Table of Contents

# Agents

Agents are the core building block in your apps. An agent is a large language model (LLM), configured with instructions and tools.

## Basic configuration

The most common properties of an agent you’ll configure are:

* instructions: also known as a developer message or system prompt.
* model: which LLM to use, and optional model\_settings to configure model tuning parameters like temperature, top\_p, etc.
* tools: Tools that the agent can use to achieve its tasks.

from agents import Agent, ModelSettings, function\_tool  
  
@function\_tool  
def get\_weather(city: str) -> str:  
 return f"The weather in {city} is sunny"  
  
agent = Agent(  
 name="Haiku agent",  
 instructions="Always respond in haiku form",  
 model="o3-mini",  
 tools=[get\_weather],  
)

## Context

Agents are generic on their context type. Context is a dependency-injection tool: it’s an object you create and pass to Runner.run(), that is passed to every agent, tool, handoff etc, and it serves as a grab bag of dependencies and state for the agent run. You can provide any Python object as the context.

@dataclass  
class UserContext:  
 uid: str  
 is\_pro\_user: bool  
  
 async def fetch\_purchases() -> list[Purchase]:  
 return ...  
  
agent = Agent[UserContext](  
 ...,  
)

## Output types

By default, agents produce plain text (i.e. str) outputs. If you want the agent to produce a particular type of output, you can use the output\_type parameter. A common choice is to use [Pydantic](https://docs.pydantic.dev/) objects, but we support any type that can be wrapped in a Pydantic [TypeAdapter](https://docs.pydantic.dev/latest/api/type_adapter/) - dataclasses, lists, TypedDict, etc.

from pydantic import BaseModel  
from agents import Agent  
  
  
class CalendarEvent(BaseModel):  
 name: str  
 date: str  
 participants: list[str]  
  
agent = Agent(  
 name="Calendar extractor",  
 instructions="Extract calendar events from text",  
 output\_type=CalendarEvent,  
)

!!! note

When you pass an `output\_type`, that tells the model to use [structured outputs](https://platform.openai.com/docs/guides/structured-outputs) instead of regular plain text responses.

## Handoffs

Handoffs are sub-agents that the agent can delegate to. You provide a list of handoffs, and the agent can choose to delegate to them if relevant. This is a powerful pattern that allows orchestrating modular, specialized agents that excel at a single task. Read more in the [handoffs](handoffs.md) documentation.

from agents import Agent  
  
booking\_agent = Agent(...)  
refund\_agent = Agent(...)  
  
triage\_agent = Agent(  
 name="Triage agent",  
 instructions=(  
 "Help the user with their questions."  
 "If they ask about booking, handoff to the booking agent."  
 "If they ask about refunds, handoff to the refund agent."  
 ),  
 handoffs=[booking\_agent, refund\_agent],  
)

## 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.

def dynamic\_instructions(  
 context: RunContextWrapper[UserContext], agent: Agent[UserContext]  
) -> str:  
 return f"The user's name is {context.context.name}. Help them with their questions."  
  
  
agent = Agent[UserContext](  
 name="Triage agent",  
 instructions=dynamic\_instructions,  
)

## Lifecycle events (hooks)

Sometimes, you want to observe the lifecycle of an agent. For example, you may want to log events, or pre-fetch data when certain events occur. You can hook into the agent lifecycle with the hooks property. Subclass the [AgentHooks][agents.lifecycle.AgentHooks] class, and override the methods you’re interested in.

## Guardrails

Guardrails allow you to run checks/validations on user input, in parallel to the agent running. For example, you could screen the user’s input for relevance. Read more in the [guardrails](guardrails.md) documentation.

## Cloning/copying agents

By using the clone() method on an agent, you can duplicate an Agent, and optionally change any properties you like.

pirate\_agent = Agent(  
 name="Pirate",  
 instructions="Write like a pirate",  
 model="o3-mini",  
)  
  
robot\_agent = pirate\_agent.clone(  
 name="Robot",  
 instructions="Write like a robot",  
)

## Forcing tool use

Supplying a list of tools doesn’t always mean the LLM will use a tool. You can force tool use by setting [ModelSettings.tool\_choice][agents.model\_settings.ModelSettings.tool\_choice]. Valid values are:

1. auto, which allows the LLM to decide whether or not to use a tool.
2. required, which requires the LLM to use a tool (but it can intelligently decide which tool).
3. none, which requires the LLM to *not* use a tool.
4. Setting a specific string e.g. my\_tool, which requires the LLM to use that specific tool.

!!! note

To prevent infinite loops, the framework automatically resets `tool\_choice` to "auto" after a tool call. This behavior is configurable via [`agent.reset\_tool\_choice`][agents.agent.Agent.reset\_tool\_choice]. The infinite loop is because tool results are sent to the LLM, which then generates another tool call because of `tool\_choice`, ad infinitum.  
  
If you want the Agent to completely stop after a tool call (rather than continuing with auto mode), you can set [`Agent.tool\_use\_behavior="stop\_on\_first\_tool"`] which will directly use the tool output as the final response without further LLM processing.

# Configuring the SDK

## API keys and clients

By default, the SDK looks for the OPENAI\_API\_KEY environment variable for LLM requests and tracing, as soon as it is imported. If you are unable to set that environment variable before your app starts, you can use the [set\_default\_openai\_key()][agents.set\_default\_openai\_key] function to set the key.

from agents import set\_default\_openai\_key  
  
set\_default\_openai\_key("sk-...")

Alternatively, you can also configure an OpenAI client to be used. By default, the SDK creates an AsyncOpenAI instance, using the API key from the environment variable or the default key set above. You can change this by using the [set\_default\_openai\_client()][agents.set\_default\_openai\_client] function.

from openai import AsyncOpenAI  
from agents import set\_default\_openai\_client  
  
custom\_client = AsyncOpenAI(base\_url="...", api\_key="...")  
set\_default\_openai\_client(custom\_client)

Finally, you can also customize the OpenAI API that is used. By default, we use the OpenAI Responses API. You can override this to use the Chat Completions API by using the [set\_default\_openai\_api()][agents.set\_default\_openai\_api] function.

from agents import set\_default\_openai\_api  
  
set\_default\_openai\_api("chat\_completions")

## Tracing

Tracing is enabled by default. It uses the OpenAI API keys from the section above by default (i.e. the environment variable or the default key you set). You can specifically set the API key used for tracing by using the [set\_tracing\_export\_api\_key][agents.set\_tracing\_export\_api\_key] function.

from agents import set\_tracing\_export\_api\_key  
  
set\_tracing\_export\_api\_key("sk-...")

You can also disable tracing entirely by using the [set\_tracing\_disabled()][agents.set\_tracing\_disabled] function.

from agents import set\_tracing\_disabled  
  
set\_tracing\_disabled(True)

## Debug logging

The SDK has two Python loggers without any handlers set. By default, this means that warnings and errors are sent to stdout, but other logs are suppressed.

To enable verbose logging, use the [enable\_verbose\_stdout\_logging()][agents.enable\_verbose\_stdout\_logging] function.

from agents import enable\_verbose\_stdout\_logging  
  
enable\_verbose\_stdout\_logging()

Alternatively, you can customize the logs by adding handlers, filters, formatters, etc. You can read more in the [Python logging guide](https://docs.python.org/3/howto/logging.html).

import logging  
  
logger = logging.getLogger("openai.agents") # or openai.agents.tracing for the Tracing logger  
  
# To make all logs show up  
logger.setLevel(logging.DEBUG)  
# To make info and above show up  
logger.setLevel(logging.INFO)  
# To make warning and above show up  
logger.setLevel(logging.WARNING)  
# etc  
  
# You can customize this as needed, but this will output to `stderr` by default  
logger.addHandler(logging.StreamHandler())

### Sensitive data in logs

Certain logs may contain sensitive data (for example, user data). If you want to disable this data from being logged, set the following environment variables.

To disable logging LLM inputs and outputs:

export OPENAI\_AGENTS\_DONT\_LOG\_MODEL\_DATA=1

To disable logging tool inputs and outputs:

export OPENAI\_AGENTS\_DONT\_LOG\_TOOL\_DATA=1

# Context management

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

1. 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.
2. Context available to LLMs: this is data the LLM sees when generating a response.

## Local context

This is represented via the [RunContextWrapper][agents.run\_context.RunContextWrapper] class and the [context][agents.run\_context.RunContextWrapper.context] property within it. The way this works is:

1. You create any Python object you want. A common pattern is to use a dataclass or a Pydantic object.
2. You pass that object to the various run methods (e.g. Runner.run(..., \*\*context=whatever\*\*)).
3. All your tool calls, lifecycle hooks etc will be passed a wrapper object, RunContextWrapper[T], where T represents your context object type which you can access via wrapper.context.

The **most important** thing to be aware of: every agent, tool function, lifecycle etc for a given agent run must use the same *type* of context.

You can use the context for things like:

* Contextual data for your run (e.g. things like a username/uid or other information about the user)
* Dependencies (e.g. logger objects, data fetchers, etc)
* Helper functions

!!! danger “Note”

The context object is \*\*not\*\* sent to the LLM. It is purely a local object that you can read from, write to and call methods on it.

import asyncio  
from dataclasses import dataclass  
  
from agents import Agent, RunContextWrapper, Runner, function\_tool  
  
@dataclass  
class UserInfo: # (1)!  
 name: str  
 uid: int  
  
@function\_tool  
async def fetch\_user\_age(wrapper: RunContextWrapper[UserInfo]) -> str: # (2)!  
 return f"User {wrapper.context.name} is 47 years old"  
  
async def main():  
 user\_info = UserInfo(name="John", uid=123)  
  
 agent = Agent[UserInfo]( # (3)!  
 name="Assistant",  
 tools=[fetch\_user\_age],  
 )  
  
 result = await Runner.run( # (4)!  
 starting\_agent=agent,  
 input="What is the age of the user?",  
 context=user\_info,  
 )  
  
 print(result.final\_output) # (5)!  
 # The user John is 47 years old.  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 asyncio.run(main())

1. This is the context object. We’ve used a dataclass here, but you can use any type.
2. This is a tool. You can see it takes a RunContextWrapper[UserInfo]. The tool implementation reads from the context.
3. We mark the agent with the generic UserInfo, so that the typechecker can catch errors (for example, if we tried to pass a tool that took a different context type).
4. The context is passed to the run function.
5. The agent correctly calls the tool and gets the age.

## Agent/LLM context

When an LLM is called, the **only** data it can see is from the conversation history. This means that if you want to make some new data available to the LLM, you must do it in a way that makes it available in that history. There are a few ways to do this:

1. You can add it to the Agent instructions. This is also known as a “system prompt” or “developer message”. System prompts can be static strings, or they can be dynamic functions that receive the context and output a string. This is a common tactic for information that is always useful (for example, the user’s name or the current date).
2. Add it to the input when calling the Runner.run functions. This is similar to the instructions tactic, but allows you to have messages that are lower in the [chain of command](https://cdn.openai.com/spec/model-spec-2024-05-08.html#follow-the-chain-of-command).
3. Expose it via function tools. This is useful for *on-demand* context - the LLM decides when it needs some data, and can call the tool to fetch that data.
4. Use retrieval or web search. These are special tools that are able to fetch relevant data from files or databases (retrieval), or from the web (web search). This is useful for “grounding” the response in relevant contextual data. # Examples

Check out a variety of sample implementations of the SDK in the examples section of the [repo](https://github.com/openai/openai-agents-python/tree/main/examples). The examples are organized into several categories that demonstrate different patterns and capabilities.

