**Daily Report for IVY Training #1**

**Date: 16/1/2024**

**Name: Bui Tuan Anh**

1. **Tech stack covered**

* Langchain

**Key Learnings:**

* Langchain LLM’s Intergration
* Langchain History Conservation handling

1. **Progress**

* **Langchain**
* **Prompt Management** (**Note**: Because the price and problem with Azure Open API key, so now i use Gemini LLM (1.5-flash) free API for testing.

**Purpose**

Create and managing multiple type of prompt to LLM in Langchain

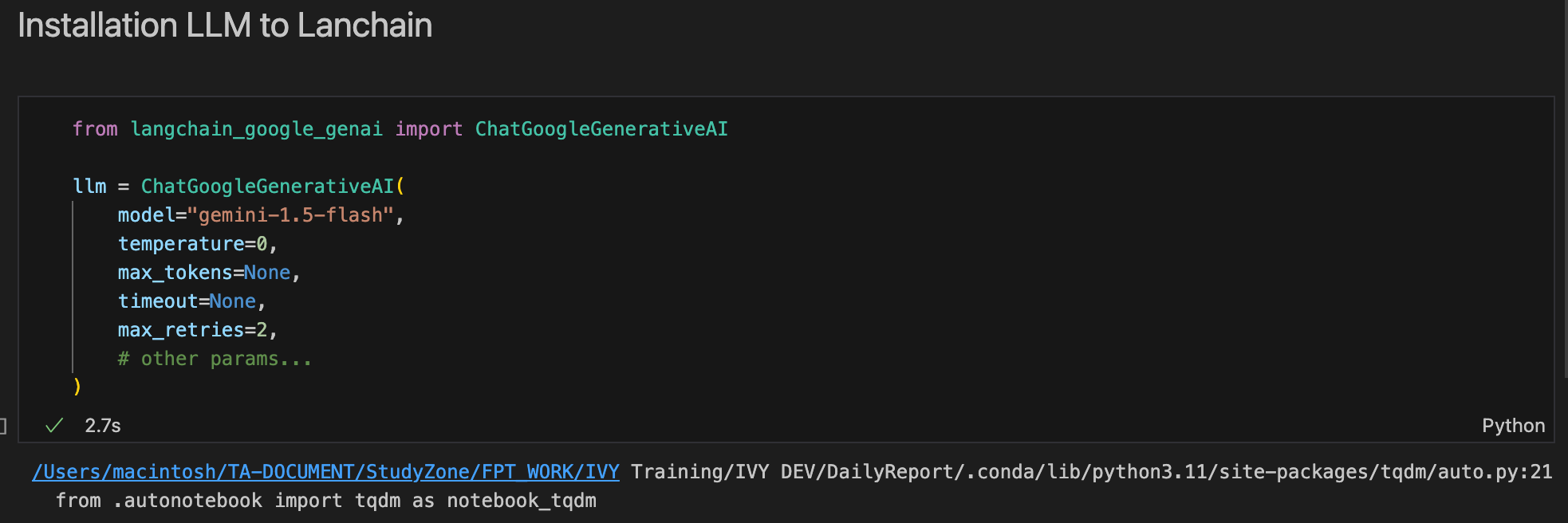
**Feature**

1. **Intergrate LLM with Langchain (Gemini 1.5 Flash)**

* **Installation**

**Parameter of ChatGoogleGenerativeAI**

* **model (**Name of Gemini model such as gemini-1.5-flash, gemini-1.5-pro,... get more [here](https://ai.google.dev/gemini-api/docs/models/gemini)**)**
* **temperature** (Control random response, more closer to 0 make response more relate to user prompt, more closer to 1 make the response more creative but sometime can make hallucination)
* **max token** (Define the max number of token in the response)
* **timeout** (max time for waiting response from llm)
* **max\_retries** (number of time allow to retry when API call fail)

