Crash Course OpenAI SDK

Playlist ➔ https://www.youtube.com/playlist?list=PL0vKVrkG4hWpQJfc8as3tD4CClyIsZcag

OpenAI SDK Official Docs ➔ https://openai.github.io/openai-agents-python/

OpenAI Agents SDK (Introduction)

- OpenAI agents enables you to build agentic AI apps in a lightweight, easy-to-use package with very few abstractions.
- It's a production-ready upgrade of our previous experimentation for agents, Swarm.
- The Agents SDK has a very small set of primitives:
  - **Agents**, which are LLMs equipped with instructions (system prompt) and tools
  - **Handoffs**, which allow agents to delegate to other agents for specific tasks
  - **Guardrails**, which enable the inputs to agents to be validated
  - **Sessions**, which automatically maintains conversation history across agent runs

Why use the Agents SDK?

The SDK is designed with two main ideas:

1. It has **enough features to be useful**, but not too many, so it’s still easy and quick to learn.
2. It **works well right away**, but you can also change things to fit your needs.

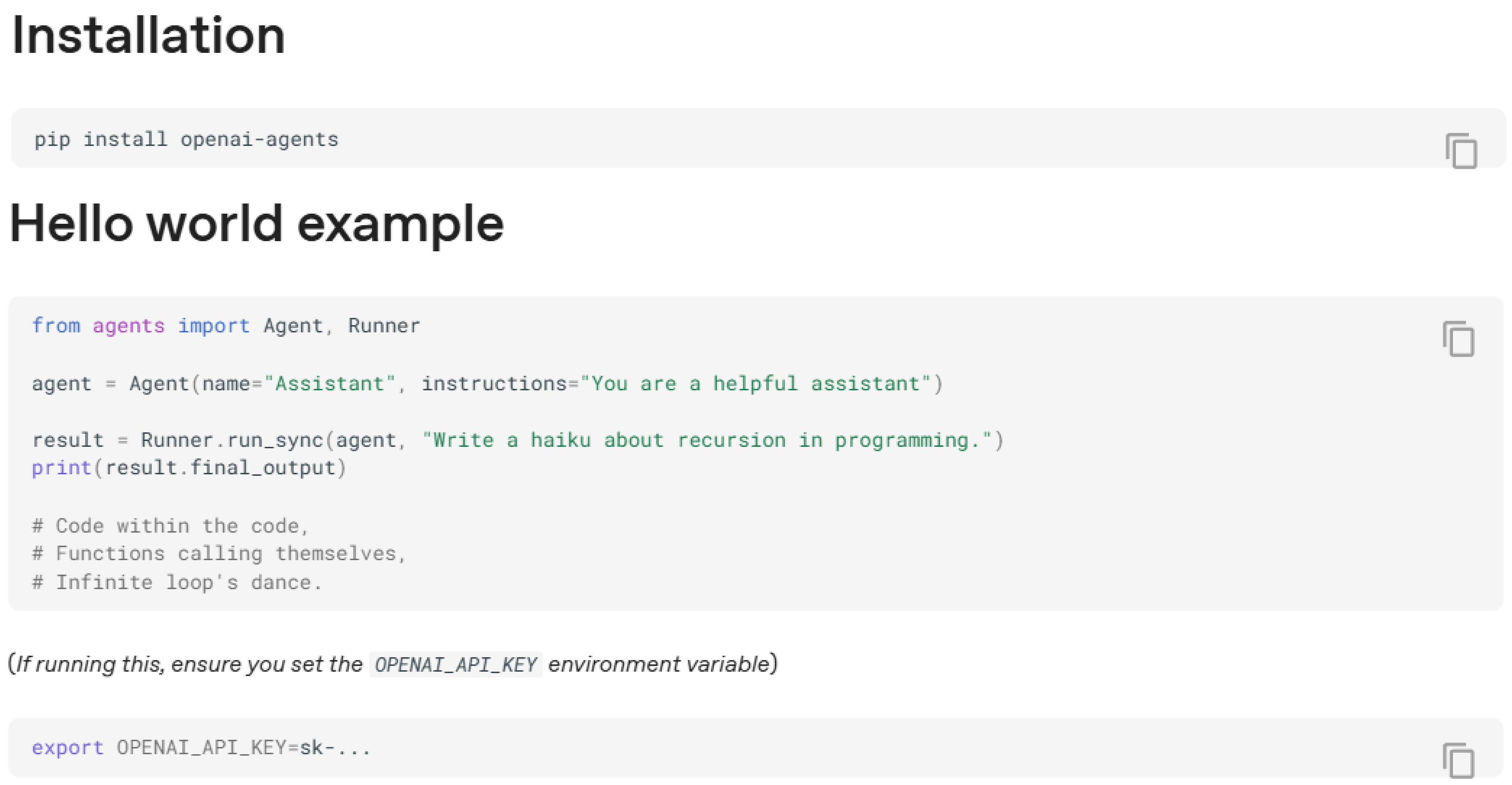
Here are the main features of the SDK:

- **Agent loop:** Automatically handles calling tools, getting answers from the AI, and repeating the process until the task is finished.
- **Python-first:** You can use normal Python code to control and link agents. No need to learn anything new.
- **Handoffs:** Lets one agent pass work to another agent, so they can work together.