## Categories

* [**agent\_patterns**](https://github.com/openai/openai-agents-python/tree/main/examples/agent_patterns)**:** Examples in this category illustrate common agent design patterns, such as
  + Deterministic workflows
  + Agents as tools
  + Parallel agent execution
* [**basic**](https://github.com/openai/openai-agents-python/tree/main/examples/basic)**:** These examples showcase foundational capabilities of the SDK, such as
  + Dynamic system prompts
  + Streaming outputs
  + Lifecycle events
* [**tool examples**](https://github.com/openai/openai-agents-python/tree/main/examples/tools)**:** Learn how to implement OAI hosted tools such as web search and file search, and integrate them into your agents.
* [**model providers**](https://github.com/openai/openai-agents-python/tree/main/examples/model_providers)**:** Explore how to use non-OpenAI models with the SDK.
* [**handoffs**](https://github.com/openai/openai-agents-python/tree/main/examples/handoffs)**:** See practical examples of agent handoffs.
* [**mcp**](https://github.com/openai/openai-agents-python/tree/main/examples/mcp)**:** Learn how to build agents with MCP.
* [**customer\_service**](https://github.com/openai/openai-agents-python/tree/main/examples/customer_service) and [**research\_bot**](https://github.com/openai/openai-agents-python/tree/main/examples/research_bot)**:** Two more built-out examples that illustrate real-world applications
  + **customer\_service**: Example customer service system for an airline.
  + **research\_bot**: Simple deep research clone.
* [**voice**](https://github.com/openai/openai-agents-python/tree/main/examples/voice)**:** See examples of voice agents, using our TTS and STT models. # Guardrails

Guardrails run *in parallel* to your agents, enabling you to do checks and validations of user input. For example, imagine you have an agent that uses a very smart (and hence slow/expensive) model to help with customer requests. You wouldn’t want malicious users to ask the model to help them with their math homework. So, you can run a guardrail with a fast/cheap model. If the guardrail detects malicious usage, it can immediately raise an error, which stops the expensive model from running and saves you time/money.

There are two kinds of guardrails:

1. Input guardrails run on the initial user input
2. Output guardrails run on the final agent output

## Input guardrails

Input guardrails run in 3 steps:

1. First, the guardrail receives the same input passed to the agent.
2. Next, the guardrail function runs to produce a [GuardrailFunctionOutput][agents.guardrail.GuardrailFunctionOutput], which is then wrapped in an [InputGuardrailResult][agents.guardrail.InputGuardrailResult]
3. Finally, we check if [.tripwire\_triggered][agents.guardrail.GuardrailFunctionOutput.tripwire\_triggered] is true. If true, an [InputGuardrailTripwireTriggered][agents.exceptions.InputGuardrailTripwireTriggered] exception is raised, so you can appropriately respond to the user or handle the exception.

!!! Note

Input guardrails are intended to run on user input, so an agent's guardrails only run if the agent is the \*first\* agent. You might wonder, why is the `guardrails` property on the agent instead of passed to `Runner.run`? It's because guardrails tend to be related to the actual Agent - you'd run different guardrails for different agents, so colocating the code is useful for readability.

## Output guardrails

Output guardrails run in 3 steps:

1. First, the guardrail receives the same input passed to the agent.
2. Next, the guardrail function runs to produce a [GuardrailFunctionOutput][agents.guardrail.GuardrailFunctionOutput], which is then wrapped in an [OutputGuardrailResult][agents.guardrail.OutputGuardrailResult]
3. Finally, we check if [.tripwire\_triggered][agents.guardrail.GuardrailFunctionOutput.tripwire\_triggered] is true. If true, an [OutputGuardrailTripwireTriggered][agents.exceptions.OutputGuardrailTripwireTriggered] exception is raised, so you can appropriately respond to the user or handle the exception.

!!! Note

Output guardrails are intended to run on the final agent output, so an agent's guardrails only run if the agent is the \*last\* agent. Similar to the input guardrails, we do this because guardrails tend to be related to the actual Agent - you'd run different guardrails for different agents, so colocating the code is useful for readability.

## Tripwires

If the input or output fails the guardrail, the Guardrail can signal this with a tripwire. As soon as we see a guardrail that has triggered the tripwires, we immediately raise a {Input,Output}GuardrailTripwireTriggered exception and halt the Agent execution.

## Implementing a guardrail

You need to provide a function that receives input, and returns a [GuardrailFunctionOutput][agents.guardrail.GuardrailFunctionOutput]. In this example, we’ll do this by running an Agent under the hood.

from pydantic import BaseModel  
from agents import (  
 Agent,  
 GuardrailFunctionOutput,  
 InputGuardrailTripwireTriggered,  
 RunContextWrapper,  
 Runner,  
 TResponseInputItem,  
 input\_guardrail,  
)  
  
class MathHomeworkOutput(BaseModel):  
 is\_math\_homework: bool  
 reasoning: str  
  
guardrail\_agent = Agent( # (1)!  
 name="Guardrail check",  
 instructions="Check if the user is asking you to do their math homework.",  
 output\_type=MathHomeworkOutput,  
)  
  
  
@input\_guardrail  
async def math\_guardrail( # (2)!  
 ctx: RunContextWrapper[None], agent: Agent, input: str | list[TResponseInputItem]  
) -> GuardrailFunctionOutput:  
 result = await Runner.run(guardrail\_agent, input, context=ctx.context)  
  
 return GuardrailFunctionOutput(  
 output\_info=result.final\_output, # (3)!  
 tripwire\_triggered=result.final\_output.is\_math\_homework,  
 )  
  
  
agent = Agent( # (4)!  
 name="Customer support agent",  
 instructions="You are a customer support agent. You help customers with their questions.",  
 input\_guardrails=[math\_guardrail],  
)  
  
async def main():  
 # This should trip the guardrail  
 try:  
 await Runner.run(agent, "Hello, can you help me solve for x: 2x + 3 = 11?")  
 print("Guardrail didn't trip - this is unexpected")  
  
 except InputGuardrailTripwireTriggered:  
 print("Math homework guardrail tripped")

1. We’ll use this agent in our guardrail function.
2. This is the guardrail function that receives the agent’s input/context, and returns the result.
3. We can include extra information in the guardrail result.
4. This is the actual agent that defines the workflow.

Output guardrails are similar.

from pydantic import BaseModel  
from agents import (  
 Agent,  
 GuardrailFunctionOutput,  
 OutputGuardrailTripwireTriggered,  
 RunContextWrapper,  
 Runner,  
 output\_guardrail,  
)  
class MessageOutput(BaseModel): # (1)!  
 response: str  
  
class MathOutput(BaseModel): # (2)!  
 reasoning: str  
 is\_math: bool  
  
guardrail\_agent = Agent(  
 name="Guardrail check",  
 instructions="Check if the output includes any math.",  
 output\_type=MathOutput,  
)  
  
@output\_guardrail  
async def math\_guardrail( # (3)!  
 ctx: RunContextWrapper, agent: Agent, output: MessageOutput  
) -> GuardrailFunctionOutput:  
 result = await Runner.run(guardrail\_agent, output.response, context=ctx.context)  
  
 return GuardrailFunctionOutput(  
 output\_info=result.final\_output,  
 tripwire\_triggered=result.final\_output.is\_math,  
 )  
  
agent = Agent( # (4)!  
 name="Customer support agent",  
 instructions="You are a customer support agent. You help customers with their questions.",  
 output\_guardrails=[math\_guardrail],  
 output\_type=MessageOutput,  
)  
  
async def main():  
 # This should trip the guardrail  
 try:  
 await Runner.run(agent, "Hello, can you help me solve for x: 2x + 3 = 11?")  
 print("Guardrail didn't trip - this is unexpected")  
  
 except OutputGuardrailTripwireTriggered:  
 print("Math output guardrail tripped")

1. This is the actual agent’s output type.
2. This is the guardrail’s output type.
3. This is the guardrail function that receives the agent’s output, and returns the result.
4. This is the actual agent that defines the workflow. # Handoffs

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.

## Creating a handoff

All agents have a [handoffs][agents.agent.Agent.handoffs] param, which can either take an Agent directly, or a Handoff object that customizes the Handoff.

You can create a handoff using the [handoff()][agents.handoffs.handoff] function provided by the Agents SDK. This function allows you to specify the agent to hand off to, along with optional overrides and input filters.

### Basic Usage

Here’s how you can create a simple handoff:

from agents import Agent, handoff  
  
billing\_agent = Agent(name="Billing agent")  
refund\_agent = Agent(name="Refund agent")  
  
# (1)!  
triage\_agent = Agent(name="Triage agent", handoffs=[billing\_agent, handoff(refund\_agent)])

1. You can use the agent directly (as in billing\_agent), or you can use the handoff() function.

### Customizing handoffs via the handoff() function

The [handoff()][agents.handoffs.handoff] function lets you customize things.

* agent: This is the agent to which things will be handed off.
* tool\_name\_override: By default, the Handoff.default\_tool\_name() function is used, which resolves to transfer\_to\_<agent\_name>. You can override this.
* tool\_description\_override: Override the default tool description from Handoff.default\_tool\_description()
* on\_handoff: A callback function executed when the handoff is invoked. This is useful for things like kicking off some data fetching as soon as you know a handoff is being invoked. This function receives the agent context, and can optionally also receive LLM generated input. The input data is controlled by the input\_type param.
* input\_type: The type of input expected by the handoff (optional).
* input\_filter: This lets you filter the input received by the next agent. See below for more.

from agents import Agent, handoff, RunContextWrapper  
  
def on\_handoff(ctx: RunContextWrapper[None]):  
 print("Handoff called")  
  
agent = Agent(name="My agent")  
  
handoff\_obj = handoff(  
 agent=agent,  
 on\_handoff=on\_handoff,  
 tool\_name\_override="custom\_handoff\_tool",  
 tool\_description\_override="Custom description",  
)

## Handoff inputs

In certain situations, you want the LLM to provide some data when it calls a handoff. For example, imagine a handoff to an “Escalation agent”. You might want a reason to be provided, so you can log it.

from pydantic import BaseModel  
  
from agents import Agent, handoff, RunContextWrapper  
  
class EscalationData(BaseModel):  
 reason: str  
  
async def on\_handoff(ctx: RunContextWrapper[None], input\_data: EscalationData):  
 print(f"Escalation agent called with reason: {input\_data.reason}")  
  
agent = Agent(name="Escalation agent")  
  
handoff\_obj = handoff(  
 agent=agent,  
 on\_handoff=on\_handoff,  
 input\_type=EscalationData,  
)

## Input filters

When a handoff occurs, it’s as though the new agent takes over the conversation, and gets to see the entire previous conversation history. If you want to change this, you can set an [input\_filter][agents.handoffs.Handoff.input\_filter]. An input filter is a function that receives the existing input via a [HandoffInputData][agents.handoffs.HandoffInputData], and must return a new HandoffInputData.