****

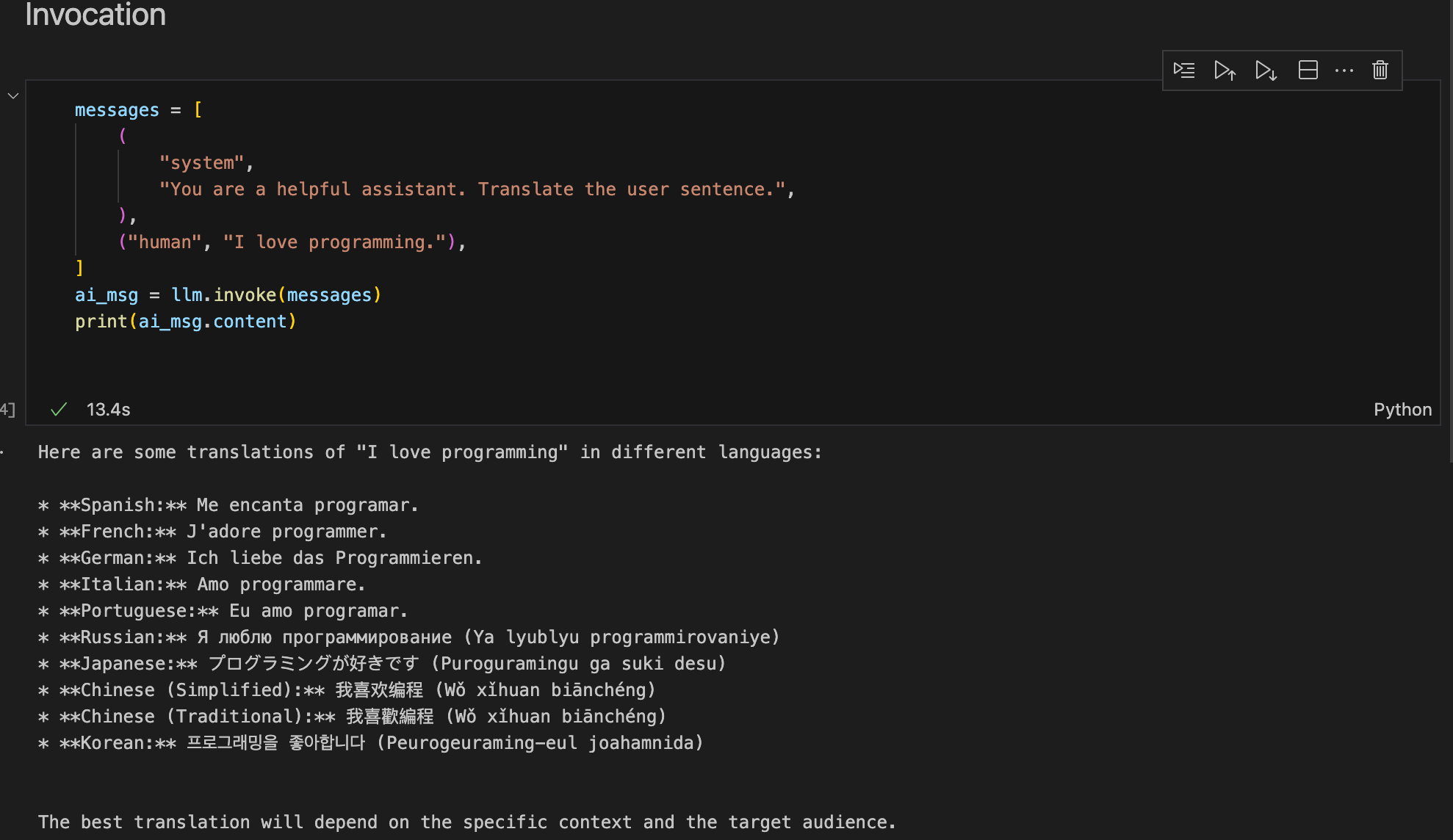
* **Invocation**

**Input processing:** Take user prompt

* **“system”**: Specifies instruction for model to response. In this case I instructs the model to translate the sentence.
* **“human”**: Here is prompt of user.