Why use the Agents SDK?

- **Guardrails:** Runs checks on the inputs while the agents work. If something is wrong, it stops early.
- **Sessions:** Keeps track of the conversation history automatically, so you don’t have to manage it yourself.
- **Function tools:** You can turn any Python function into a tool. The SDK will automatically create rules (schema) and check inputs using Pydantic.
- **Tracing:** Helps you see and debug how your code is running. You can also use OpenAI’s tools to evaluate, fine-tune, and improve your agents.

Installation & Example



Installation

```
pip install openai-agents
```

Hello world example

```python
from agents import Agent, Runner

agent = Agent(name="Assistant", instructions="You are a helpful assistant")

result = Runner.run_sync(agent, "Write a haiku about recursion in programming.")
print(result.final_output)

# Code within the code,
# Functions calling themselves,
# Infinite loop's dance.
```

(If running this, ensure you set the `OPENAI_API_KEY` environment variable)

```
export OPENAI_API_KEY=sk-...
```

API’s Types

- Most of the power features of OpenAI SDK will be used with OpenAI API key.
- OpenAI introduces 3 types of API
  - **Chat Completion** (take single input, give answer and nothing remember anything)
  - **Assistant API** (previously introduced, thread had been made of your chat and save on server)
  - **Responsive API** (assistant API merge with responsive API)

4 things we give to (any advance) LLMs

- We can give four stuffs/things to any LLMs
  - **System Prompt:**
    - The system prompt defines the persona or overall behavior of the LLM. It sets the tone, style, and role the model should adopt during the conversation.
    - For example, you might instruct the model to act as a helpful assistant, a friendly tutor, or a professional consultant. This prompt helps align the model’s responses with the intended context or use case.
  - **User Prompt:**
    - This is the *direct input* from the user — a question, instruction, or any form of message that the user sends to the LLM.
    - The user prompt is what the model responds to in real time, based on the previously defined system prompt and any context available.
  - **Tool Schema:**

    A tool schema provides the *definition and structure* of any external tools or functions the LLM can use. It includes:

4 things we give to (any advance) LLMs

- The name of the tool
- A description of what it does
- Its parameters (inputs), including their names, types, and whether they are required or optional

This schema helps the model understand how to correctly call the tool when it's needed.

- **Tool Message (or Tool Output):**
  - When the LLM calls an external tool (like a function or API), the tool returns a result. This result is called the tool message, and it is passed back to the LLM.
  - The model then uses this output to continue the conversation, often interpreting or summarizing the result for the user.

2 LLMs Responses

Large Language Models (LLMs) are capable of generating two primary types of responses, depending on the context and requirements of the application.

Plain Text Response

- In most cases, the LLM will return a response in natural language — a plain text message written in human-readable form. While this is useful for direct communication, it may need to be transformed into a more structured format for programmatic use.
- To extract meaningful data from this text, developers can use techniques such as:
  - **Pydantic**: A Python library that allows you to define data models and validate or parse structured content (like JSON) from plain text using defined schemas.

2 LLMs Responses

Tool Invocation

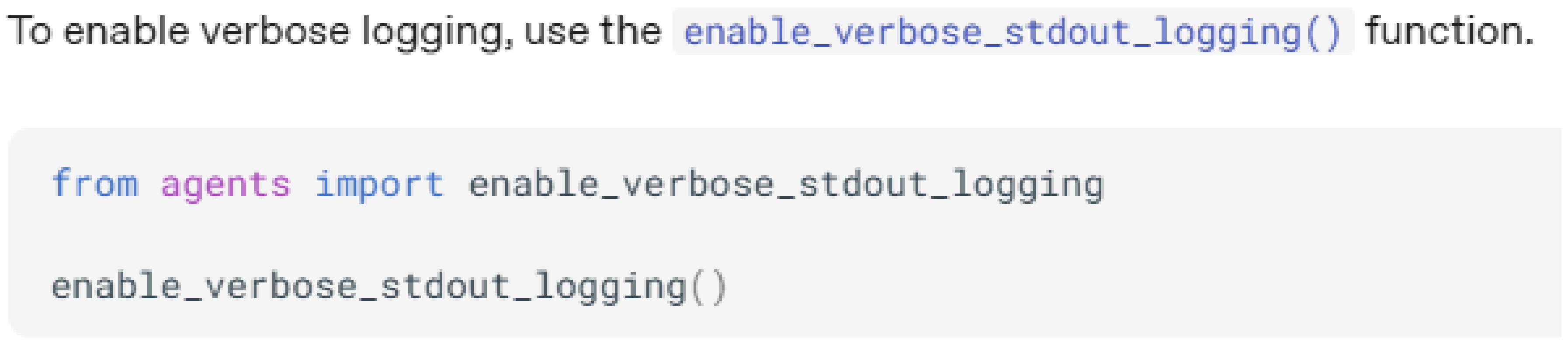
- In more advanced workflows, an LLM can go beyond plain text and decide to **call a tool** — for example, a function, API, database query, or external service.
- This typically happens when the LLM determines that answering the user’s request requires real-world data, computation, or an external capability beyond its own model knowledge.
- To enable this, developers define:
    - A **tool schema**, which describes what tools are available, including their names, descriptions, parameters, and constraints.
    - A **mechanism to handle the tool call**, so that when the LLM decides a tool is needed, the call is executed and the result is returned to the LLM.