There are some common patterns (for example removing all tool calls from the history), which are implemented for you in [agents.extensions.handoff\_filters][]

from agents import Agent, handoff  
from agents.extensions import handoff\_filters  
  
agent = Agent(name="FAQ agent")  
  
handoff\_obj = handoff(  
 agent=agent,  
 input\_filter=handoff\_filters.remove\_all\_tools, # (1)!  
)

1. This will automatically remove all tools from the history when FAQ agent is called.

## Recommended prompts

To make sure that LLMs understand handoffs properly, we recommend including information about handoffs in your agents. We have a suggested prefix in [agents.extensions.handoff\_prompt.RECOMMENDED\_PROMPT\_PREFIX][], or you can call [agents.extensions.handoff\_prompt.prompt\_with\_handoff\_instructions][] to automatically add recommended data to your prompts.

from agents import Agent  
from agents.extensions.handoff\_prompt import RECOMMENDED\_PROMPT\_PREFIX  
  
billing\_agent = Agent(  
 name="Billing agent",  
 instructions=f"""{RECOMMENDED\_PROMPT\_PREFIX}  
 <Fill in the rest of your prompt here>.""",  
)

# OpenAI Agents SDK

The [OpenAI Agents SDK](https://github.com/openai/openai-agents-python) 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](https://github.com/openai/swarm/tree/main). The Agents SDK has a very small set of primitives:

* **Agents**, which are LLMs equipped with instructions and tools
* **Handoffs**, which allow agents to delegate to other agents for specific tasks
* **Guardrails**, which enable the inputs to agents to be validated

In combination with Python, these primitives are powerful enough to express complex relationships between tools and agents, and allow you to build real-world applications without a steep learning curve. In addition, the SDK comes with built-in **tracing** that lets you visualize and debug your agentic flows, as well as evaluate them and even fine-tune models for your application.

## Why use the Agents SDK

The SDK has two driving design principles:

1. Enough features to be worth using, but few enough primitives to make it quick to learn.
2. Works great out of the box, but you can customize exactly what happens.

Here are the main features of the SDK:

* Agent loop: Built-in agent loop that handles calling tools, sending results to the LLM, and looping until the LLM is done.
* Python-first: Use built-in language features to orchestrate and chain agents, rather than needing to learn new abstractions.
* Handoffs: A powerful feature to coordinate and delegate between multiple agents.
* Guardrails: Run input validations and checks in parallel to your agents, breaking early if the checks fail.
* Function tools: Turn any Python function into a tool, with automatic schema generation and Pydantic-powered validation.
* Tracing: Built-in tracing that lets you visualize, debug and monitor your workflows, as well as use the OpenAI suite of evaluation, fine-tuning and distillation tools.

## Installation

pip install openai-agents

## Hello world example

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-...

# Model context protocol (MCP)

The [Model context protocol](https://modelcontextprotocol.io/introduction) (aka MCP) is a way to provide tools and context to the LLM. From the MCP docs:

MCP is an open protocol that standardizes how applications provide context to LLMs. Think of MCP like a USB-C port for AI applications. Just as USB-C provides a standardized way to connect your devices to various peripherals and accessories, MCP provides a standardized way to connect AI models to different data sources and tools.

The Agents SDK has support for MCP. This enables you to use a wide range of MCP servers to provide tools to your Agents.

## MCP servers

Currently, the MCP spec defines two kinds of servers, based on the transport mechanism they use:

1. **stdio** servers run as a subprocess of your application. You can think of them as running “locally”.
2. **HTTP over SSE** servers run remotely. You connect to them via a URL.

You can use the [MCPServerStdio][agents.mcp.server.MCPServerStdio] and [MCPServerSse][agents.mcp.server.MCPServerSse] classes to connect to these servers.

For example, this is how you’d use the [official MCP filesystem server](https://www.npmjs.com/package/@modelcontextprotocol/server-filesystem).

async with MCPServerStdio(  
 params={  
 "command": "npx",  
 "args": ["-y", "@modelcontextprotocol/server-filesystem", samples\_dir],  
 }  
) as server:  
 tools = await server.list\_tools()

## Using MCP servers

MCP servers can be added to Agents. The Agents SDK will call list\_tools() on the MCP servers each time the Agent is run. This makes the LLM aware of the MCP server’s tools. When the LLM calls a tool from an MCP server, the SDK calls call\_tool() on that server.

agent=Agent(  
 name="Assistant",  
 instructions="Use the tools to achieve the task",  
 mcp\_servers=[mcp\_server\_1, mcp\_server\_2]  
)

## Caching

Every time an Agent runs, it calls list\_tools() on the MCP server. This can be a latency hit, especially if the server is a remote server. To automatically cache the list of tools, you can pass cache\_tools\_list=True to both [MCPServerStdio][agents.mcp.server.MCPServerStdio] and [MCPServerSse][agents.mcp.server.MCPServerSse]. You should only do this if you’re certain the tool list will not change.

If you want to invalidate the cache, you can call invalidate\_tools\_cache() on the servers.

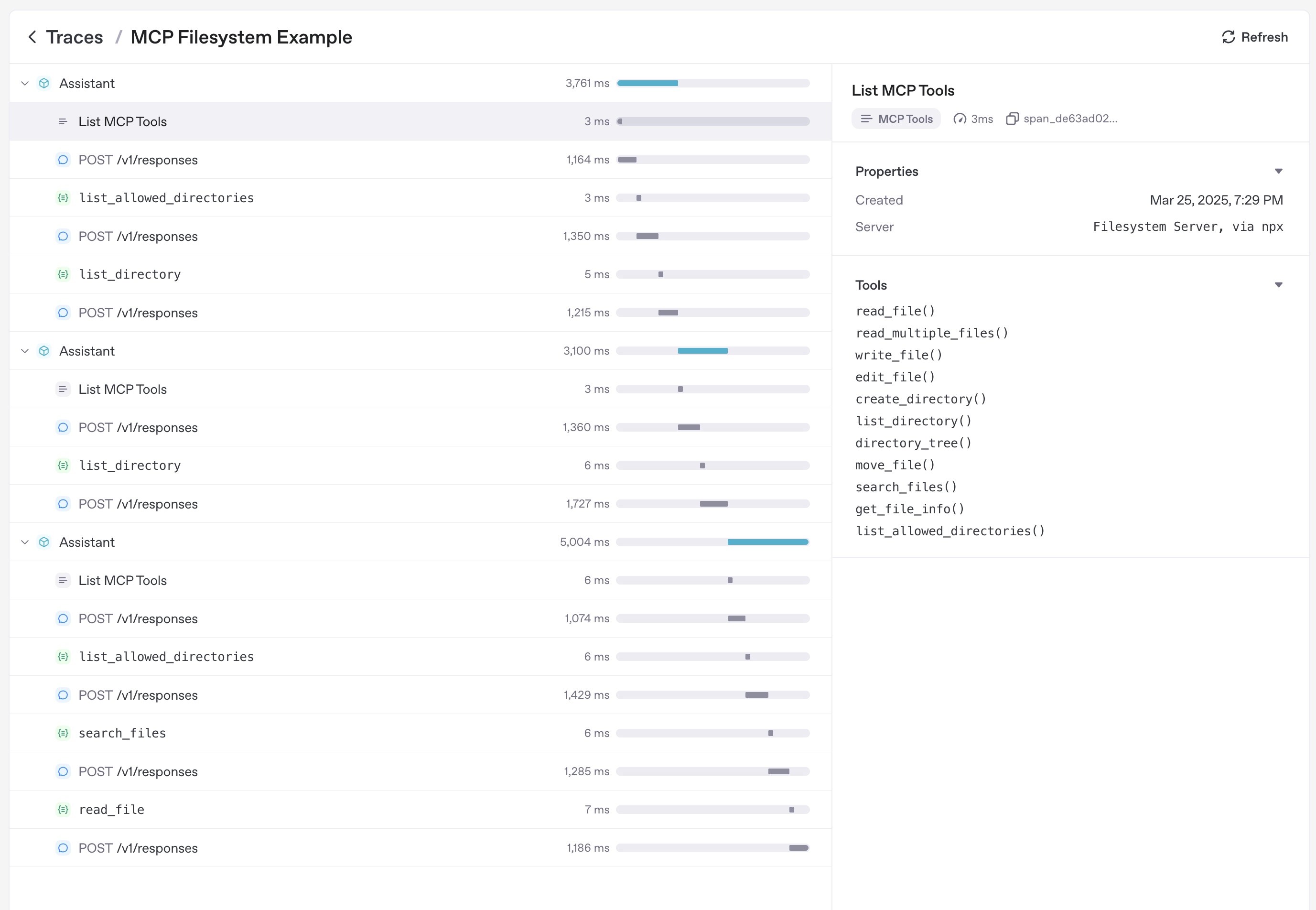
## End-to-end examples

View complete working examples at [examples/mcp](https://github.com/openai/openai-agents-python/tree/main/examples/mcp).

## Tracing

[Tracing](./tracing.md) automatically captures MCP operations, including:

1. Calls to the MCP server to list tools
2. MCP-related info on function calls

 # Models

The Agents SDK comes with out-of-the-box support for OpenAI models in two flavors:

* **Recommended**: the [OpenAIResponsesModel][agents.models.openai\_responses.OpenAIResponsesModel], which calls OpenAI APIs using the new [Responses API](https://platform.openai.com/docs/api-reference/responses).
* The [OpenAIChatCompletionsModel][agents.models.openai\_chatcompletions.OpenAIChatCompletionsModel], which calls OpenAI APIs using the [Chat Completions API](https://platform.openai.com/docs/api-reference/chat).

## Non-OpenAI models

You can use most other non-OpenAI models via the [LiteLLM integration](./litellm.md). First, install the litellm dependency group:

pip install "openai-agents[litellm]"

Then, use any of the [supported models](https://docs.litellm.ai/docs/providers) with the litellm/ prefix:

claude\_agent = Agent(model="litellm/anthropic/claude-3-5-sonnet-20240620", ...)  
gemini\_agent = Agent(model="litellm/gemini/gemini-2.5-flash-preview-04-17", ...)

### Other ways to use non-OpenAI models

You can integrate other LLM providers in 3 more ways (examples [here](https://github.com/openai/openai-agents-python/tree/main/examples/model_providers/)):

1. [set\_default\_openai\_client][agents.set\_default\_openai\_client] is useful in cases where you want to globally use an instance of AsyncOpenAI as the LLM client. This is for cases where the LLM provider has an OpenAI compatible API endpoint, and you can set the base\_url and api\_key. See a configurable example in [examples/model\_providers/custom\_example\_global.py](https://github.com/openai/openai-agents-python/tree/main/examples/model_providers/custom_example_global.py).
2. [ModelProvider][agents.models.interface.ModelProvider] is at the Runner.run level. This lets you say “use a custom model provider for all agents in this run”. See a configurable example in [examples/model\_providers/custom\_example\_provider.py](https://github.com/openai/openai-agents-python/tree/main/examples/model_providers/custom_example_provider.py).
3. [Agent.model][agents.agent.Agent.model] lets you specify the model on a specific Agent instance. This enables you to mix and match different providers for different agents. See a configurable example in [examples/model\_providers/custom\_example\_agent.py](https://github.com/openai/openai-agents-python/tree/main/examples/model_providers/custom_example_agent.py). An easy way to use most available models is via the [LiteLLM integration](./litellm.md).

In cases where you do not have an API key from platform.openai.com, we recommend disabling tracing via set\_tracing\_disabled(), or setting up a [different tracing processor](../tracing.md).

!!! note

In these examples, we use the Chat Completions API/model, because most LLM providers don't yet support the Responses API. If your LLM provider does support it, we recommend using Responses.

## Mixing and matching models

Within a single workflow, you may want to use different models for each agent. For example, you could use a smaller, faster model for triage, while using a larger, more capable model for complex tasks. When configuring an [Agent][agents.Agent], you can select a specific model by either:

1. Passing the name of a model.
2. Passing any model name + a [ModelProvider][agents.models.interface.ModelProvider] that can map that name to a Model instance.
3. Directly providing a [Model][agents.models.interface.Model] implementation.