**Model Processing:** Send input to LLM for process

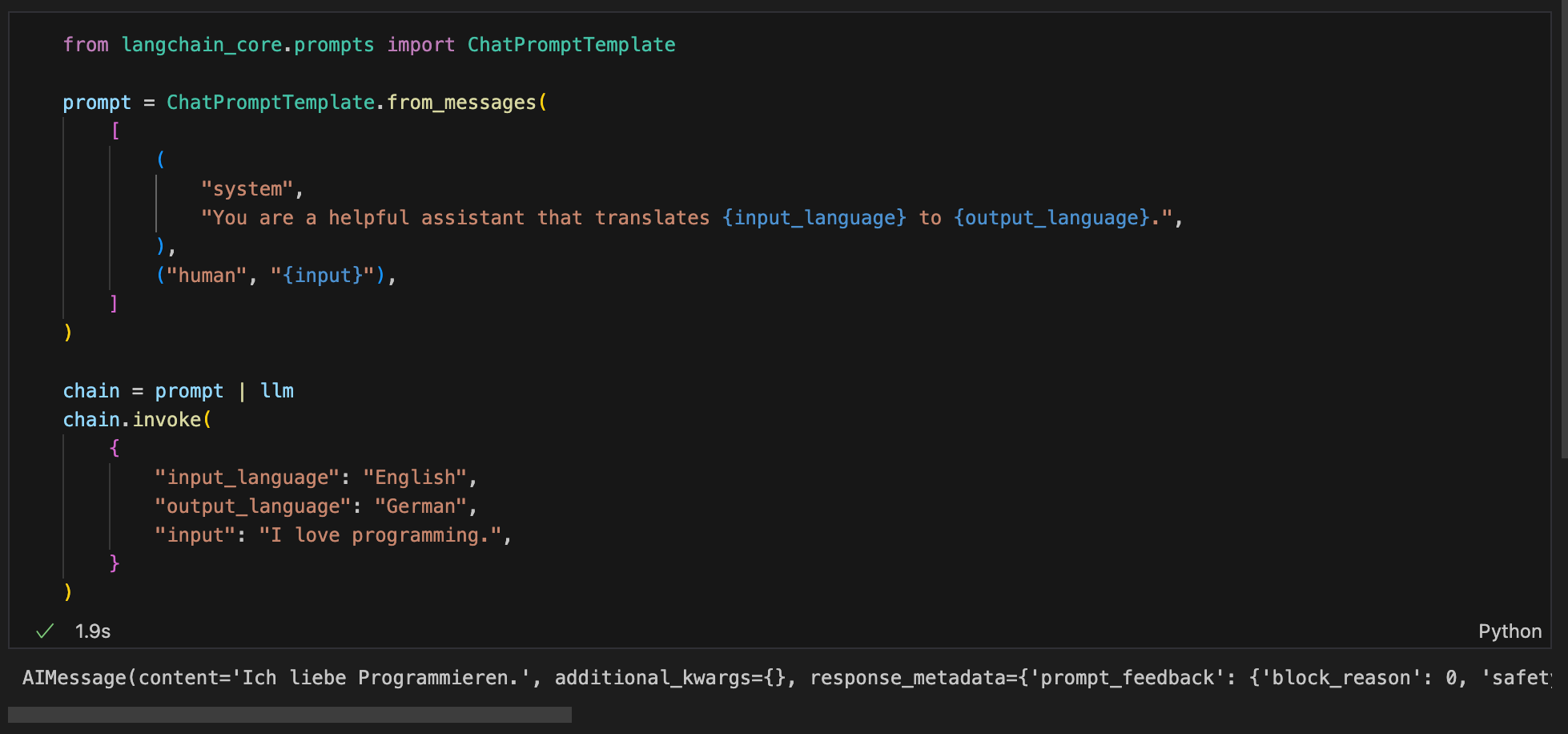
**Output response:** Retrieve response from LLM