Debugging mode of Code

- The **debugging mode** allows developers to enable **verbose logging** to the standard output (stdout). This mode is especially useful when you are developing, testing, or troubleshooting agent workflows.
- When **enabled**, the system will output detailed logs showing the internal behavior of the agent loop, including:
  - Prompts sent to the LLM
  - Tool calls and their parameters
  - Responses from tools
  - Intermediate and final decisions made by the agent
  - Errors or validation issues, if any

Debugging mode of Code

- This level of visibility can help you:
  - Understand how the agent is making decisions
  - Identify why an agent might be failing or behaving unexpectedly
  - Optimize tool integration or prompt design
- Check this link for more detail ➔ https://openai.github.io/openai-agents-python/config/#debug-logging



To enable verbose logging, use the `enable_verbose_stdout_logging()` function.

```python
from agents import enable_verbose_stdout_logging

enable_verbose_stdout_logging()
```

When tracing is enabled in the OpenAI Agents SDK?

- When **tracing** is enabled in the **OpenAI Agents SDK**, several things happen in the background to help you **monitor, debug, and analyze** agent behavior. Here's a brief breakdown:

1. **Unique Tracing ID (Trace ID)**
   - Each agent run is assigned a unique trace ID.
   - This ID helps identify and group all events related to a single agent execution.
2. **Span IDs**
   - Each step or operation (e.g., LLM call, tool call, validation check) gets a span ID.
   - These spans are nested under the main trace, allowing detailed tracking of every sub-process.
   - Spans show timing, order, and duration of events.

When tracing is enabled in the OpenAI Agents SDK?

3. Structured Logging and Event Capture

- The SDK captures detailed metadata for each step: inputs, outputs, tool names, parameters, LLM prompts/responses, and errors.
- This data is stored in a structured way for inspection.

4. Visualization and Monitoring

- If connected to OpenAI’s platform (or tools like LangSmith in LangChain), you can visualize traces as a tree or timeline.
- You can step through each phase of agent reasoning, helping you understand how decisions were made.

5. Support for Evaluation & Fine-Tuning

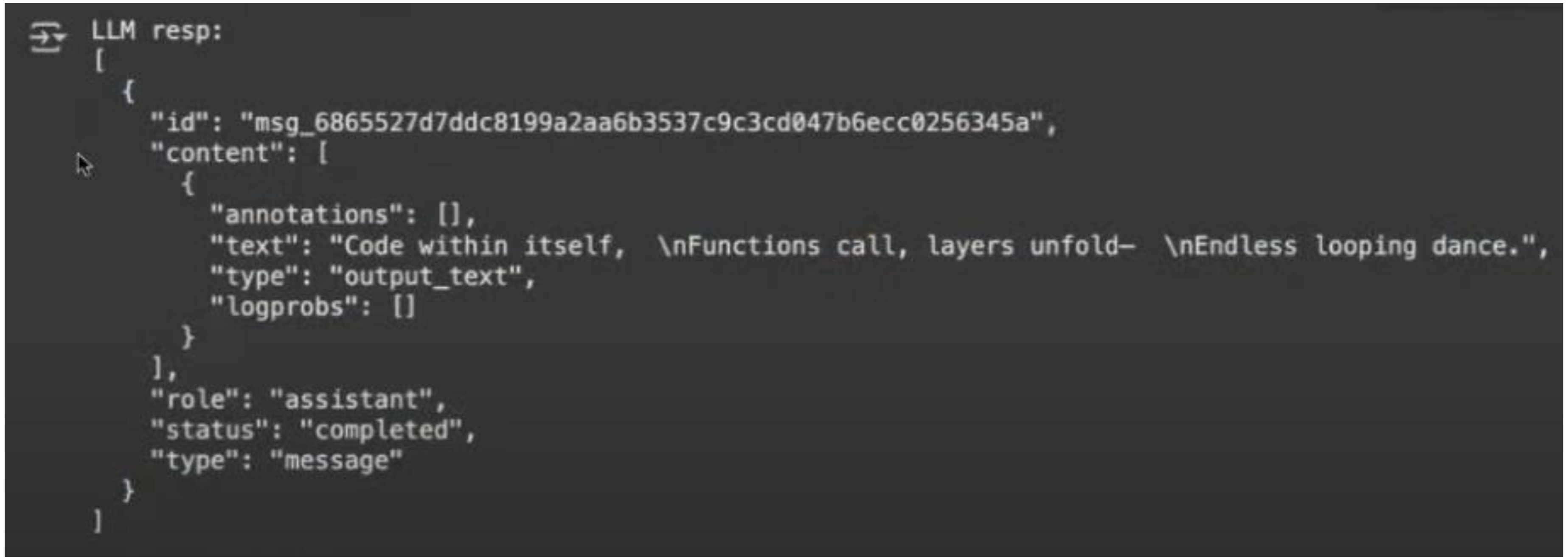
- The trace data can be used for automated evaluations, performance tuning, and even fine-tuning models based on real-world usage patterns.