!!!note

While our SDK supports both the [`OpenAIResponsesModel`][agents.models.openai\_responses.OpenAIResponsesModel] and the [`OpenAIChatCompletionsModel`][agents.models.openai\_chatcompletions.OpenAIChatCompletionsModel] shapes, we recommend using a single model shape for each workflow because the two shapes support a different set of features and tools. If your workflow requires mixing and matching model shapes, make sure that all the features you're using are available on both.

from agents import Agent, Runner, AsyncOpenAI, OpenAIChatCompletionsModel  
import asyncio  
  
spanish\_agent = Agent(  
 name="Spanish agent",  
 instructions="You only speak Spanish.",  
 model="o3-mini", # (1)!  
)  
  
english\_agent = Agent(  
 name="English agent",  
 instructions="You only speak English",  
 model=OpenAIChatCompletionsModel( # (2)!  
 model="gpt-4o",  
 openai\_client=AsyncOpenAI()  
 ),  
)  
  
triage\_agent = Agent(  
 name="Triage agent",  
 instructions="Handoff to the appropriate agent based on the language of the request.",  
 handoffs=[spanish\_agent, english\_agent],  
 model="gpt-3.5-turbo",  
)  
  
async def main():  
 result = await Runner.run(triage\_agent, input="Hola, ¿cómo estás?")  
 print(result.final\_output)

1. Sets the name of an OpenAI model directly.
2. Provides a [Model][agents.models.interface.Model] implementation.

When you want to further configure the model used for an agent, you can pass [ModelSettings][agents.models.interface.ModelSettings], which provides optional model configuration parameters such as temperature.

from agents import Agent, ModelSettings  
  
english\_agent = Agent(  
 name="English agent",  
 instructions="You only speak English",  
 model="gpt-4o",  
 model\_settings=ModelSettings(temperature=0.1),  
)

## Common issues with using other LLM providers

### Tracing client error 401

If you get errors related to tracing, this is because traces are uploaded to OpenAI servers, and you don’t have an OpenAI API key. You have three options to resolve this:

1. Disable tracing entirely: [set\_tracing\_disabled(True)][agents.set\_tracing\_disabled].
2. Set an OpenAI key for tracing: [set\_tracing\_export\_api\_key(...)][agents.set\_tracing\_export\_api\_key]. This API key will only be used for uploading traces, and must be from [platform.openai.com](https://platform.openai.com/).
3. Use a non-OpenAI trace processor. See the [tracing docs](../tracing.md#custom-tracing-processors).

### Responses API support

The SDK uses the Responses API by default, but most other LLM providers don’t yet support it. You may see 404s or similar issues as a result. To resolve, you have two options:

1. Call [set\_default\_openai\_api("chat\_completions")][agents.set\_default\_openai\_api]. This works if you are setting OPENAI\_API\_KEY and OPENAI\_BASE\_URL via environment vars.
2. Use [OpenAIChatCompletionsModel][agents.models.openai\_chatcompletions.OpenAIChatCompletionsModel]. There are examples [here](https://github.com/openai/openai-agents-python/tree/main/examples/model_providers/).

### Structured outputs support

Some model providers don’t have support for [structured outputs](https://platform.openai.com/docs/guides/structured-outputs). This sometimes results in an error that looks something like this:

BadRequestError: Error code: 400 - {'error': {'message': "'response\_format.type' : value is not one of the allowed values ['text','json\_object']", 'type': 'invalid\_request\_error'}}

This is a shortcoming of some model providers - they support JSON outputs, but don’t allow you to specify the json\_schema to use for the output. We are working on a fix for this, but we suggest relying on providers that do have support for JSON schema output, because otherwise your app will often break because of malformed JSON.

## Mixing models across providers

You need to be aware of feature differences between model providers, or you may run into errors. For example, OpenAI supports structured outputs, multimodal input, and hosted file search and web search, but many other providers don’t support these features. Be aware of these limitations:

* Don’t send unsupported tools to providers that don’t understand them
* Filter out multimodal inputs before calling models that are text-only
* Be aware that providers that don’t support structured JSON outputs will occasionally produce invalid JSON. # Using any model via LiteLLM

!!! note

The LiteLLM integration is in beta. You may run into issues with some model providers, especially smaller ones. Please report any issues via [Github issues](https://github.com/openai/openai-agents-python/issues) and we'll fix quickly.

[LiteLLM](https://docs.litellm.ai/docs/) is a library that allows you to use 100+ models via a single interface. We’ve added a LiteLLM integration to allow you to use any AI model in the Agents SDK.

## Setup

You’ll need to ensure litellm is available. You can do this by installing the optional litellm dependency group:

pip install "openai-agents[litellm]"

Once done, you can use [LitellmModel][agents.extensions.models.litellm\_model.LitellmModel] in any agent.

## Example

This is a fully working example. When you run it, you’ll be prompted for a model name and API key. For example, you could enter:

* openai/gpt-4.1 for the model, and your OpenAI API key
* anthropic/claude-3-5-sonnet-20240620 for the model, and your Anthropic API key
* etc

For a full list of models supported in LiteLLM, see the [litellm providers docs](https://docs.litellm.ai/docs/providers).

from \_\_future\_\_ import annotations  
  
import asyncio  
  
from agents import Agent, Runner, function\_tool, set\_tracing\_disabled  
from agents.extensions.models.litellm\_model import LitellmModel  
  
@function\_tool  
def get\_weather(city: str):  
 print(f"[debug] getting weather for {city}")  
 return f"The weather in {city} is sunny."  
  
  
async def main(model: str, api\_key: str):  
 agent = Agent(  
 name="Assistant",  
 instructions="You only respond in haikus.",  
 model=LitellmModel(model=model, api\_key=api\_key),  
 tools=[get\_weather],  
 )  
  
 result = await Runner.run(agent, "What's the weather in Tokyo?")  
 print(result.final\_output)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 # First try to get model/api key from args  
 import argparse  
  
 parser = argparse.ArgumentParser()  
 parser.add\_argument("--model", type=str, required=False)  
 parser.add\_argument("--api-key", type=str, required=False)  
 args = parser.parse\_args()  
  
 model = args.model  
 if not model:  
 model = input("Enter a model name for Litellm: ")  
  
 api\_key = args.api\_key  
 if not api\_key:  
 api\_key = input("Enter an API key for Litellm: ")  
  
 asyncio.run(main(model, api\_key))

# 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.

You can mix and match these patterns. Each has their own tradeoffs, described below.

## Orchestrating via LLM

An agent is an LLM equipped with instructions, tools and handoffs. This means that given an open-ended task, the LLM can autonomously plan how it will tackle the task, using tools to take actions and acquire data, and using handoffs to delegate tasks to sub-agents. For example, a research agent could be equipped with tools like:

* Web search to find information online
* File search and retrieval to search through proprietary data and connections
* Computer use to take actions on a computer
* Code execution to do data analysis
* Handoffs to specialized agents that are great at planning, report writing and more.

This pattern is great when the task is open-ended and you want to rely on the intelligence of an LLM. The most important tactics here are:

1. Invest in good prompts. Make it clear what tools are available, how to use them, and what parameters it must operate within.
2. Monitor your app and iterate on it. See where things go wrong, and iterate on your prompts.
3. Allow the agent to introspect and improve. For example, run it in a loop, and let it critique itself; or, provide error messages and let it improve.
4. Have specialized agents that excel in one task, rather than having a general purpose agent that is expected to be good at anything.
5. Invest in [evals](https://platform.openai.com/docs/guides/evals). This lets you train your agents to improve and get better at tasks.

## Orchestrating via code

While orchestrating via LLM is powerful, orchestrating via code makes tasks more deterministic and predictable, in terms of speed, cost and performance. Common patterns here are:

* Using [structured outputs](https://platform.openai.com/docs/guides/structured-outputs) to generate well formed data that you can inspect with your code. For example, you might ask an agent to classify the task into a few categories, and then pick the next agent based on the category.
* Chaining multiple agents by transforming the output of one into the input of the next. You can decompose a task like writing a blog post into a series of steps - do research, write an outline, write the blog post, critique it, and then improve it.
* Running the agent that performs the task in a while loop with an agent that evaluates and provides feedback, until the evaluator says the output passes certain criteria.
* Running multiple agents in parallel, e.g. via Python primitives like asyncio.gather. This is useful for speed when you have multiple tasks that don’t depend on each other.

We have a number of examples in [examples/agent\_patterns](https://github.com/openai/openai-agents-python/tree/main/examples/agent_patterns). # Quickstart

## Create a project and virtual environment

You’ll only need to do this once.

mkdir my\_project  
cd my\_project  
python -m venv .venv

### Activate the virtual environment

Do this every time you start a new terminal session.

source .venv/bin/activate

### Install the Agents SDK

pip install openai-agents # or `uv add openai-agents`, etc

### Set an OpenAI API key

If you don’t have one, follow [these instructions](https://platform.openai.com/docs/quickstart#create-and-export-an-api-key) to create an OpenAI API key.

export OPENAI\_API\_KEY=sk-...

## Create your first agent

Agents are defined with instructions, a name, and optional config (such as model\_config)

from agents import Agent  
  
agent = Agent(  
 name="Math Tutor",  
 instructions="You provide help with math problems. Explain your reasoning at each step and include examples",  
)

## Add a few more agents

Additional agents can be defined in the same way. handoff\_descriptions provide additional context for determining handoff routing

from agents import Agent  
  
history\_tutor\_agent = Agent(  
 name="History Tutor",  
 handoff\_description="Specialist agent for historical questions",  
 instructions="You provide assistance with historical queries. Explain important events and context clearly.",  
)  
  
math\_tutor\_agent = Agent(  
 name="Math Tutor",  
 handoff\_description="Specialist agent for math questions",  
 instructions="You provide help with math problems. Explain your reasoning at each step and include examples",  
)

## Define your handoffs

On each agent, you can define an inventory of outgoing handoff options that the agent can choose from to decide how to make progress on their task.

triage\_agent = Agent(  
 name="Triage Agent",  
 instructions="You determine which agent to use based on the user's homework question",  
 handoffs=[history\_tutor\_agent, math\_tutor\_agent]  
)

## Run the agent orchestration

Let’s check that the workflow runs and the triage agent correctly routes between the two specialist agents.

from agents import Runner  
  
async def main():  
 result = await Runner.run(triage\_agent, "What is the capital of France?")  
 print(result.final\_output)

## Add a guardrail

You can define custom guardrails to run on the input or output.

from agents import GuardrailFunctionOutput, Agent, Runner  
from pydantic import BaseModel  
  
class HomeworkOutput(BaseModel):  
 is\_homework: bool  
 reasoning: str  
  
guardrail\_agent = Agent(  
 name="Guardrail check",  
 instructions="Check if the user is asking about homework.",  
 output\_type=HomeworkOutput,  
)  
  
async def homework\_guardrail(ctx, agent, input\_data):  
 result = await Runner.run(guardrail\_agent, input\_data, context=ctx.context)  
 final\_output = result.final\_output\_as(HomeworkOutput)  
 return GuardrailFunctionOutput(  
 output\_info=final\_output,  
 tripwire\_triggered=not final\_output.is\_homework,  
 )

## Put it all together

Let’s put it all together and run the entire workflow, using handoffs and the input guardrail.

from agents import Agent, InputGuardrail, GuardrailFunctionOutput, Runner  
from pydantic import BaseModel  
import asyncio  
  
class HomeworkOutput(BaseModel):  
 is\_homework: bool  
 reasoning: str  
  
guardrail\_agent = Agent(  
 name="Guardrail check",  
 instructions="Check if the user is asking about homework.",  
 output\_type=HomeworkOutput,  
)  
  
math\_tutor\_agent = Agent(  
 name="Math Tutor",  
 handoff\_description="Specialist agent for math questions",  
 instructions="You provide help with math problems. Explain your reasoning at each step and include examples",  
)  
  
history\_tutor\_agent = Agent(  
 name="History Tutor",  
 handoff\_description="Specialist agent for historical questions",  
 instructions="You provide assistance with historical queries. Explain important events and context clearly.",  
)  
  
  
async def homework\_guardrail(ctx, agent, input\_data):  
 result = await Runner.run(guardrail\_agent, input\_data, context=ctx.context)  
 final\_output = result.final\_output\_as(HomeworkOutput)  
 return GuardrailFunctionOutput(  
 output\_info=final\_output,  
 tripwire\_triggered=not final\_output.is\_homework,  
 )  
  
triage\_agent = Agent(  
 name="Triage Agent",  
 instructions="You determine which agent to use based on the user's homework question",  
 handoffs=[history\_tutor\_agent, math\_tutor\_agent],  
 input\_guardrails=[  
 InputGuardrail(guardrail\_function=homework\_guardrail),  
 ],  
)  
  
async def main():  
 result = await Runner.run(triage\_agent, "who was the first president of the united states?")  
 print(result.final\_output)  
  
 result = await Runner.run(triage\_agent, "what is life")  
 print(result.final\_output)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 asyncio.run(main())

## View your traces

To review what happened during your agent run, navigate to the [Trace viewer in the OpenAI Dashboard](https://platform.openai.com/traces) to view traces of your agent runs.