* **Chainning**

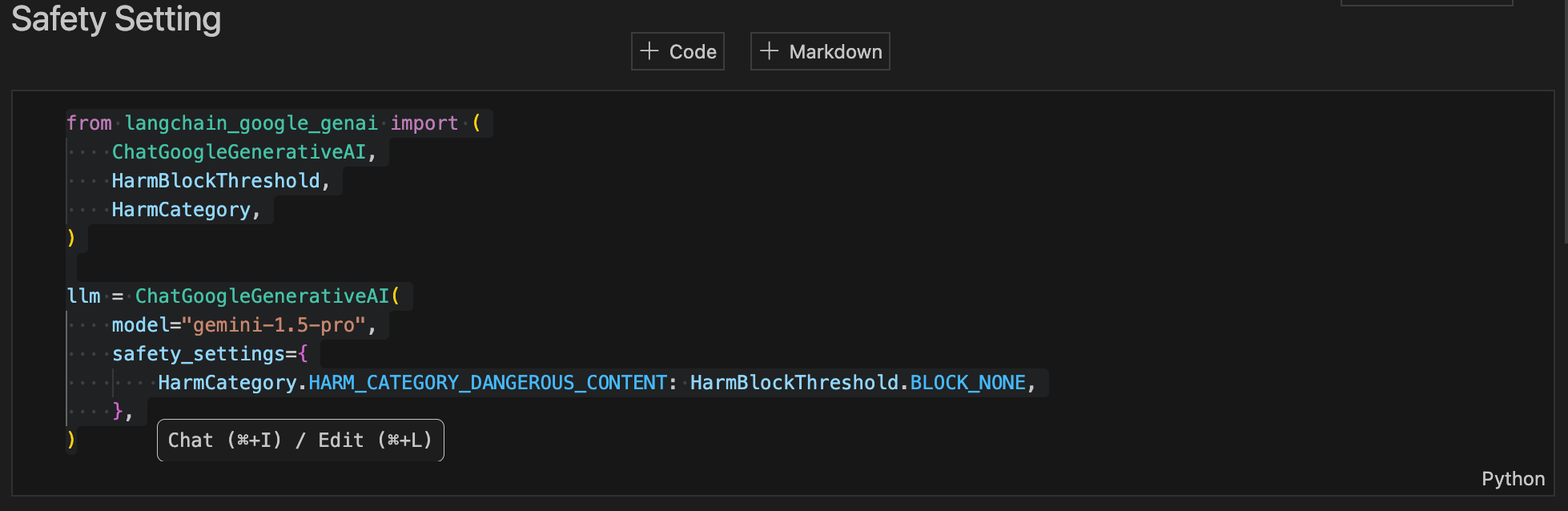
**Dynamic Prompts:** Different from Invocation, chaining use ({}) allow to use same template for multiple cases.

