witnessing a new era of artistic collaboration? 🎉 Share your thoughts! #AIAgents #ContentCreation #Innovation #FutureIsNow

HUMAN

START TIME: 02/13/2025, 01:07:37 PM

END TIME: 02/13/2025, 01:08:24 PM

TIME TO FIRST TOKEN: N/A

STATUS: Success

TOTAL TOKENS: 7,179 tokens / \$0.0310425

LATENCY: 46.82s

TYPE: Chain



smith.langchain.com/o/e4a9f3da-d95f-42b8-be4e-c631ca42cc3b/projects/p/8101fd03-6f4d-494f-af92-acb651c8fb3a?timeModel=%7B"duration"

**pr-elderly-excuse-27**

Runs Threads Monitor Setup

1 filter Last 7 days Root Runs LLM Calls

Name Input

LangGraph human: AI Agents ta

TRACE

LangGraph 46.82s

- generate 7.37s
- gpt-4o 7.36s
- should\_continue 0.00s
- reflect 8.16s
- gpt-4o 8.16s
- generate 2.64s
- gpt-4o 2.63s
- should\_continue 0.00s
- reflect 8.13s
- gpt-4o 8.13s
- generate 4.38s
- gpt-4o 4.38s
- should\_continue 0.00s
- reflect 7.83s
- gpt-4o 7.83s
- generate 8.26s
- gpt-4o 8.25s
- should\_continue 0.00s

**LangGraph**

Run Feedback Metadata

witnessing a new era of artistic collaboration? 🎉 Share your thoughts! #AIAgents #ContentCreation #Innovation #FutureIsNow

HUMAN  
alongside humans."

2. \*\*Engage with a Question\*\*: You already have a call to action by asking for thoughts, which is great! Consider refining it or making it more engaging: "Are AI-powered creations as meaningful as human-made ones? Let's debate!"

3. \*\*Utilize Emojis\*\*: You're using symbols effectively to capture attention. Ensure to maintain a balance so the tweet doesn't feel cluttered. Highlight key phrases like "lightning-fast innovation" 🚀 and "artistic collaboration" 🎨.

4. \*\*Hashtag Strategy\*\*: You're using some effective hashtags already, but consider adding a few more that target specific communities or interests, such as #AIArt, #DigitalAge, or #TechRevolution for more specialized reach.

TYPE: Chain



## LangGraph Crash Course #9.5 - Structured LLM Outputs



Harish Neel | AI  
7.72K subscribers

Subscribe

92 | 0 | Share | Clip | Save | ...

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 screenshot shows a Jupyter Notebook interface with the following code:

```
from pydantic import BaseModel, Field
from langchain.openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4o")

class Country(BaseModel):
    """Information about a country"""

    name: str = Field(description="name of the country")
    language: str = Field(description="language of the country")
    capital: str = Field(description="Capital of the country")

structured_llm = llm.with_structured_output(Country)
structured_llm

structured_llm.invoke("Tell me about France")
```

Below the code, there is a line of code from a different file:

```
from typing_extensions import Annotated, TypedDict
```

The screenshot shows a Jupyter Notebook interface with two code cells. The first cell contains Python code defining a Pydantic model named `Country` and using it with a `ChatOpenAI` LLM. The second cell shows the result of running the code, which is a `Country` object with attributes `name`, `language`, and `capital`. A video overlay of a man speaking is visible in the background.

```
from langchain.openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4o")

class Country(BaseModel):
    """Information about a country"""

    name: str = Field(description="name of the country")
    language: str = Field(description="language of the country")
    capital: str = Field(description="Capital of the country")

structured_llm = llm.with_structured_output(Country)
structured_llm

[12] ✓ 0.0s
...
... root_async_client=<openai.AsyncOpenAI object at 0x1177c7170>, model_name='gpt-4o', model_kwargs={}, op

[13] ✓ 1.5s
...
... Country(name='France', language='French', capital='Paris')
```

The screenshot shows a Jupyter Notebook interface with a code cell containing a JSON object. The object represents a request payload for a function call, specifically for a "Country" tool. It includes fields like "role", "content", and "name". A video overlay of a man speaking is visible in the background.

```
const sampleRequestPayload = {
  "messages": [
    {
      "role": "user",
      "content": "Tell me about France"
    }
  ],
  "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"
      }
    }
  ]
}
```