## Next steps

Learn how to build more complex agentic flows:

* Learn about how to configure [Agents](agents.md).
* Learn about [running agents](running_agents.md).
* Learn about [tools](tools.md), [guardrails](guardrails.md) and [models](models/index.md). # Results

When you call the Runner.run methods, you either get a:

* [RunResult][agents.result.RunResult] if you call run or run\_sync
* [RunResultStreaming][agents.result.RunResultStreaming] if you call run\_streamed

Both of these inherit from [RunResultBase][agents.result.RunResultBase], which is where most useful information is present.

## Final output

The [final\_output][agents.result.RunResultBase.final\_output] property contains the final output of the last agent that ran. This is either:

* a str, if the last agent didn’t have an output\_type defined
* an object of type last\_agent.output\_type, if the agent had an output type defined.

!!! note

`final\_output` is of type `Any`. We can't statically type this, because of handoffs. If handoffs occur, that means any Agent might be the last agent, so we don't statically know the set of possible output types.

## Inputs for the next turn

You can use [result.to\_input\_list()][agents.result.RunResultBase.to\_input\_list] to turn the result into an input list that concatenates the original input you provided, to the items generated during the agent run. This makes it convenient to take the outputs of one agent run and pass them into another run, or to run it in a loop and append new user inputs each time.

## Last agent

The [last\_agent][agents.result.RunResultBase.last\_agent] property contains the last agent that ran. Depending on your application, this is often useful for the next time the user inputs something. For example, if you have a frontline triage agent that hands off to a language-specific agent, you can store the last agent, and re-use it the next time the user messages the agent.

## New items

The [new\_items][agents.result.RunResultBase.new\_items] property contains the new items generated during the run. The items are [RunItem][agents.items.RunItem]s. A run item wraps the raw item generated by the LLM.

* [MessageOutputItem][agents.items.MessageOutputItem] indicates a message from the LLM. The raw item is the message generated.
* [HandoffCallItem][agents.items.HandoffCallItem] indicates that the LLM called the handoff tool. The raw item is the tool call item from the LLM.
* [HandoffOutputItem][agents.items.HandoffOutputItem] indicates that a handoff occurred. The raw item is the tool response to the handoff tool call. You can also access the source/target agents from the item.
* [ToolCallItem][agents.items.ToolCallItem] indicates that the LLM invoked a tool.
* [ToolCallOutputItem][agents.items.ToolCallOutputItem] indicates that a tool was called. The raw item is the tool response. You can also access the tool output from the item.
* [ReasoningItem][agents.items.ReasoningItem] indicates a reasoning item from the LLM. The raw item is the reasoning generated.

## Other information

### Guardrail results

The [input\_guardrail\_results][agents.result.RunResultBase.input\_guardrail\_results] and [output\_guardrail\_results][agents.result.RunResultBase.output\_guardrail\_results] properties contain the results of the guardrails, if any. Guardrail results can sometimes contain useful information you want to log or store, so we make these available to you.

### Raw responses

The [raw\_responses][agents.result.RunResultBase.raw\_responses] property contains the [ModelResponse][agents.items.ModelResponse]s generated by the LLM.

### Original input

The [input][agents.result.RunResultBase.input] property contains the original input you provided to the run method. In most cases you won’t need this, but it’s available in case you do. # Running agents

You can run agents via the [Runner][agents.run.Runner] class. You have 3 options:

1. [Runner.run()][agents.run.Runner.run], which runs async and returns a [RunResult][agents.result.RunResult].
2. [Runner.run\_sync()][agents.run.Runner.run\_sync], which is a sync method and just runs .run() under the hood.
3. [Runner.run\_streamed()][agents.run.Runner.run\_streamed], which runs async and returns a [RunResultStreaming][agents.result.RunResultStreaming]. It calls the LLM in streaming mode, and streams those events to you as they are received.

from agents import Agent, Runner  
  
async def main():  
 agent = Agent(name="Assistant", instructions="You are a helpful assistant")  
  
 result = await Runner.run(agent, "Write a haiku about recursion in programming.")  
 print(result.final\_output)  
 # Code within the code,  
 # Functions calling themselves,  
 # Infinite loop's dance.

Read more in the [results guide](results.md).

## The agent loop

When you use the run method in Runner, you pass in a starting agent and input. The input can either be a string (which is considered a user message), or a list of input items, which are the items in the OpenAI Responses API.

The runner then runs a loop:

1. We call the LLM for the current agent, with the current input.
2. The LLM produces its output.
   1. If the LLM returns a final\_output, the loop ends and we return the result.
   2. If the LLM does a handoff, we update the current agent and input, and re-run the loop.
   3. If the LLM produces tool calls, we run those tool calls, append the results, and re-run the loop.
3. If we exceed the max\_turns passed, we raise a [MaxTurnsExceeded][agents.exceptions.MaxTurnsExceeded] exception.

!!! note

The rule for whether the LLM output is considered as a "final output" is that it produces text output with the desired type, and there are no tool calls.

## Streaming

Streaming allows you to additionally receive streaming events as the LLM runs. Once the stream is done, the [RunResultStreaming][agents.result.RunResultStreaming] will contain the complete information about the run, including all the new outputs produces. You can call .stream\_events() for the streaming events. Read more in the [streaming guide](streaming.md).

## Run config

The run\_config parameter lets you configure some global settings for the agent run:

* [model][agents.run.RunConfig.model]: Allows setting a global LLM model to use, irrespective of what model each Agent has.
* [model\_provider][agents.run.RunConfig.model\_provider]: A model provider for looking up model names, which defaults to OpenAI.
* [model\_settings][agents.run.RunConfig.model\_settings]: Overrides agent-specific settings. For example, you can set a global temperature or top\_p.
* [input\_guardrails][agents.run.RunConfig.input\_guardrails], [output\_guardrails][agents.run.RunConfig.output\_guardrails]: A list of input or output guardrails to include on all runs.
* [handoff\_input\_filter][agents.run.RunConfig.handoff\_input\_filter]: A global input filter to apply to all handoffs, if the handoff doesn’t already have one. The input filter allows you to edit the inputs that are sent to the new agent. See the documentation in [Handoff.input\_filter][agents.handoffs.Handoff.input\_filter] for more details.
* [tracing\_disabled][agents.run.RunConfig.tracing\_disabled]: Allows you to disable [tracing](tracing.md) for the entire run.
* [trace\_include\_sensitive\_data][agents.run.RunConfig.trace\_include\_sensitive\_data]: Configures whether traces will include potentially sensitive data, such as LLM and tool call inputs/outputs.
* [workflow\_name][agents.run.RunConfig.workflow\_name], [trace\_id][agents.run.RunConfig.trace\_id], [group\_id][agents.run.RunConfig.group\_id]: Sets the tracing workflow name, trace ID and trace group ID for the run. We recommend at least setting workflow\_name. The group ID is an optional field that lets you link traces across multiple runs.
* [trace\_metadata][agents.run.RunConfig.trace\_metadata]: Metadata to include on all traces.

## Conversations/chat threads

Calling any of the run methods can result in one or more agents running (and hence one or more LLM calls), but it represents a single logical turn in a chat conversation. For example:

1. User turn: user enter text
2. Runner run: first agent calls LLM, runs tools, does a handoff to a second agent, second agent runs more tools, and then produces an output.

At the end of the agent run, you can choose what to show to the user. For example, you might show the user every new item generated by the agents, or just the final output. Either way, the user might then ask a followup question, in which case you can call the run method again.

You can use the base [RunResultBase.to\_input\_list()][agents.result.RunResultBase.to\_input\_list] method to get the inputs for the next turn.

async def main():  
 agent = Agent(name="Assistant", instructions="Reply very concisely.")  
  
 with trace(workflow\_name="Conversation", group\_id=thread\_id):  
 # First turn  
 result = await Runner.run(agent, "What city is the Golden Gate Bridge in?")  
 print(result.final\_output)  
 # San Francisco  
  
 # Second turn  
 new\_input = result.to\_input\_list() + [{"role": "user", "content": "What state is it in?"}]  
 result = await Runner.run(agent, new\_input)  
 print(result.final\_output)  
 # California

## Exceptions

The SDK raises exceptions in certain cases. The full list is in [agents.exceptions][]. As an overview:

* [AgentsException][agents.exceptions.AgentsException] is the base class for all exceptions raised in the SDK.
* [MaxTurnsExceeded][agents.exceptions.MaxTurnsExceeded] is raised when the run exceeds the max\_turns passed to the run methods.
* [ModelBehaviorError][agents.exceptions.ModelBehaviorError] is raised when the model produces invalid outputs, e.g. malformed JSON or using non-existent tools.
* [UserError][agents.exceptions.UserError] is raised when you (the person writing code using the SDK) make an error using the SDK.
* [InputGuardrailTripwireTriggered][agents.exceptions.InputGuardrailTripwireTriggered], [OutputGuardrailTripwireTriggered][agents.exceptions.OutputGuardrailTripwireTriggered] is raised when a [guardrail](guardrails.md) is tripped. # Streaming

Streaming lets you subscribe to updates of the agent run as it proceeds. This can be useful for showing the end-user progress updates and partial responses.

To stream, you can call [Runner.run\_streamed()][agents.run.Runner.run\_streamed], which will give you a [RunResultStreaming][agents.result.RunResultStreaming]. Calling result.stream\_events() gives you an async stream of [StreamEvent][agents.stream\_events.StreamEvent] objects, which are described below.

## Raw response events

[RawResponsesStreamEvent][agents.stream\_events.RawResponsesStreamEvent] are raw events passed directly from the LLM. They are in OpenAI Responses API format, which means each event has a type (like response.created, response.output\_text.delta, etc) and data. These events are useful if you want to stream response messages to the user as soon as they are generated.

For example, this will output the text generated by the LLM token-by-token.

import asyncio  
from openai.types.responses import ResponseTextDeltaEvent  
from agents import Agent, Runner  
  
async def main():  
 agent = Agent(  
 name="Joker",  
 instructions="You are a helpful assistant.",  
 )  
  
 result = Runner.run\_streamed(agent, input="Please tell me 5 jokes.")  
 async for event in result.stream\_events():  
 if event.type == "raw\_response\_event" and isinstance(event.data, ResponseTextDeltaEvent):  
 print(event.data.delta, end="", flush=True)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 asyncio.run(main())

## Run item events and agent events

[RunItemStreamEvent][agents.stream\_events.RunItemStreamEvent]s are higher level events. They inform you when an item has been fully generated. This allows you to push progress updates at the level of “message generated”, “tool ran”, etc, instead of each token. Similarly, [AgentUpdatedStreamEvent][agents.stream\_events.AgentUpdatedStreamEvent] gives you updates when the current agent changes (e.g. as the result of a handoff).