**Combining Components**: “|” Operator to joint the prompt and LLM to a chain.



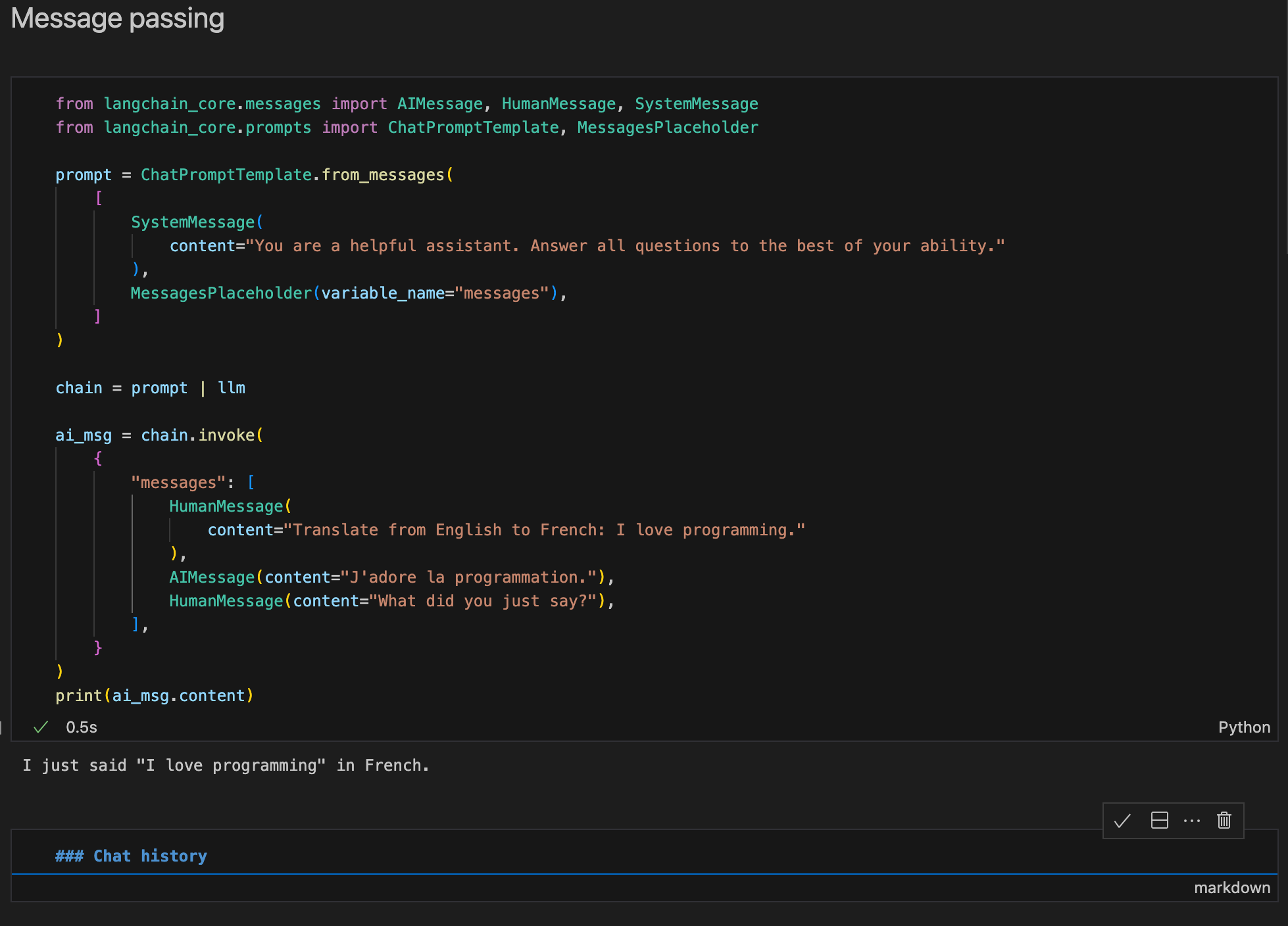
* **Safety Setting**

Gemini models have default safety settings that can be overridden. If you are receiving lots of "Safety Warnings" from your models, you can try tweaking the safety\_settings attribute of the model. For example, to turn off safety blocking for dangerous content, you can construct your LLM as follows:



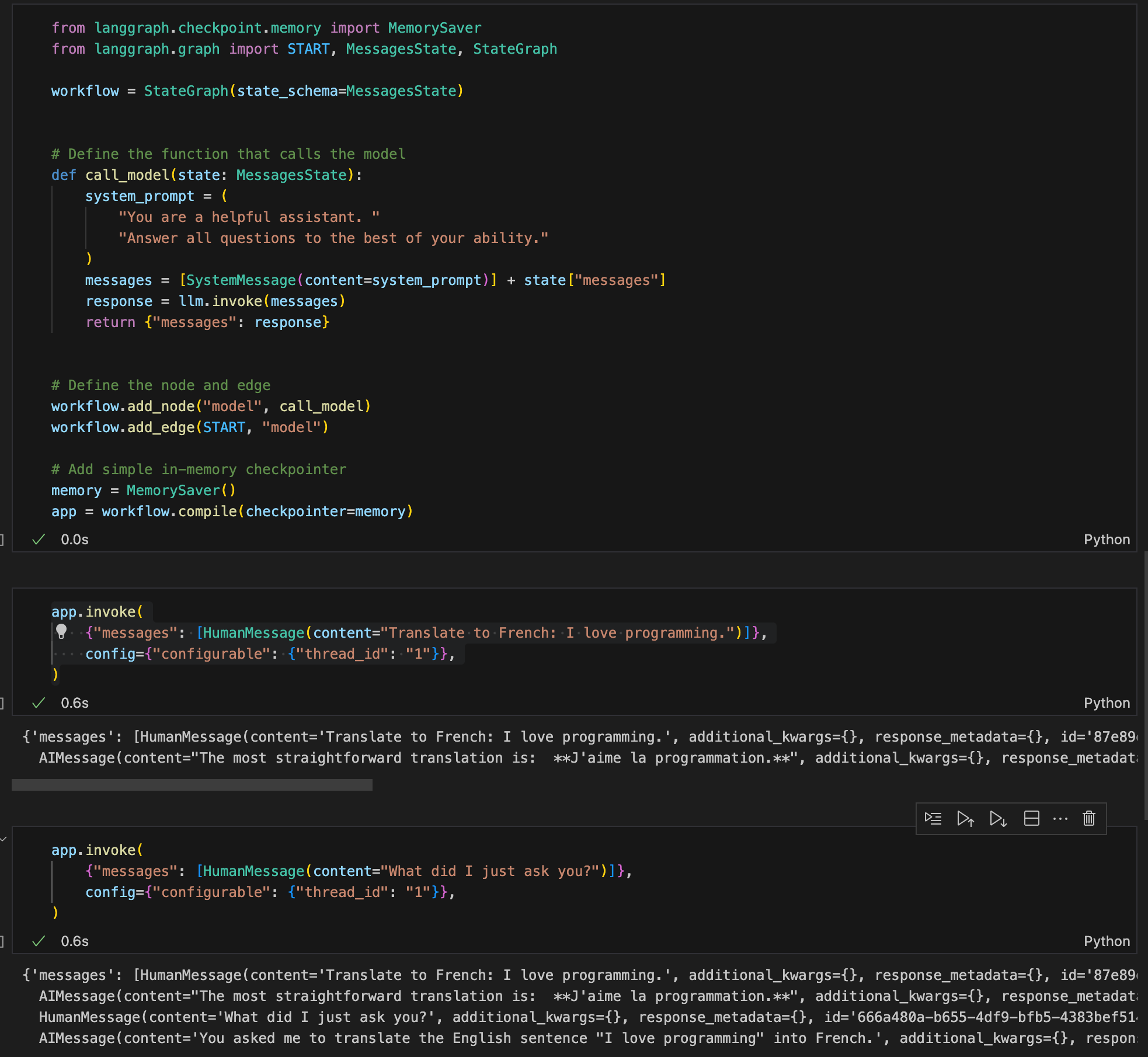
* **History Management**
* **Message passing**

The simplest form of memory is simply passing chat history messages into a chain.



A placeholder (variable\_name="messages") that will be replaced with the actual chat history at runtime.

* **Automatic history management**



**Core concepts**:

**StateGraph**: Conversational workflow as state machine, use to creating Nodes and Edges.

**Memory**: Memory saver is a checkpoint, storing and retrieving conservation state (message).

**Threading**: thread\_id allow managing multiple independent conservations (threads).

Code:

messages = [SystemMessage(content=system\_prompt)] + state["messages"]

=> Combine user prompt with previous message from state

workflow.add\_node("model", call\_model)

workflow.add\_edge(START, "model")

=> Each node in graph represent a processing (in this case is call\_model)

=> The edge connects the START (node) to “model” node. (More detail in Langgraph section)

app.invoke(

{"messages": [HumanMessage(content="Translate to French: I love programming.")]},

config={"configurable": {"thread\_id": "1"}},)