6. Optional Integration with OpenAI's Suite

- Tracing works seamlessly with OpenAI’s evaluation tools, distillation pipelines, and monitoring dashboards, if configured.

LLM Response

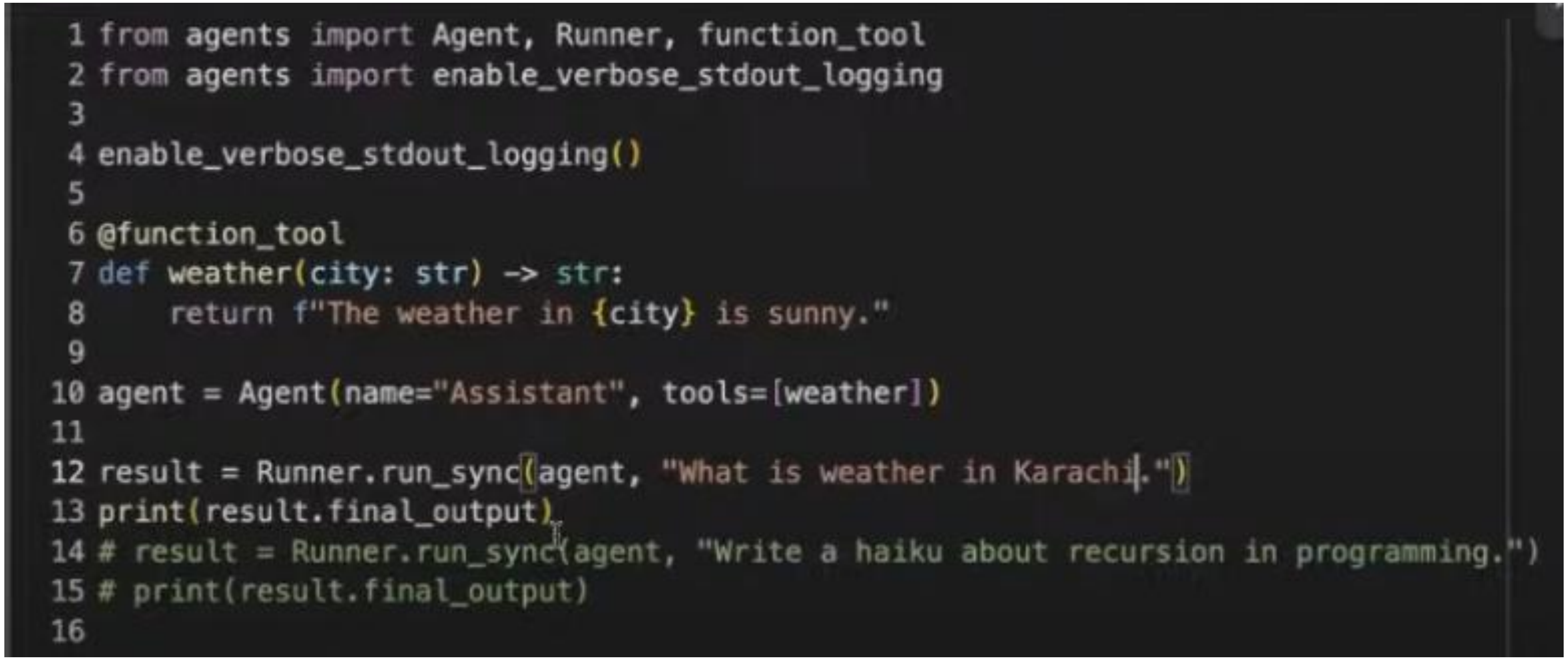
- LLMs are responsive APIs and it save history on backend; see below picture we have response id of that chat as well.
- Because we majorly work on stateless Chat API that why we don’t bother by it previously.



```
LLM resp:
[
  {
    "id": "msg_6865527d7ddc8199a2aa6b3537c9c3cd047b6ecc0256345a",
    "content": [
      {
        "annotations": [],
        "text": "Code within itself,  \nFunctions call, layers unfold—  \nEndless looping dance.",
        "type": "output_text",
        "logprobs": []
      }
    ],
    "role": "assistant",
    "status": "completed",
    "type": "message"
  }
]
```