For example, this will ignore raw events and stream updates to the user.

import asyncio  
import random  
from agents import Agent, ItemHelpers, Runner, function\_tool  
  
@function\_tool  
def how\_many\_jokes() -> int:  
 return random.randint(1, 10)  
  
  
async def main():  
 agent = Agent(  
 name="Joker",  
 instructions="First call the `how\_many\_jokes` tool, then tell that many jokes.",  
 tools=[how\_many\_jokes],  
 )  
  
 result = Runner.run\_streamed(  
 agent,  
 input="Hello",  
 )  
 print("=== Run starting ===")  
  
 async for event in result.stream\_events():  
 # We'll ignore the raw responses event deltas  
 if event.type == "raw\_response\_event":  
 continue  
 # When the agent updates, print that  
 elif event.type == "agent\_updated\_stream\_event":  
 print(f"Agent updated: {event.new\_agent.name}")  
 continue  
 # When items are generated, print them  
 elif event.type == "run\_item\_stream\_event":  
 if event.item.type == "tool\_call\_item":  
 print("-- Tool was called")  
 elif event.item.type == "tool\_call\_output\_item":  
 print(f"-- Tool output: {event.item.output}")  
 elif event.item.type == "message\_output\_item":  
 print(f"-- Message output:\n {ItemHelpers.text\_message\_output(event.item)}")  
 else:  
 pass # Ignore other event types  
  
 print("=== Run complete ===")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 asyncio.run(main())

# Tools

Tools let agents take actions: things like fetching data, running code, calling external APIs, and even using a computer. There are three classes of tools in the Agent SDK:

* Hosted tools: these run on LLM servers alongside the AI models. OpenAI offers retrieval, web search and computer use as hosted tools.
* Function calling: these allow you to use any Python function as a tool.
* Agents as tools: this allows you to use an agent as a tool, allowing Agents to call other agents without handing off to them.

## Hosted tools

OpenAI offers a few built-in tools when using the [OpenAIResponsesModel][agents.models.openai\_responses.OpenAIResponsesModel]:

* The [WebSearchTool][agents.tool.WebSearchTool] lets an agent search the web.
* The [FileSearchTool][agents.tool.FileSearchTool] allows retrieving information from your OpenAI Vector Stores.
* The [ComputerTool][agents.tool.ComputerTool] allows automating computer use tasks.

from agents import Agent, FileSearchTool, Runner, WebSearchTool  
  
agent = Agent(  
 name="Assistant",  
 tools=[  
 WebSearchTool(),  
 FileSearchTool(  
 max\_num\_results=3,  
 vector\_store\_ids=["VECTOR\_STORE\_ID"],  
 ),  
 ],  
)  
  
async def main():  
 result = await Runner.run(agent, "Which coffee shop should I go to, taking into account my preferences and the weather today in SF?")  
 print(result.final\_output)

## Function tools

You can use any Python function as a tool. The Agents SDK will setup the tool automatically:

* The name of the tool will be the name of the Python function (or you can provide a name)
* Tool description will be taken from the docstring of the function (or you can provide a description)
* The schema for the function inputs is automatically created from the function’s arguments
* Descriptions for each input are taken from the docstring of the function, unless disabled

We use Python’s inspect module to extract the function signature, along with [griffe](https://mkdocstrings.github.io/griffe/) to parse docstrings and pydantic for schema creation.

import json  
  
from typing\_extensions import TypedDict, Any  
  
from agents import Agent, FunctionTool, RunContextWrapper, function\_tool  
  
  
class Location(TypedDict):  
 lat: float  
 long: float  
  
@function\_tool # (1)!  
async def fetch\_weather(location: Location) -> str:  
 # (2)!  
 """Fetch the weather for a given location.  
  
 Args:  
 location: The location to fetch the weather for.  
 """  
 # In real life, we'd fetch the weather from a weather API  
 return "sunny"  
  
  
@function\_tool(name\_override="fetch\_data") # (3)!  
def read\_file(ctx: RunContextWrapper[Any], path: str, directory: str | None = None) -> str:  
 """Read the contents of a file.  
  
 Args:  
 path: The path to the file to read.  
 directory: The directory to read the file from.  
 """  
 # In real life, we'd read the file from the file system  
 return "<file contents>"  
  
  
agent = Agent(  
 name="Assistant",  
 tools=[fetch\_weather, read\_file], # (4)!  
)  
  
for tool in agent.tools:  
 if isinstance(tool, FunctionTool):  
 print(tool.name)  
 print(tool.description)  
 print(json.dumps(tool.params\_json\_schema, indent=2))  
 print()

1. You can use any Python types as arguments to your functions, and the function can be sync or async.
2. Docstrings, if present, are used to capture descriptions and argument descriptions
3. Functions can optionally take the context (must be the first argument). You can also set overrides, like the name of the tool, description, which docstring style to use, etc.
4. You can pass the decorated functions to the list of tools.

??? note “Expand to see output”

```  
fetch\_weather  
Fetch the weather for a given location.  
{  
"$defs": {  
 "Location": {  
 "properties": {  
 "lat": {  
 "title": "Lat",  
 "type": "number"  
 },  
 "long": {  
 "title": "Long",  
 "type": "number"  
 }  
 },  
 "required": [  
 "lat",  
 "long"  
 ],  
 "title": "Location",  
 "type": "object"  
 }  
},  
"properties": {  
 "location": {  
 "$ref": "#/$defs/Location",  
 "description": "The location to fetch the weather for."  
 }  
},  
"required": [  
 "location"  
],  
"title": "fetch\_weather\_args",  
"type": "object"  
}  
  
fetch\_data  
Read the contents of a file.  
{  
"properties": {  
 "path": {  
 "description": "The path to the file to read.",  
 "title": "Path",  
 "type": "string"  
 },  
 "directory": {  
 "anyOf": [  
 {  
 "type": "string"  
 },  
 {  
 "type": "null"  
 }  
 ],  
 "default": null,  
 "description": "The directory to read the file from.",  
 "title": "Directory"  
 }  
},  
"required": [  
 "path"  
],  
"title": "fetch\_data\_args",  
"type": "object"  
}  
```

### Custom function tools

Sometimes, you don’t want to use a Python function as a tool. You can directly create a [FunctionTool][agents.tool.FunctionTool] if you prefer. You’ll need to provide:

* name
* description
* params\_json\_schema, which is the JSON schema for the arguments
* on\_invoke\_tool, which is an async function that receives the context and the arguments as a JSON string, and must return the tool output as a string.

from typing import Any  
  
from pydantic import BaseModel  
  
from agents import RunContextWrapper, FunctionTool  
  
  
  
def do\_some\_work(data: str) -> str:  
 return "done"  
  
  
class FunctionArgs(BaseModel):  
 username: str  
 age: int  
  
  
async def run\_function(ctx: RunContextWrapper[Any], args: str) -> str:  
 parsed = FunctionArgs.model\_validate\_json(args)  
 return do\_some\_work(data=f"{parsed.username} is {parsed.age} years old")  
  
  
tool = FunctionTool(  
 name="process\_user",  
 description="Processes extracted user data",  
 params\_json\_schema=FunctionArgs.model\_json\_schema(),  
 on\_invoke\_tool=run\_function,  
)

### Automatic argument and docstring parsing

As mentioned before, we automatically parse the function signature to extract the schema for the tool, and we parse the docstring to extract descriptions for the tool and for individual arguments. Some notes on that:

1. The signature parsing is done via the inspect module. We use type annotations to understand the types for the arguments, and dynamically build a Pydantic model to represent the overall schema. It supports most types, including Python primitives, Pydantic models, TypedDicts, and more.
2. We use griffe to parse docstrings. Supported docstring formats are google, sphinx and numpy. We attempt to automatically detect the docstring format, but this is best-effort and you can explicitly set it when calling function\_tool. You can also disable docstring parsing by setting use\_docstring\_info to False.

The code for the schema extraction lives in [agents.function\_schema][].

## Agents as tools

In some workflows, you may want a central agent to orchestrate a network of specialized agents, instead of handing off control. You can do this by modeling agents as tools.

from agents import Agent, Runner  
import asyncio  
  
spanish\_agent = Agent(  
 name="Spanish agent",  
 instructions="You translate the user's message to Spanish",  
)  
  
french\_agent = Agent(  
 name="French agent",  
 instructions="You translate the user's message to French",  
)  
  
orchestrator\_agent = Agent(  
 name="orchestrator\_agent",  
 instructions=(  
 "You are a translation agent. You use the tools given to you to translate."  
 "If asked for multiple translations, you call the relevant tools."  
 ),  
 tools=[  
 spanish\_agent.as\_tool(  
 tool\_name="translate\_to\_spanish",  
 tool\_description="Translate the user's message to Spanish",  
 ),  
 french\_agent.as\_tool(  
 tool\_name="translate\_to\_french",  
 tool\_description="Translate the user's message to French",  
 ),  
 ],  
)  
  
async def main():  
 result = await Runner.run(orchestrator\_agent, input="Say 'Hello, how are you?' in Spanish.")  
 print(result.final\_output)

### Customizing tool-agents

The agent.as\_tool function is a convenience method to make it easy to turn an agent into a tool. It doesn’t support all configuration though; for example, you can’t set max\_turns. For advanced use cases, use Runner.run directly in your tool implementation:

@function\_tool  
async def run\_my\_agent() -> str:  
 """A tool that runs the agent with custom configs".  
  
 agent = Agent(name="My agent", instructions="...")  
  
 result = await Runner.run(  
 agent,  
 input="...",  
 max\_turns=5,  
 run\_config=...  
 )  
  
 return str(result.final\_output)

## Handling errors in function tools

When you create a function tool via @function\_tool, you can pass a failure\_error\_function. This is a function that provides an error response to the LLM in case the tool call crashes.

* By default (i.e. if you don’t pass anything), it runs a default\_tool\_error\_function which tells the LLM an error occurred.
* If you pass your own error function, it runs that instead, and sends the response to the LLM.
* If you explicitly pass None, then any tool call errors will be re-raised for you to handle. This could be a ModelBehaviorError if the model produced invalid JSON, or a UserError if your code crashed, etc.

If you are manually creating a FunctionTool object, then you must handle errors inside the on\_invoke\_tool function. # Tracing

The Agents SDK includes built-in tracing, collecting a comprehensive record of events during an agent run: LLM generations, tool calls, handoffs, guardrails, and even custom events that occur. Using the [Traces dashboard](https://platform.openai.com/traces), you can debug, visualize, and monitor your workflows during development and in production.

!!!note

Tracing is enabled by default. There are two ways to disable tracing:  
  
1. You can globally disable tracing by setting the env var `OPENAI\_AGENTS\_DISABLE\_TRACING=1`  
2. You can disable tracing for a single run by setting [`agents.run.RunConfig.tracing\_disabled`][] to `True`

***For organizations operating under a Zero Data Retention (ZDR) policy using OpenAI’s APIs, tracing is unavailable.***

## Traces and spans

* **Traces** represent a single end-to-end operation of a “workflow”. They’re composed of Spans. Traces have the following properties:
  + workflow\_name: This is the logical workflow or app. For example “Code generation” or “Customer service”.
  + trace\_id: A unique ID for the trace. Automatically generated if you don’t pass one. Must have the format trace\_<32\_alphanumeric>.
  + group\_id: Optional group ID, to link multiple traces from the same conversation. For example, you might use a chat thread ID.
  + disabled: If True, the trace will not be recorded.
  + metadata: Optional metadata for the trace.
* **Spans** represent operations that have a start and end time. Spans have:
  + started\_at and ended\_at timestamps.
  + trace\_id, to represent the trace they belong to
  + parent\_id, which points to the parent Span of this Span (if any)
  + span\_data, which is information about the Span. For example, AgentSpanData contains information about the Agent, GenerationSpanData contains information about the LLM generation, etc.