app.invoke(

{"messages": [HumanMessage(content="What did I just ask you?")]},

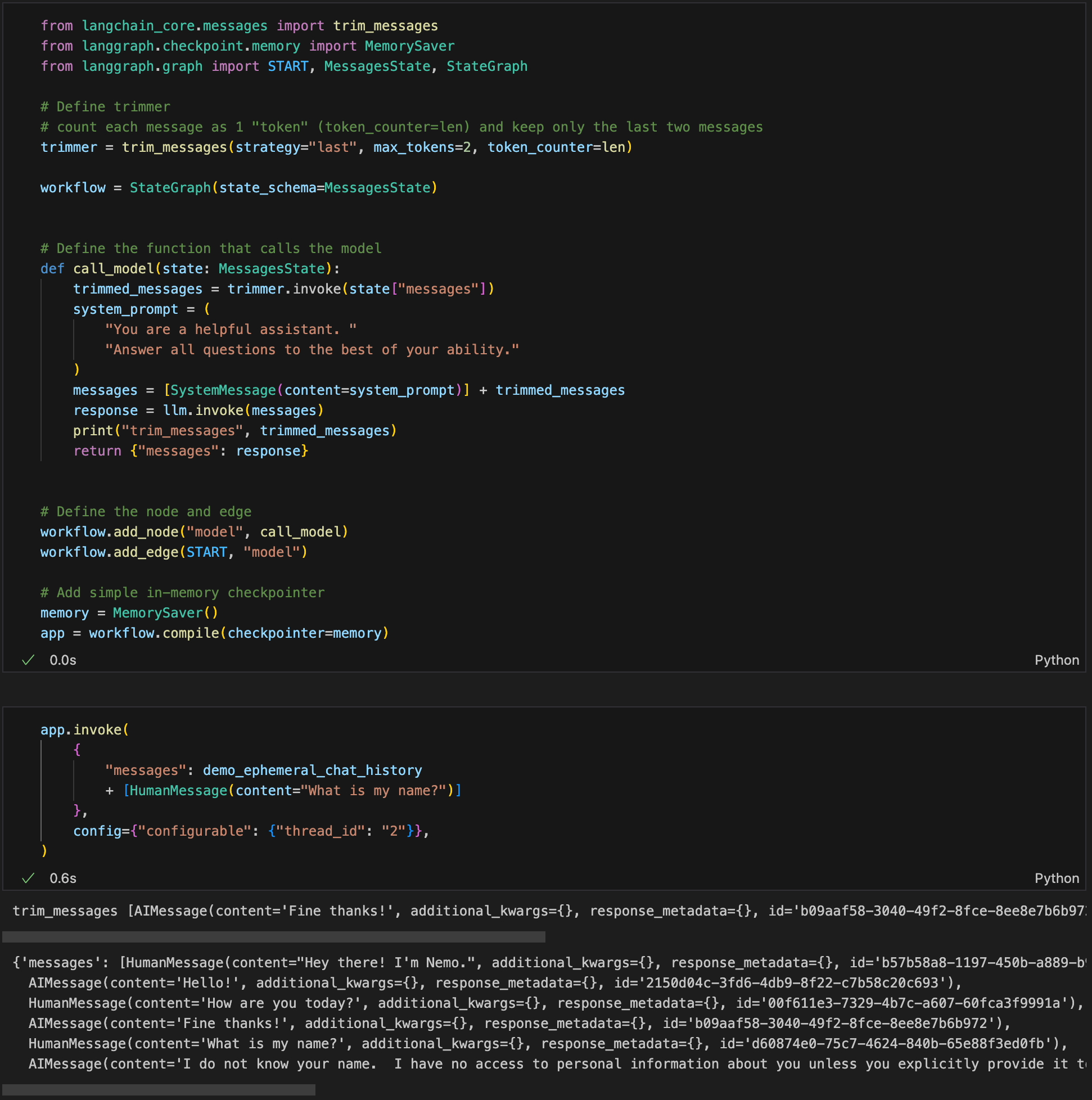
config={"configurable": {"thread\_id": "1"}},)

1st Prompt:

The conversation state (including this message and the response) is saved under thread\_id="1"

2nd Prompt  
The persisted state (from the first invocation) is automatically retrieved by using thread\_id="1"

* **Trim Message**

****

But let's say we have a very small context window, and we want to trim the number of messages passed to the model to only the 2 most recent ones. We can use the built in trim\_messages util to trim messages based on their token count before they reach our prompt. In this case we'll count each message as 1 "token" and keep only the last two messages. ( max\_tokens = 2)

=> AIMessage(content='I do not know your name. I have no access….)

