ChatGPT

Glossary of OpenAI Agent SDK (Healthcare Tutorial)

Agent

A core class in the OpenAI Agent SDK representing an AI agent (powered by a large language model, LLM). An Agent encapsulates a model, instructions (prompt), and optionally tools. It can produce responses directly or invoke tools to assist in generating structured outputs [1].

Agent-as-Tools Pattern

A design pattern where an agent can use other specialized sub-agents as tools. In this approach, a "parent" agent calls child agents as if they were functions, enabling modular task delegation and flexible orchestration [2].

async

A keyword in Python used to define **coroutine** functions. Coroutines declared with the `async/await` syntax allow writing concurrent code using the `asyncio` library. This is the preferred way to write asynchronous applications in Python [3].

asyncio.gather

An `asyncio` function that runs multiple awaitable objects concurrently and waits for all of them to complete. It takes in a sequence of coroutines/Tasks and returns their results in a list once all have finished executing [4]. This is commonly used to implement parallel agent or tool execution.

await

A Python keyword used inside an `async` function to pause execution until an awaitable (like a coroutine or Future) is resolved. Using `await` on a coroutine yields control back to the event loop until the coroutine finishes, allowing other tasks to run in the meantime [3].

@function_tool

A decorator provided by the Agent SDK to turn a Python function into a tool that an agent can call. When a function is annotated with `@function_tool`, the SDK exposes it (with its name, arguments, and docstring) to the LLM, enabling the agent to invoke it via function calling [5].

CodeInterpreterTool

A built-in tool offered by OpenAI that allows an agent to execute Python code in a sandboxed environment. It's often used for data analysis, generating charts, or transforming data within an agent's workflow [6]. Using this tool, an agent can run code to produce results (for example, plotting a graph or performing calculations) which are then returned to the conversation.

Completion

In the context of LLM APIs, a *completion* is the model-generated output given a prompt. The user provides an input prompt (which may include system/user instructions), and the model responds with a text completion – essentially the continuation or answer that the model produces [7]. In chat-based APIs, this corresponds to the assistant's reply.

context

A keyword argument in the Agent SDK's `Runner.run()` method that supplies background data or state to an agent run. The context is typically a structured object (e.g. a Python dataclass) containing information the agent or its tools might need while executing. Tools can access this `context` during their function calls [8]. (This is distinct from "context" in prompt engineering, which refers to the surrounding text given to the model.)

Dataclass

In Python, a `@dataclass` is a decorator that automatically generates special methods (like `__init__`) for classes, making it convenient to define simple data structures. Dataclasses are used in the Agent SDK to define input/output schemas for tools or agents, because they provide a clear, type-annotated way to structure data [9]. For example, you might use a dataclass to define the expected fields of a patient record that an agent will output.

Deterministic Workflow

An agent orchestration approach where the sequence of steps is fixed in code, rather than decided dynamically by the AI. In a deterministic workflow (or pipeline), each agent or function runs in a predefined order with predetermined inputs/outputs. This predictability makes the automation easier to trace and debug [10]. The healthcare tutorial evolves a simple deterministic sequence into more complex pipelines while still maintaining a fixed overall flow.

final_output

The primary result produced by an agent after a run, cast to the specified `output_type`. This is essentially the agent's answer or output in its final form (for instance, a string message or a dataclass object). The `RunResult` of an agent contains `final_output` as the ready-to-use result of that agent's execution [11].

Function Calling

A capability of OpenAI's LLMs that allows them to invoke developer-defined functions (tools) during a conversation. Function calling works by letting you describe functions (their name, parameters, etc.) to the model; the model can then decide to output a JSON indicating a function name and arguments when appropriate [12]. The SDK leverages this so that an agent can **call tools** to handle specific tasks (like database queries, calculations, API calls) and then incorporate the results back into its response.

MCPServer / MCPServerSse

"MCP" stands for **Model Context Protocol**, an open standard (pioneered by Anthropic) for connecting AI models to external data and services. In the Agent SDK, `MCPServer` (such as `MCPServerSse`) is used to interface with an MCP endpoint. Practically, an MCPServer tool lets an agent send or receive data via the MCP – for example, using a Zapier webhook as a server to perform an action like sending an email [13]. In the tutorial, an `MCPServerSse` is configured to trigger an external email-sending service, demonstrating how agents can effect real-world actions through MCP.