LLM Response

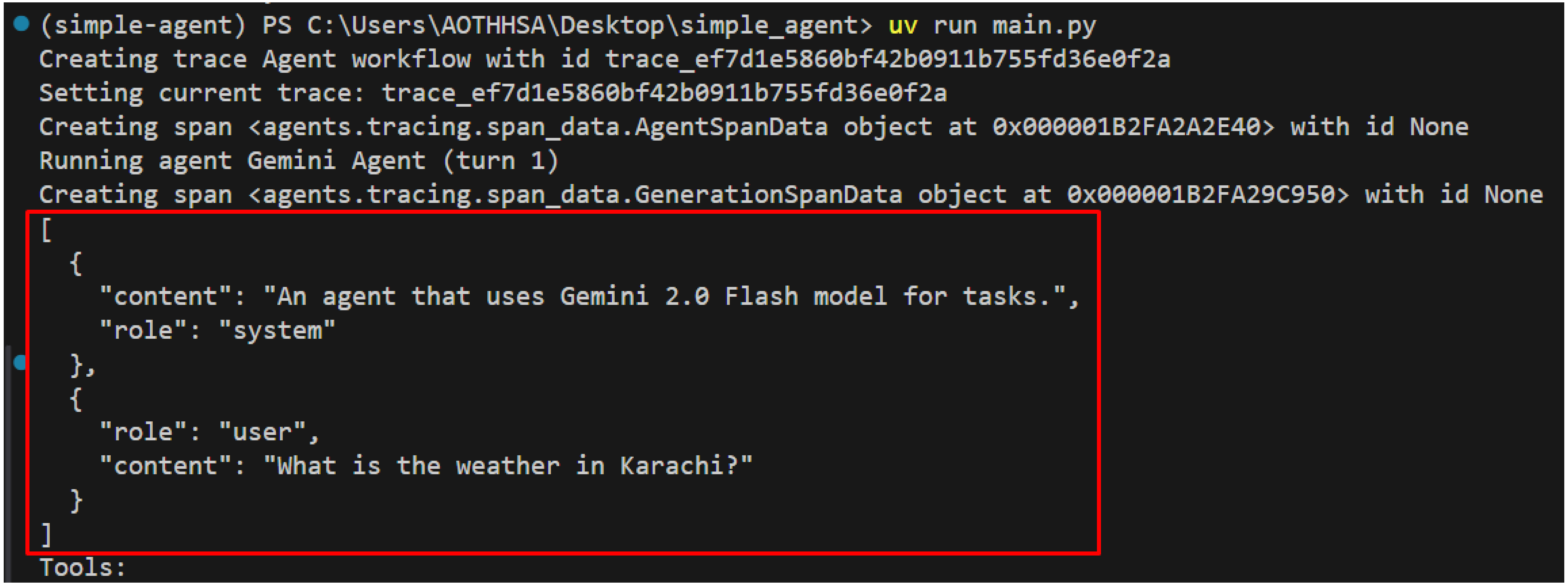
- Below is the code we wrote and observed what happen in it & its outcome



```python
from agents import Agent, Runner, function_tool
from agents import enable_verbose_stdout_logging

enable_verbose_stdout_logging()

@function_tool
def weather(city: str) -> str:
    return f"The weather in {city} is sunny."

agent = Agent(name="Assistant", tools=[weather])

result = Runner.run_sync(agent, "What is weather in Karachi.")
print(result.final_output)
# result = Runner.run_sync(agent, "Write a haiku about recursion in programming.")
# print(result.final_output)