**Summarize History**

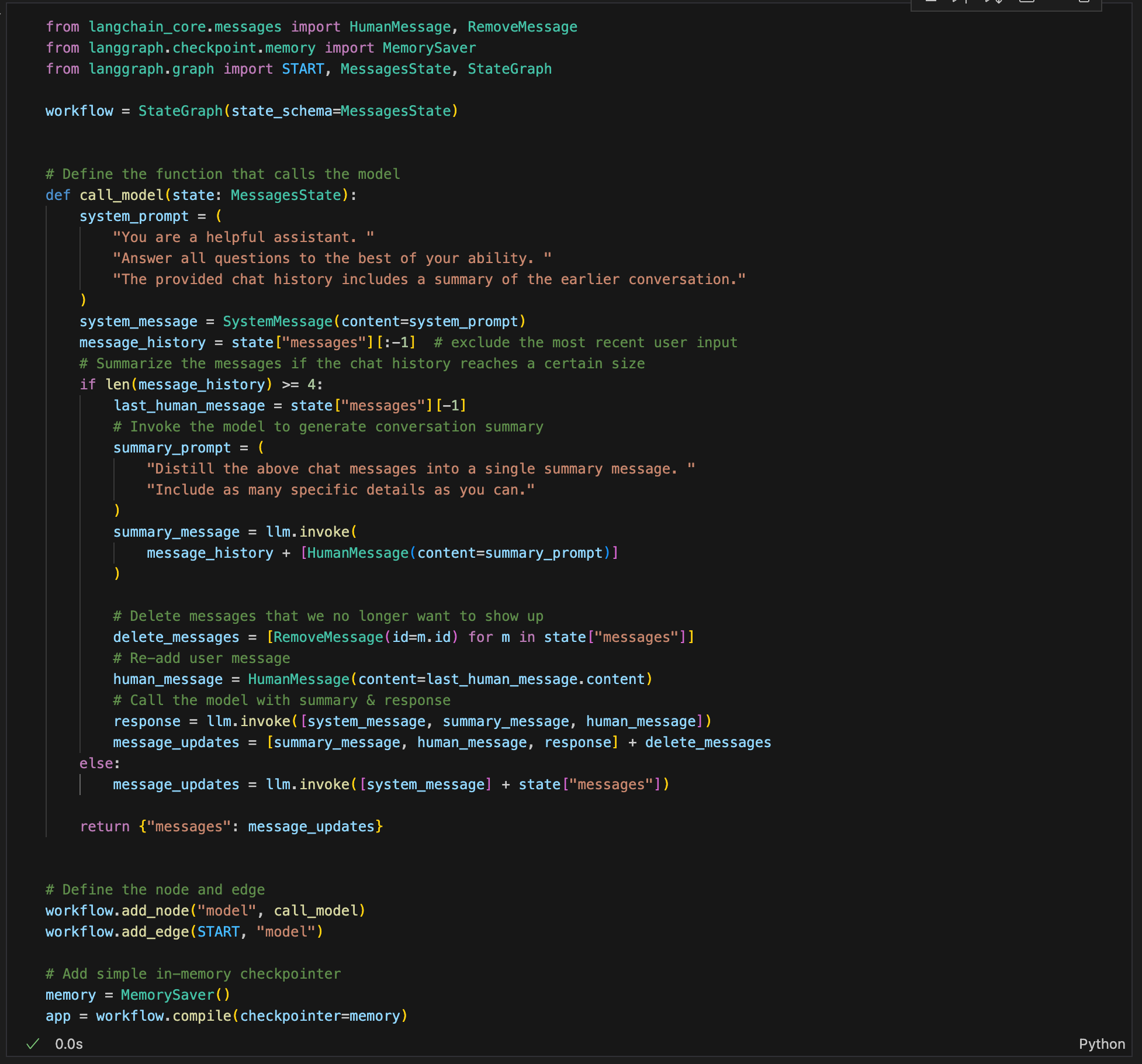
**demo\_ephemeral\_chat\_history = [**

**HumanMessage(content="Hey there! I'm Nemo."),**

**AIMessage(content="Hello!"),**

**HumanMessage(content="How are you today?"),**

**AIMessage(content="Fine thanks!"),]**

****

**Key point:**

**If the message history has 4 or more messages, summarization is triggered.**

**A special** summary\_prompt **asks the assistant to summarize the previous chat messages into one concise summary.**

**The assistant generates a** summary\_message **summarizing the earlier conversation.**

delete\_messages = [RemoveMessage(id=m.id) for m in state["messages"]]

**=> Marks older messages for removal, preventing them from being sent to the model in the future.**

else: message\_updates = model.invoke([system\_message] + state["messages"])

If the chat history is small (fewer than 4 messages), all messages are passed directly to the model without summarization.