## Default tracing

By default, the SDK traces the following:

* The entire Runner.{run, run\_sync, run\_streamed}() is wrapped in a trace().
* Each time an agent runs, it is wrapped in agent\_span()
* LLM generations are wrapped in generation\_span()
* Function tool calls are each wrapped in function\_span()
* Guardrails are wrapped in guardrail\_span()
* Handoffs are wrapped in handoff\_span()
* Audio inputs (speech-to-text) are wrapped in a transcription\_span()
* Audio outputs (text-to-speech) are wrapped in a speech\_span()
* Related audio spans may be parented under a speech\_group\_span()

By default, the trace is named “Agent trace”. You can set this name if you use trace, or you can can configure the name and other properties with the [RunConfig][agents.run.RunConfig].

In addition, you can set up [custom trace processors](#custom-tracing-processors) to push traces to other destinations (as a replacement, or secondary destination).

## Higher level traces

Sometimes, you might want multiple calls to run() to be part of a single trace. You can do this by wrapping the entire code in a trace().

from agents import Agent, Runner, trace  
  
async def main():  
 agent = Agent(name="Joke generator", instructions="Tell funny jokes.")  
  
 with trace("Joke workflow"): # (1)!  
 first\_result = await Runner.run(agent, "Tell me a joke")  
 second\_result = await Runner.run(agent, f"Rate this joke: {first\_result.final\_output}")  
 print(f"Joke: {first\_result.final\_output}")  
 print(f"Rating: {second\_result.final\_output}")

1. Because the two calls to Runner.run are wrapped in a with trace(), the individual runs will be part of the overall trace rather than creating two traces.

## Creating traces

You can use the [trace()][agents.tracing.trace] function to create a trace. Traces need to be started and finished. You have two options to do so:

1. **Recommended**: use the trace as a context manager, i.e. with trace(...) as my\_trace. This will automatically start and end the trace at the right time.
2. You can also manually call [trace.start()][agents.tracing.Trace.start] and [trace.finish()][agents.tracing.Trace.finish].

The current trace is tracked via a Python [contextvar](https://docs.python.org/3/library/contextvars.html). This means that it works with concurrency automatically. If you manually start/end a trace, you’ll need to pass mark\_as\_current and reset\_current to start()/finish() to update the current trace.

## Creating spans

You can use the various [\*\_span()][agents.tracing.create] methods to create a span. In general, you don’t need to manually create spans. A [custom\_span()][agents.tracing.custom\_span] function is available for tracking custom span information.

Spans are automatically part of the current trace, and are nested under the nearest current span, which is tracked via a Python [contextvar](https://docs.python.org/3/library/contextvars.html).

## Sensitive data

Certain spans may capture potentially sensitive data.

The generation\_span() stores the inputs/outputs of the LLM generation, and function\_span() stores the inputs/outputs of function calls. These may contain sensitive data, so you can disable capturing that data via [RunConfig.trace\_include\_sensitive\_data][agents.run.RunConfig.trace\_include\_sensitive\_data].

Similarly, Audio spans include base64-encoded PCM data for input and output audio by default. You can disable capturing this audio data by configuring [VoicePipelineConfig.trace\_include\_sensitive\_audio\_data][agents.voice.pipeline\_config.VoicePipelineConfig.trace\_include\_sensitive\_audio\_data].

## Custom tracing processors

The high level architecture for tracing is:

* At initialization, we create a global [TraceProvider][agents.tracing.setup.TraceProvider], which is responsible for creating traces.
* We configure the TraceProvider with a [BatchTraceProcessor][agents.tracing.processors.BatchTraceProcessor] that sends traces/spans in batches to a [BackendSpanExporter][agents.tracing.processors.BackendSpanExporter], which exports the spans and traces to the OpenAI backend in batches.

To customize this default setup, to send traces to alternative or additional backends or modifying exporter behavior, you have two options:

1. [add\_trace\_processor()][agents.tracing.add\_trace\_processor] lets you add an **additional** trace processor that will receive traces and spans as they are ready. This lets you do your own processing in addition to sending traces to OpenAI’s backend.
2. [set\_trace\_processors()][agents.tracing.set\_trace\_processors] lets you **replace** the default processors with your own trace processors. This means traces will not be sent to the OpenAI backend unless you include a TracingProcessor that does so.

## External tracing processors list

* [Weights & Biases](https://weave-docs.wandb.ai/guides/integrations/openai_agents)
* [Arize-Phoenix](https://docs.arize.com/phoenix/tracing/integrations-tracing/openai-agents-sdk)
* [Future AGI](https://docs.futureagi.com/future-agi/products/observability/auto-instrumentation/openai_agents)
* [MLflow (self-hosted/OSS](https://mlflow.org/docs/latest/tracing/integrations/openai-agent)
* [MLflow (Databricks hosted](https://docs.databricks.com/aws/en/mlflow/mlflow-tracing#-automatic-tracing)
* [Braintrust](https://braintrust.dev/docs/guides/traces/integrations#openai-agents-sdk)
* [Pydantic Logfire](https://logfire.pydantic.dev/docs/integrations/llms/openai/#openai-agents)
* [AgentOps](https://docs.agentops.ai/v1/integrations/agentssdk)
* [Scorecard](https://docs.scorecard.io/docs/documentation/features/tracing#openai-agents-sdk-integration)
* [Keywords AI](https://docs.keywordsai.co/integration/development-frameworks/openai-agent)
* [LangSmith](https://docs.smith.langchain.com/observability/how_to_guides/trace_with_openai_agents_sdk)
* [Maxim AI](https://www.getmaxim.ai/docs/observe/integrations/openai-agents-sdk)
* [Comet Opik](https://www.comet.com/docs/opik/tracing/integrations/openai_agents)
* [Langfuse](https://langfuse.com/docs/integrations/openaiagentssdk/openai-agents)
* [Langtrace](https://docs.langtrace.ai/supported-integrations/llm-frameworks/openai-agents-sdk)
* [Okahu-Monocle](https://github.com/monocle2ai/monocle) # Agent Visualization

Agent visualization allows you to generate a structured graphical representation of agents and their relationships using **Graphviz**. This is useful for understanding how agents, tools, and handoffs interact within an application.

## Installation

Install the optional viz dependency group:

pip install "openai-agents[viz]"

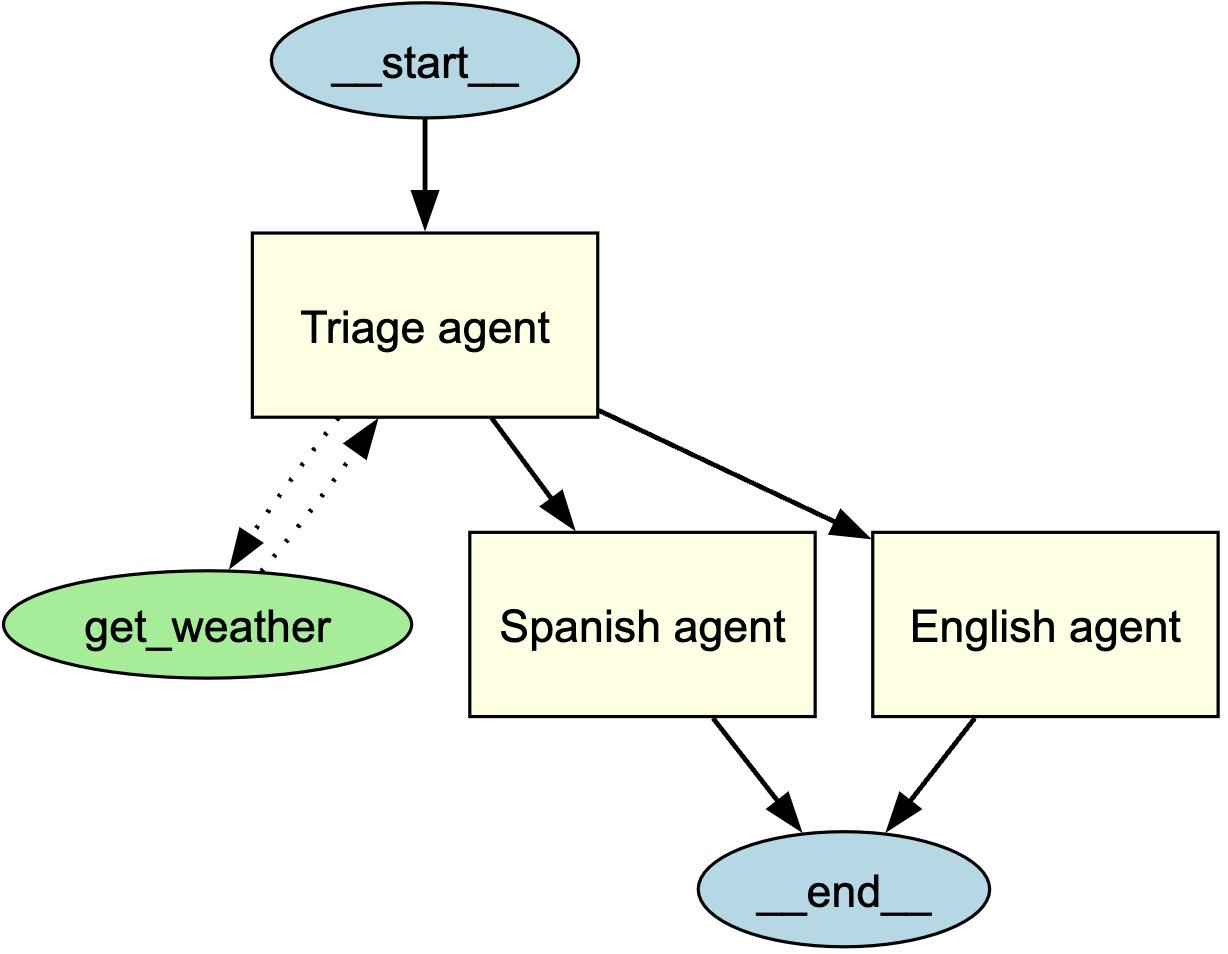
## Generating a Graph

You can generate an agent visualization using the draw\_graph function. This function creates a directed graph where:

* **Agents** are represented as yellow boxes.
* **Tools** are represented as green ellipses.
* **Handoffs** are directed edges from one agent to another.

### Example Usage

from agents import Agent, function\_tool  
from agents.extensions.visualization import draw\_graph  
  
@function\_tool  
def get\_weather(city: str) -> str:  
 return f"The weather in {city} is sunny."  
  
spanish\_agent = Agent(  
 name="Spanish agent",  
 instructions="You only speak Spanish.",  
)  
  
english\_agent = Agent(  
 name="English agent",  
 instructions="You only speak English",  
)  
  
triage\_agent = Agent(  
 name="Triage agent",  
 instructions="Handoff to the appropriate agent based on the language of the request.",  
 handoffs=[spanish\_agent, english\_agent],  
 tools=[get\_weather],  
)  
  
draw\_graph(triage\_agent)



Agent Graph

This generates a graph that visually represents the structure of the **triage agent** and its connections to sub-agents and tools.