```

LLM Response

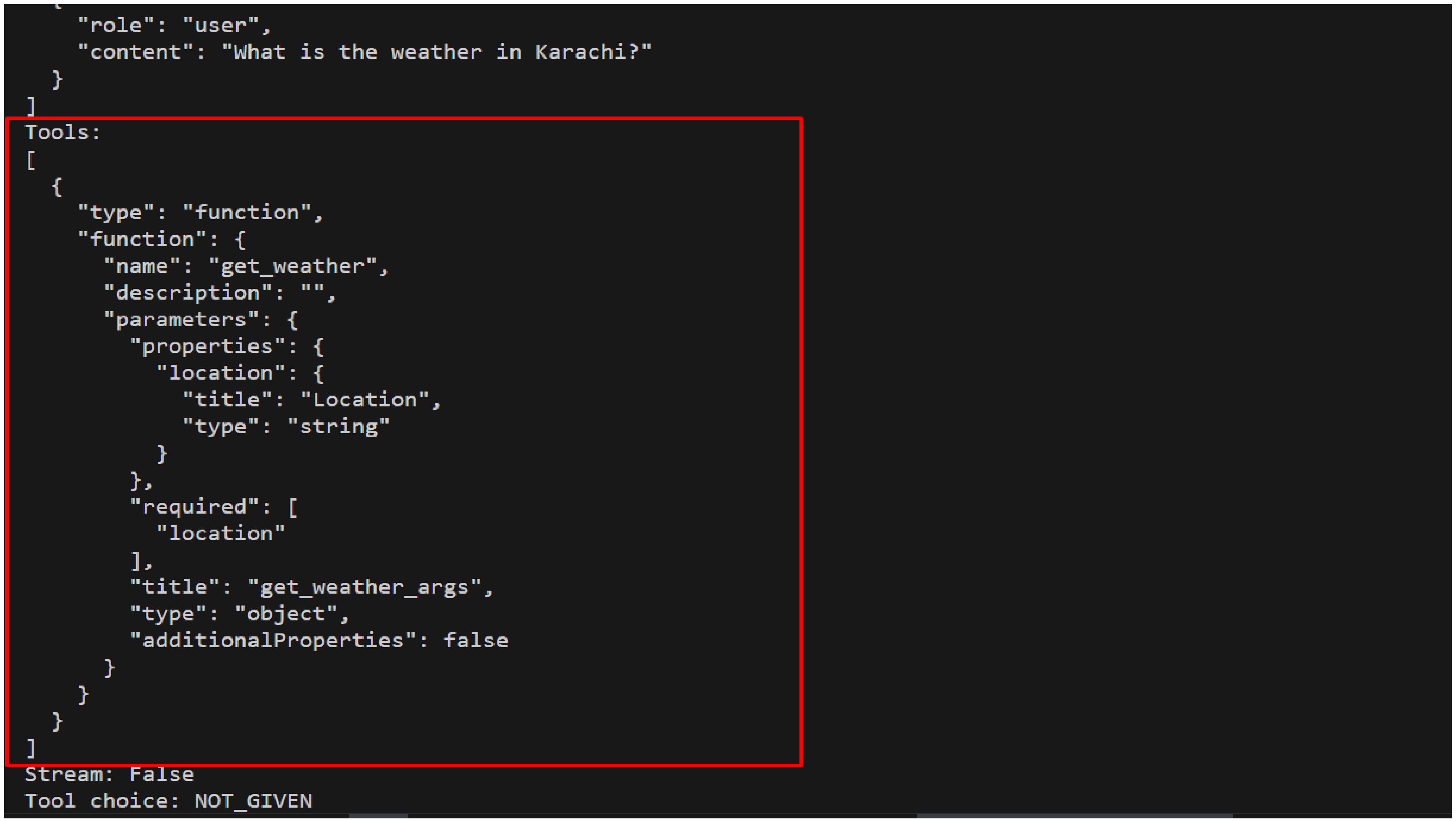
- In result (output), it will show all work which happen during execution
- As you know, 4 things we give to LLMs
  - **First is user input and second is system prompt, see below picture**



```
(simple-agent) PS C:\Users\AOTHHSA\Desktop\simple_agent> uv run main.py
Creating trace Agent workflow with id trace_ef7d1e5860bf42b0911b755fd36e0f2a
Setting current trace: trace_ef7d1e5860bf42b0911b755fd36e0f2a
Creating span <agents.tracing.span_data.AgentSpanData object at 0x000001B2FA2A2E40> with id None
Running agent Gemini Agent (turn 1)
Creating span <agents.tracing.span_data.GenerationSpanData object at 0x000001B2FA29C950> with id None
[
  {
    "content": "An agent that uses Gemini 2.0 Flash model for tasks.",
    "role": "system"
  },
  {
    "role": "user",
    "content": "What is the weather in Karachi?"
  }
]
Tools:
```