ModelSettings

A configuration object in the Agent SDK that specifies how the model should behave or which model features to enforce. `ModelSettings` can include parameters like `tool_choice` (to require or allow skipping tool use) or specify a particular model variant/settings for the agent [14]. For example, setting `tool_choice="required"` in `ModelSettings` ensures the agent must use a tool if one is available, rather than answering from the model alone.

new_items

Part of an agent's `RunResult` that captures any new messages or intermediate content generated during the run (especially in multi-step or tool-using agents). For instance, if an agent call results in multiple assistant messages (perhaps one being a tool invocation and another the final answer), those would appear in `new_items`. It's a way to inspect the raw chain of outputs/actions that occurred besides the final result [15].

Output Parsing

The process of converting the raw output from the LLM into a structured format as defined by the agent's `output_type`. The Agent SDK automatically attempts to parse the LLM's response (often JSON or code block content) into Python objects or dataclasses. If a strict schema is enforced, the parser will validate and ensure the output matches the expected format exactly [16]. For example, if `output_type` is a dataclass with specific fields, output parsing will try to parse the model's text into that dataclass, possibly raising errors or correcting format issues if the model's response doesn't comply.

output_type

A parameter defining the expected type/format of an agent's final output. This can be a Python type like `str`, a dataclass, or a Pydantic model, and it tells the SDK how to interpret and cast the model's response. For instance, `output_type=dict` would indicate the agent should return a dictionary (likely by outputting JSON that gets parsed into a dict). Using `output_type` helps the SDK to parse and validate the model's output against an expected schema [17].

Parallel Execution Pattern

An orchestration pattern where multiple agents or tools run at the same time (concurrently) rather than sequentially. In implementation, this often uses `asyncio.gather()` or similar concurrency primitives to launch tasks in parallel [18]. The tutorial mentions this as a way to speed up workflows – for example, running two data analysis agents simultaneously on different parts of a problem, then combining their results. Parallel execution must be managed carefully with `async` / `await` to collect all results once done.

Prompt

A prompt is the input given to an LLM, consisting of instructions or a question that we want the model to respond to. In chat models, a prompt often includes a **system message** (to set behavior) and a **user message** (the query or task). The model then generates a **completion**. In other words, the prompt is everything fed into the model to elicit a response [7]. Crafting a good prompt (clear instructions, necessary context) is crucial for getting useful outputs from the model.

Pydantic

A Python library used for defining data models with validation. In the context of the Agent SDK, Pydantic models can serve as complex `output_type` or tool output schemas. Pydantic ensures that data conforms to types/constraints and can be easily converted from JSON. By using Pydantic (or dataclasses) for agent outputs and tool returns, developers get a guarantee (or at least a validation mechanism) that the LLM's output matches the expected format [19].

raw_item / raw_response

The low-level response data from the model, as opposed to the processed final output. In the Agent SDK, `raw_item` (or `raw_response`) is often an object that contains the full detail of what the model returned, including the message content and any function call arguments. It's useful for debugging or inspecting exactly what the model said or did. For example, if an agent uses a tool, the raw response might show the function call and arguments the model requested [20].

RunContextWrapper

A utility class in the Agent SDK that wraps tool call arguments along with the run context. When a function decorated with `@function_tool` is called by the agent, if the function signature expects a special `RunContextWrapper`, the SDK will pass it in. This wrapper provides access to both the invocation context

and the tool's parameters within the function, enabling advanced use like modifying the agent's state or logging [21]. In simpler terms, it's a helper to get extra info inside tool functions when needed.

Runner.run()

The primary method to execute an agent. `Runner.run(agent, input, context=...)` will asynchronously run the given agent with the provided input (and optional context). It returns a `RunResult` which contains the agent's outputs. Under the hood, this handles sending the prompt to the model, managing tool invocations (if any), and assembling the final result [22] [23]. There is also a `Runner.run_sync()` for synchronous execution. In the tutorial's code, `Runner.run()` is used to chain the agents in sequence, passing the outputs of one as inputs to the next.

RunResult

The object returned when you run an agent (via `Runner.run`). It contains various fields capturing the outcome of the run: notably `final_output` (the agent's final answer in the specified type), `new_items` (any intermediate messages or actions), and possibly metadata like `raw_response`. Essentially, `RunResult` is a structured record of everything the agent did and produced during a single run [24].

strict schema (output_schema_strict)

A setting that controls how strictly the agent enforces the output schema. When strict schema mode is on (`output_schema_strict=True` by default), the agent SDK will treat any deviation from the expected `output_type` format as an error – for example, malformed JSON or missing fields will cause a failure or retry. If turned off, the agent may be allowed to return outputs that don't perfectly match the schema (the SDK will try to coerce or ignore errors) [16]. Toggling this can help when the model has minor formatting issues – relaxing it (`False`) can make the system more tolerant, at the cost of weaker guarantees on structure.