## Understanding the Visualization

The generated graph includes:

* A **start node** (\_\_start\_\_) indicating the entry point.
* Agents represented as **rectangles** with yellow fill.
* Tools represented as **ellipses** with green fill.
* Directed edges indicating interactions:
  + **Solid arrows** for agent-to-agent handoffs.
  + **Dotted arrows** for tool invocations.
* An **end node** (\_\_end\_\_) indicating where execution terminates.

## Customizing the Graph

### Showing the Graph

By default, draw\_graph displays the graph inline. To show the graph in a separate window, write the following:

draw\_graph(triage\_agent).view()

### Saving the Graph

By default, draw\_graph displays the graph inline. To save it as a file, specify a filename:

draw\_graph(triage\_agent, filename="agent\_graph")

This will generate agent\_graph.png in the working directory. # Pipelines and workflows

[VoicePipeline][agents.voice.pipeline.VoicePipeline] is a class that makes it easy to turn your agentic workflows into a voice app. You pass in a workflow to run, and the pipeline takes care of transcribing input audio, detecting when the audio ends, calling your workflow at the right time, and turning the workflow output back into audio.

graph LR  
 %% Input  
 A["🎤 Audio Input"]  
  
 %% Voice Pipeline  
 subgraph Voice\_Pipeline [Voice Pipeline]  
 direction TB  
 B["Transcribe (speech-to-text)"]  
 C["Your Code"]:::highlight  
 D["Text-to-speech"]  
 B --> C --> D  
 end  
  
 %% Output  
 E["🎧 Audio Output"]  
  
 %% Flow  
 A --> Voice\_Pipeline  
 Voice\_Pipeline --> E  
  
 %% Custom styling  
 classDef highlight fill:#ffcc66,stroke:#333,stroke-width:1px,font-weight:700;

## Configuring a pipeline

When you create a pipeline, you can set a few things:

1. The [workflow][agents.voice.workflow.VoiceWorkflowBase], which is the code that runs each time new audio is transcribed.
2. The [speech-to-text][agents.voice.model.STTModel] and [text-to-speech][agents.voice.model.TTSModel] models used
3. The [config][agents.voice.pipeline\_config.VoicePipelineConfig], which lets you configure things like:
   * A model provider, which can map model names to models
   * Tracing, including whether to disable tracing, whether audio files are uploaded, the workflow name, trace IDs etc.
   * Settings on the TTS and STT models, like the prompt, language and data types used.

## Running a pipeline

You can run a pipeline via the [run()][agents.voice.pipeline.VoicePipeline.run] method, which lets you pass in audio input in two forms:

1. [AudioInput][agents.voice.input.AudioInput] is used when you have a full audio transcript, and just want to produce a result for it. This is useful in cases where you don’t need to detect when a speaker is done speaking; for example, when you have pre-recorded audio or in push-to-talk apps where it’s clear when the user is done speaking.
2. [StreamedAudioInput][agents.voice.input.StreamedAudioInput] is used when you might need to detect when a user is done speaking. It allows you to push audio chunks as they are detected, and the voice pipeline will automatically run the agent workflow at the right time, via a process called “activity detection”.

## Results

The result of a voice pipeline run is a [StreamedAudioResult][agents.voice.result.StreamedAudioResult]. This is an object that lets you stream events as they occur. There are a few kinds of [VoiceStreamEvent][agents.voice.events.VoiceStreamEvent], including:

1. [VoiceStreamEventAudio][agents.voice.events.VoiceStreamEventAudio], which contains a chunk of audio.
2. [VoiceStreamEventLifecycle][agents.voice.events.VoiceStreamEventLifecycle], which informs you of lifecycle events like a turn starting or ending.
3. [VoiceStreamEventError][agents.voice.events.VoiceStreamEventError], is an error event.

result = await pipeline.run(input)  
  
async for event in result.stream():  
 if event.type == "voice\_stream\_event\_audio":  
 # play audio  
 elif event.type == "voice\_stream\_event\_lifecycle":  
 # lifecycle  
 elif event.type == "voice\_stream\_event\_error"  
 # error  
 ...

## Best practices

### Interruptions

The Agents SDK currently does not support any built-in interruptions support for [StreamedAudioInput][agents.voice.input.StreamedAudioInput]. Instead for every detected turn it will trigger a separate run of your workflow. If you want to handle interruptions inside your application you can listen to the [VoiceStreamEventLifecycle][agents.voice.events.VoiceStreamEventLifecycle] events. turn\_started will indicate that a new turn was transcribed and processing is beginning. turn\_ended will trigger after all the audio was dispatched for a respective turn. You could use these events to mute the microphone of the speaker when the model starts a turn and unmute it after you flushed all the related audio for a turn. # Quickstart

## Prerequisites

Make sure you’ve followed the base [quickstart instructions](../quickstart.md) for the Agents SDK, and set up a virtual environment. Then, install the optional voice dependencies from the SDK:

pip install 'openai-agents[voice]'

## Concepts

The main concept to know about is a [VoicePipeline][agents.voice.pipeline.VoicePipeline], which is a 3 step process:

1. Run a speech-to-text model to turn audio into text.
2. Run your code, which is usually an agentic workflow, to produce a result.
3. Run a text-to-speech model to turn the result text back into audio.

graph LR  
 %% Input  
 A["🎤 Audio Input"]  
  
 %% Voice Pipeline  
 subgraph Voice\_Pipeline [Voice Pipeline]  
 direction TB  
 B["Transcribe (speech-to-text)"]  
 C["Your Code"]:::highlight  
 D["Text-to-speech"]  
 B --> C --> D  
 end  
  
 %% Output  
 E["🎧 Audio Output"]  
  
 %% Flow  
 A --> Voice\_Pipeline  
 Voice\_Pipeline --> E  
  
 %% Custom styling  
 classDef highlight fill:#ffcc66,stroke:#333,stroke-width:1px,font-weight:700;

## Agents

First, let’s set up some Agents. This should feel familiar to you if you’ve built any agents with this SDK. We’ll have a couple of Agents, a handoff, and a tool.

import asyncio  
import random  
  
from agents import (  
 Agent,  
 function\_tool,  
)  
from agents.extensions.handoff\_prompt import prompt\_with\_handoff\_instructions  
  
  
  
@function\_tool  
def get\_weather(city: str) -> str:  
 """Get the weather for a given city."""  
 print(f"[debug] get\_weather called with city: {city}")  
 choices = ["sunny", "cloudy", "rainy", "snowy"]  
 return f"The weather in {city} is {random.choice(choices)}."  
  
  
spanish\_agent = Agent(  
 name="Spanish",  
 handoff\_description="A spanish speaking agent.",  
 instructions=prompt\_with\_handoff\_instructions(  
 "You're speaking to a human, so be polite and concise. Speak in Spanish.",  
 ),  
 model="gpt-4o-mini",  
)  
  
agent = Agent(  
 name="Assistant",  
 instructions=prompt\_with\_handoff\_instructions(  
 "You're speaking to a human, so be polite and concise. If the user speaks in Spanish, handoff to the spanish agent.",  
 ),  
 model="gpt-4o-mini",  
 handoffs=[spanish\_agent],  
 tools=[get\_weather],  
)

## Voice pipeline

We’ll set up a simple voice pipeline, using [SingleAgentVoiceWorkflow][agents.voice.workflow.SingleAgentVoiceWorkflow] as the workflow.

from agents.voice import SingleAgentVoiceWorkflow, VoicePipeline  
pipeline = VoicePipeline(workflow=SingleAgentVoiceWorkflow(agent))

## Run the pipeline

import numpy as np  
import sounddevice as sd  
from agents.voice import AudioInput  
  
# For simplicity, we'll just create 3 seconds of silence  
# In reality, you'd get microphone data  
buffer = np.zeros(24000 \* 3, dtype=np.int16)  
audio\_input = AudioInput(buffer=buffer)  
  
result = await pipeline.run(audio\_input)  
  
# Create an audio player using `sounddevice`  
player = sd.OutputStream(samplerate=24000, channels=1, dtype=np.int16)  
player.start()  
  
# Play the audio stream as it comes in  
async for event in result.stream():  
 if event.type == "voice\_stream\_event\_audio":  
 player.write(event.data)

## Put it all together

import asyncio  
import random  
  
import numpy as np  
import sounddevice as sd  
  
from agents import (  
 Agent,  
 function\_tool,  
 set\_tracing\_disabled,  
)  
from agents.voice import (  
 AudioInput,  
 SingleAgentVoiceWorkflow,  
 VoicePipeline,  
)  
from agents.extensions.handoff\_prompt import prompt\_with\_handoff\_instructions  
  
  
@function\_tool  
def get\_weather(city: str) -> str:  
 """Get the weather for a given city."""  
 print(f"[debug] get\_weather called with city: {city}")  
 choices = ["sunny", "cloudy", "rainy", "snowy"]  
 return f"The weather in {city} is {random.choice(choices)}."  
  
  
spanish\_agent = Agent(  
 name="Spanish",  
 handoff\_description="A spanish speaking agent.",  
 instructions=prompt\_with\_handoff\_instructions(  
 "You're speaking to a human, so be polite and concise. Speak in Spanish.",  
 ),  
 model="gpt-4o-mini",  
)  
  
agent = Agent(  
 name="Assistant",  
 instructions=prompt\_with\_handoff\_instructions(  
 "You're speaking to a human, so be polite and concise. If the user speaks in Spanish, handoff to the spanish agent.",  
 ),  
 model="gpt-4o-mini",  
 handoffs=[spanish\_agent],  
 tools=[get\_weather],  
)  
  
  
async def main():  
 pipeline = VoicePipeline(workflow=SingleAgentVoiceWorkflow(agent))  
 buffer = np.zeros(24000 \* 3, dtype=np.int16)  
 audio\_input = AudioInput(buffer=buffer)  
  
 result = await pipeline.run(audio\_input)  
  
 # Create an audio player using `sounddevice`  
 player = sd.OutputStream(samplerate=24000, channels=1, dtype=np.int16)  
 player.start()  
  
 # Play the audio stream as it comes in  
 async for event in result.stream():  
 if event.type == "voice\_stream\_event\_audio":  
 player.write(event.data)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 asyncio.run(main())

If you run this example, the agent will speak to you! Check out the example in [examples/voice/static](https://github.com/openai/openai-agents-python/tree/main/examples/voice/static) to see a demo where you can speak to the agent yourself. # Tracing

Just like the way [agents are traced](../tracing.md), voice pipelines are also automatically traced.

You can read the tracing doc above for basic tracing information, but you can additionally configure tracing of a pipeline via [VoicePipelineConfig][agents.voice.pipeline\_config.VoicePipelineConfig].

Key tracing related fields are:

* [tracing\_disabled][agents.voice.pipeline\_config.VoicePipelineConfig.tracing\_disabled]: controls whether tracing is disabled. By default, tracing is enabled.
* [trace\_include\_sensitive\_data][agents.voice.pipeline\_config.VoicePipelineConfig.trace\_include\_sensitive\_data]: controls whether traces include potentially sensitive data, like audio transcripts. This is specifically for the voice pipeline, and not for anything that goes on inside your Workflow.
* [trace\_include\_sensitive\_audio\_data][agents.voice.pipeline\_config.VoicePipelineConfig.trace\_include\_sensitive\_audio\_data]: controls whether traces include audio data.
* [workflow\_name][agents.voice.pipeline\_config.VoicePipelineConfig.workflow\_name]: The name of the trace workflow.
* [group\_id][agents.voice.pipeline\_config.VoicePipelineConfig.group\_id]: The group\_id of the trace, which lets you link multiple traces.
* [trace\_metadata][agents.voice.pipeline\_config.VoicePipelineConfig.tracing\_disabled]: Additional metadata to include with the trace.