LLM Response

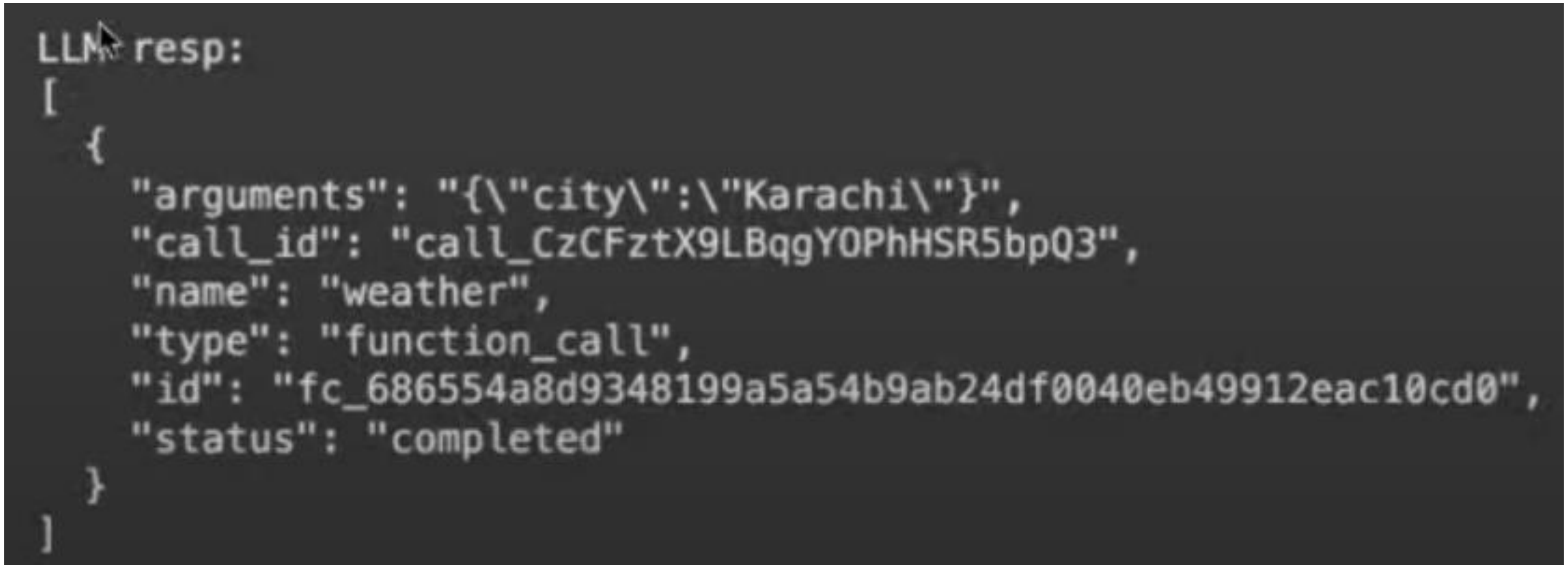
- **Third is tool schema**



```
    "role": "user",
    "content": "What is the weather in Karachi?"
  }
]
Tools:
[
  {
    "type": "function",
    "function": {
      "name": "get_weather",
      "description": "",
      "parameters": {
        "properties": {
          "location": {
            "title": "Location",
            "type": "string"
          }
        },
        "required": [
          "location"
        ],
        "title": "get_weather_args",
        "type": "object",
        "additionalProperties": false
      }
    }
  }
]
Stream: False
Tool choice: NOT_GIVEN
```

LLM Response

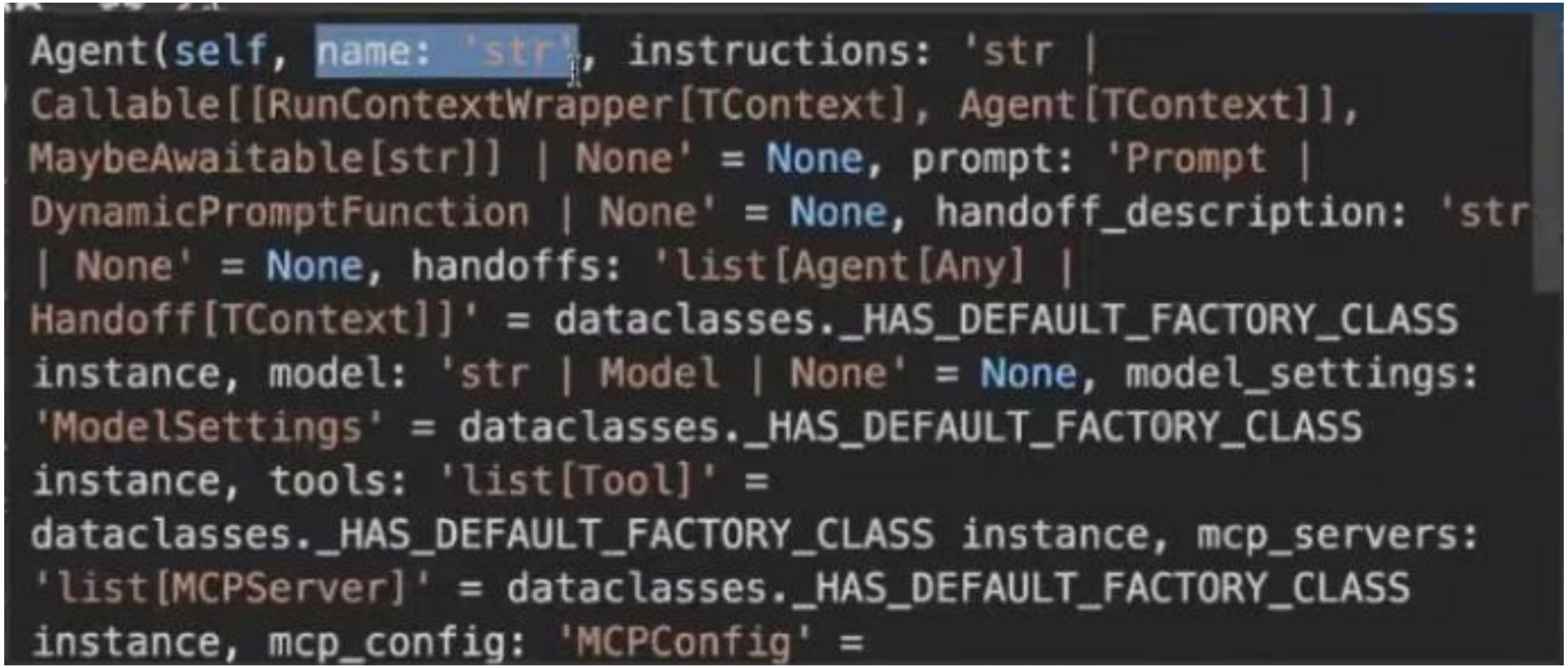
- **Last (fourth) is tool response (tool output)**



```
LLM resp:
[
  {
    "arguments": "{\"city\":\"Karachi\"}",
    "call_id": "call_CzCFztX9LBqgYOPhHSR5bpQ3",
    "name": "weather",
    "type": "function_call",
    "id": "fc_686554a8d9348199a5a54b9ab24df0040eb49912eac10cd0",
    "status": "completed"
  }
]
```