System Message

A special initial message in a chat prompt that sets the context, instructions, or persona for the AI. The system message is not from the user but from the developer – it **guides the model's behavior**. For example, a system message might say, "You are a polite medical assistant." This influences all subsequent answers. In the API, the system message is prepended to the conversation and is used to specify the role or style the model should adopt [25]. (Contrast with the user message, which actually asks the question or gives the task.)

ToolChoice

A setting (often part of `ModelSettings`) that determines whether an agent is required to use a tool or can answer directly. Common values are `"auto"` (let the model decide when to use tools) or `"required"` (the model **must** call a tool if one is available, rather than responding from its own knowledge) [26]. For instance, in the tutorial's enhanced agent, `tool_choice="required"` ensures the agent uses the `synthesize_report` tool instead of free-forming an answer.

Tool Context

When an agent invokes a tool, the SDK provides the tool with a context object (sometimes wrapped in `RunContextWrapper`). This **Tool Context** may include the current `context` data passed to the agent and other metadata about the run. It allows the tool function to know about the environment in which it's called. In most cases, you don't need to handle this explicitly – the SDK manages passing context to tools automatically (usually via the `RunContextWrapper` as needed) [27].

Tool Output Schema

The expected structure of the data that a tool returns. Since tools in the Agent SDK are normal Python functions, their return type (possibly a dataclass or Pydantic model) defines the tool's output schema. The SDK will serialize the tool's result (e.g. into JSON) to pass it back to the LLM. Defining a strict tool output schema ensures the model receives reliable, structured information from the tool [28]. For example, a tool that returns a `WeatherReport` dataclass will produce a JSON like `{"temperature": ..., "condition": ...}` – the agent's prompt can then incorporate those fields easily.

Tool Use

The act of an agent leveraging an external function or API during its reasoning process. A "tool" in this context is a function (like those decorated with `@function_tool`) or built-in capability (web search, code execution, etc.) that the agent can call via the function calling mechanism. Tool use allows the agent to extend its abilities beyond the base model – for example, by calling a calculator function, querying a database, or sending an email. In prompt terms, the agent decides to output a function call, which the SDK executes, and then the agent uses the function's result to continue the conversation [29]. Tool use is crucial for creating agents that can interact with real-world data or perform actions.

Further Reading

- **OpenAI Blog – *New Tools for Building Agents* (Mar 2025):** Official announcement and overview of the Agents SDK, built-in tools (web search, code interpreter, etc.), and orchestration features [30] [31]. *(Great for understanding the big picture and OpenAI's vision for agentic AI.)*
- **Towards AI – *OpenAI's Agent SDK: A Comprehensive Guide* (2025):** A step-by-step tutorial on setting up the Agents SDK, building agents with tools, and best practices for organizing agent projects. Provides hands-on examples (Hello World agent, weather tool, etc.) in a beginner-friendly manner.
- **Humanloop Blog – *OpenAI Agent SDK Overview*:** An explanatory article covering the key concepts of the Agent SDK (agents, tools, runs, guardrails) and how it differs from prior approaches. Helps reinforce core terminology and workflow with simple examples.
- **OpenAI Help Center – *Best Practices for Prompt Engineering*:** Practical tips from OpenAI on crafting effective prompts and system messages. Covers strategies like providing clear instructions, using examples, and refining outputs. Useful for designing the system/user prompts for your agents.
- **OpenAI Cookbook – *Evaluating Agents with Langfuse*:** Demonstrates how to instrument agents for tracing and evaluation. While more advanced, this guide shows how to monitor agent reasoning steps and assess performance using the Langfuse tool, which can be insightful as you build more complex agent workflows.

---

1 2 5 6 8 9 10 11 13 14 15 16 17 18 19 20 21 22 24 26 27 28
OpenAI_Agent_SDK_Glossary_Expanded.pdf
file://file-Cd2KiVCjHumbuGWdUiCjPY

3 4 Coroutines and Tasks — Python 3.13.5 documentation
https://docs.python.org/3/library/asyncio-task.html

7 Azure OpenAI in Azure AI Foundry Models - Azure OpenAI | Microsoft Learn
https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/prompt-engineering

12 OpenAI NodeJS Documentation
https://www.dynamikapps.com/openai-nodejs-documentation/

23 29 Healthcare Patient Journey – Deterministic Agent Workflows.pdf
file://file-LMqLGAbN187i7kQjRYJeeT

25 OpenAI GPT Prompts | PushMetrics
https://pushmetrics.io/docs/executable-tasks/openai-gpt-prompts/

30 31 New tools for building agents | OpenAI
https://openai.com/index/new-tools-for-building-agents/