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

types.ipynb M JS pydantic\_outputs.js 3, U X

3\_structured\_outputs > JS pydantic\_outputs.js > sampleRequestPayload

```
1 // API REQUEST PAYLOAD
2 // This is what LangChain sends to the OpenAI API
3 const sampleRequestPayload = [
4   "model": "gpt-4o",
5   "messages": [
6     {
7       "role": "user",
8       "content": "Tell me about France"
9     ],
10   ],
11   "tools": [
12     {
13       "type": "function",
14       "function": {
15         "name": "Country",
16         "description": "Information about a country",
17         "parameters": {
18           "type": "object",
19           "properties": {
20             "name": {
21               "type": "string",
22               "description": "name of the country"
23             },
24             "language": {
25               "type": "string",
26               "description": "language of the country"
27             },
28             "capital": {
29               "type": "string",
30               "description": "Capital of the country"
31             }
32           },
33           "required": [
34             "name",
35             "language",
36             "capital"
37           ]
38         }
39       }
40     ]
41   ]
42 ]
```



types.ipynb M JS pydantic\_outputs.js 3, U X

3\_structured\_outputs > JS pydantic\_outputs.js > sampleRequestPayload > "tools" > "function" > "parameters"

```
3 const sampleRequestPayload = {
4   "model": "gpt-4o",
5   "messages": [
6     {
7       "role": "user",
8       "content": "Tell me about France"
9     ],
10   ],
11   "tools": [
12     {
13       "type": "function",
14       "function": {
15         "name": "Country",
16         "description": "Information about a country",
17         "parameters": {
18           "type": "object",
19           "properties": {
20             "name": {
21               "type": "string",
22               "description": "name of the country"
23             },
24             "language": {
25               "type": "string",
26               "description": "language of the country"
27             },
28             "capital": {
29               "type": "string",
30               "description": "Capital of the country"
31             }
32           },
33           "required": [
34             "name",
35             "language",
36             "capital"
37           ]
38         }
39       }
40     ]
41   ]
42 };
```



types.ipynb M JS pydantic\_outputs.js 3, U

```
_structured_outputs > JS pydantic_outputs.js > [e]sampleRequestPayload
  3 const sampleRequestPayload = {
  4   "tools": [
  5     {
  6       "function": {
  7         "parameters": {
  8           "type": "object",
  9           "properties": {
 10             "name": {
 11               "type": "string",
 12               "description": "name of the country"
 13             },
 14             "language": {
 15               "type": "string",
 16               "description": "language of the country"
 17             },
 18             "capital": {
 19               "type": "string",
 20               "description": "Capital of the country"
 21             }
 22           },
 23           "required": [
 24             "name",
 25             "language",
 26             "capital"
 27           ]
 28         }
 29       }
 30     }
 31   }
 32 }
 33 }
 34 }
 35 }
 36 }
 37 }
 38 }
 39 }
 40 }
 41 }
```



types.ipynb M JS pydantic\_outputs.js 3, U

```
3_structured_outputs > JS pydantic_outputs.js > [e]sampleRequestPayload
  1 // API REQUEST PAYLOAD
  2 // This is what LangChain sends to the OpenAI API
  3 const sampleRequestPayload = [
  4   {
  5     "model": "gpt-4o",
  6     "messages": [
  7       {
  8         "role": "user",
  9         "content": "What is the capital of France?"
 10       }
 11     ],
 12     "tools": [
 13       {
 14         "type": "function",
 15         "function": {
 16           "name": "Country"
 17         }
 18       }
 19     ]
 20   }
 21 ]
 22 // API RESPONSE
 23 // This is what OpenAI returns to LangChain
 24 const sampleResponsePayload = [
 25   {
 26     "id": "completion-456abc123def",
 27     "object": "chat.completion",
 28     "created": 1712052000,
 29     "model": "gpt-4o",
 30     "choices": [
 31       {
 32         "index": 0,
 33         "message": {
 34           "role": "assistant",
 35           "content": null
 36         }
 37       }
 38     ]
 39   }
 40 ]
 41 ]
```