Agent Class

- Why name, instructions and other attributes are showing string within single quote instead of just string in Agent class?



```
Agent(self, name: 'str', instructions: 'str |
Callable[[RunContextWrapper[TContext], Agent[TContext]],
MaybeAwaitable[str]] | None' = None, prompt: 'Prompt |
DynamicPromptFunction | None' = None, handoff_description: 'str
| None' = None, handoffs: 'list[Agent[Any] |
Handoff[TContext]]' = dataclasses._HAS_DEFAULT_FACTORY_CLASS
instance, model: 'str | Model | None' = None, model_settings:
'ModelSettings' = dataclasses._HAS_DEFAULT_FACTORY_CLASS
instance, tools: 'list[Tool]' =
dataclasses._HAS_DEFAULT_FACTORY_CLASS instance, mcp_servers:
'list[MCPServer]' = dataclasses._HAS_DEFAULT_FACTORY_CLASS
instance, mcp_config: 'MCPConfig' =
```

Agent Class

- The reason you see types like 'str' (i.e. a string in single quotes) instead of just str in the Agent class signature is likely due to the use of **forward references** in the type annotations.

Why 'str' Instead of str?

- In Python, when defining a class or function signature, if the type you're referencing hasn't been fully defined or imported yet, you can use a string literal as a forward reference. This tells Python: "Interpret this string as a type later, when the full context is available."
- In your example: ➔ name: 'str’
- It’s functionally the same as: ➔ name: str
- But using 'str' delays the evaluation of the type annotation — it avoids errors in environments like dynamic loading, circular imports, or partial class evaluation (e.g., during static analysis or doc generation).

Agent Class

- When we give agent name as an **integer** or **None**, it still run but in Agent class it is showing name type as ‘**str**’, that we need to explore more.

Agent Class

- Instruction are the persona of Agent
- LLM is a brain of Agent
- Selection of right model of LLM is very important
- Agent uses tools to achieve its tasks

Context

- 3 types of object we can used to make context
  - By using @dataclass decorator
  - By using Pydantic
  - Normal class with attributes, attributes can used as a context and you have to manage them as run time

Output Type & Dynamic Instructions

Output Type:

- Due to output type or structure type, the output received from agent, we can integrate that output with existing API
- We normally used Pydantic base model to declare output types

Dynamic Instructions:

- In most cases, you can provide instructions when you create the agent. However, you can also provide dynamic instructions via a function. The function will receive the agent and context, and must return the prompt. Both regular and async functions are accepted.

Context

Context is an overloaded term. There are two main classes of context you might care about:

- Context available locally to your code: this is data and dependencies you might need when tool functions run, during callbacks like on_handoff, in lifecycle hooks, etc.
- Context available to LLMs: this is data the LLM sees when generating a response.

- **Local context**
- This is represented via the RunContextWrapper class and the context property within it.

Handoff

- Handoffs allow an agent to delegate tasks to another agent. This is particularly useful in scenarios where different agents specialize in distinct areas. For example, a customer support app might have agents that each specifically handle tasks like order status, refunds, FAQs, etc.
- Handoffs are represented as tools to the LLM. So, if there's a handoff to an agent named Refund Agent, the tool would be called **transfer_to_refund_agent**.

Custom Runner

- We can make custom Runner by using AgentRunner class
- Import AgentRunner class from agents.run
- Define custom runner class in this way.

```
class CustomAgentRunner(AgentRunner):
    async def run (self, starting_agent:Agent, input:str, **kwargs):
        # Custom Preprocessing
        print(f"CustomAgentRunner.run()")
        # input = await self.preprocess(input)


        # Call parent with custom logic
        result = await super().run(starting_agent, input, **kwargs)


        # Custom Postprocessing and analytics
        # await self.log_analytics(result)
        return result
```

Orchestrating multiple agents

- Orchestration refers to the flow of agents in your app. Which agents run, in what order, and how do they decide what happens next? There are two main ways to orchestrate agents:
    1) Allowing the LLM to make decisions: this uses the intelligence of an LLM to plan, reason, and decide on what steps to take based on that.
    2) Orchestrating via code: determining the flow of agents via your code.
- Check this link for more example which we covered in this session
- https://github.com/openai/openai-agents-python/tree/main/examples/agent_patterns