types.ipynb M JS pydantic\_outputs.js 3, U X

3\_structured\_outputs > JS pydantic\_outputs.js > [e] sampleRequestPayload

```
3 const sampleRequestPayload = {
42     "tool_choice": {
44         "function": {
45             "name": "Country"
46         }
47     }
48 }
49 // API RESPONSE
50 // This is what OpenAI returns to LangChain
51 const sampleResponsePayload = {
52     "id": "completion-456abc123def",
53     "object": "chat.completion",
54     "created": 1712052000,
55     "model": "gpt-4o",
56     "choices": [
57         {
58             "index": 0,
59             "message": {
60                 "role": "assistant",
61                 "content": null,
62                 "tool_calls": [
63                     {
64                         "id": "call_789xyz456",
65                         "type": "function",
66                         "function": {
67                             "name": "Country",
68                             "arguments": "{\"name\":\"France\",\"language\":\"French\",
\\"capital\":\\\"Paris\\\"}"
69                         }
70                     }
71                 ],
72             },
73             "finish_reason": "tool_calls"
74         },
75     ],
76     "usage": {
77         "prompt_tokens": 75,
78         "completion_tokens": 24,
79         "total_tokens": 99
80     }
81 }
```



types.ipynb M JS pydantic\_outputs.js 3, U X

3\_structured\_outputs > JS pydantic\_outputs.js > [e] sampleResponsePayload > [e] "choices" > [e] "message" > [e] "tool\_calls" > [e] "function" > [e] "arguments"

```
51 const sampleResponsePayload = {
54     "created": 1712052000,
55     "model": "gpt-4o",
56     "choices": [
57         {
58             "index": 0,
59             "message": {
60                 "role": "assistant",
61                 "content": null,
62                 "tool_calls": [
63                     {
64                         "id": "call_789xyz456",
65                         "type": "function",
66                         "function": [
67                             {
68                                 "name": "Country",
69                                 "arguments": "{\"name\":\"France\",\"language\":\"French\",
\\"capital\":\\\"Paris\\\"}"
70                             }
71                         ]
72                     }
73                 ],
74             },
75             "finish_reason": "tool_calls"
76         },
77     ],
78     "usage": {
79         "prompt_tokens": 75,
80         "completion_tokens": 24,
81         "total_tokens": 99
82     }
83 }
```





```
from typing_extensions import Annotated, TypedDict
from typing import Optional

# TypedDict
class Joke(TypedDict):
    """Joke to tell user."""
    setup: Annotated[str, ..., "The setup of the joke"]

    # Alternatively, we could have specified setup as:

    # setup: str          # no default, no description
    # setup: Annotated[str, ...]  # no default, no description
    # setup: Annotated[str, "foo"] # default, no description

    punchline: Annotated[str, ..., "The punchline of the joke"]
    rating: Annotated[Optional[int], None, "How funny the joke is, from 1 to 10"]

structured_llm = llm.with_structured_output(Joke)

structured_llm.invoke("Tell me a joke about cats")
```

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



```
# Alternatively, we could have specified setup as:

# setup: str          # no default, no description
# setup: Annotated[str, ...]  # no default, no description
# setup: Annotated[str, "foo"] # default, no description

punchline: Annotated[str, ..., "The punchline of the joke"]
rating: Annotated[Optional[int], None, "How funny the joke is, from 1 to 10"]

structured_llm = llm.with_structured_output(Joke)

structured_llm.invoke("Tell me a joke about cats")
```

[14] ✓ 8.5s

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

```
json_schema = {  
     "title": "joke",  
     "description": "Joke to tell user.",  
     "type": "object",  
     "properties": {  
         "setup": ...,  
         "punchline": ...,  
         "rating": ...  
     }  
}
```

The screenshot shows a Jupyter Notebook interface with two open files: `types.ipynb` and `pydantic_outputs.js`. The `pydantic_outputs.js` file contains the following code:

```
json_schema = {
    "title": "joke",
    "description": "Joke to tell user.",
    "type": "object",
    "properties": {
        "setup": {
            "type": "string",
            "description": "The setup of the joke",
        },
        "punchline": {
            "type": "string",
            "description": "The punchline to the joke",
        },
        "rating": {
            "type": "integer",
            "description": "How funny the joke is, from 1 to 10",
            "default": None,
        },
    },
    "required": ["setup", "punchline"],
}
structured_llm = llm.with_structured_output(json_schema)

structured_llm.invoke("Tell me a joke about cats")
```

A video feed of a person is visible in the top right corner of the notebook window.

The 3<sup>rd</sup> way of getting structured output from the LLM is by using a **json\_schema** as above

The screenshot shows the same Jupyter Notebook interface after running the code. The output cell [15] shows the generated JSON response:

```
[15] ✓ 0.9s
...
{'setup': 'Why was the cat sitting on the computer?',
 'punchline': 'Because it wanted to keep an eye on the mouse!',
 'rating': 7}
```

A video feed of a person is visible in the bottom right corner of the notebook window.

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