OpenAI Agents SDK

The OpenAI Agents SDK enables you to build agentic AI apps in a lightweight, easy-to-use package with very few abstractions. It's a production-ready upgrade of our previous experimentation for agents, Swarm. The Agents SDK has a very small set of primitives:

Agents, which are LLMs equipped with instructions 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:

Enough features to be worth using, but few enough primitives to make it quick to learn.

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-...

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 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 to view traces of your agent runs.

Next steps

Learn how to build more complex agentic flows:

Learn about how to configure Agents.

Learn about running agents.

Learn about tools, guardrails and models.

# **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:** Examples in this category illustrate common agent design patterns, such as
  + Deterministic workflows
  + Agents as tools
  + Parallel agent execution
* **basic:** These examples showcase foundational capabilities of the SDK, such as
  + Dynamic system prompts
  + Streaming outputs
  + Lifecycle events
* **tool examples:** Learn how to implement OAI hosted tools such as web search and file search, and integrate them into your agents.
* **model providers:** Explore how to use non-OpenAI models with the SDK.
* **handoffs:** See practical examples of agent handoffs.
* **customer\_service** and **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.

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 objects, but we support any type that can be wrapped in a Pydantic TypeAdapter - 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 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 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 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 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. Valid values are:

auto, which allows the LLM to decide whether or not to use a tool.

required, which requires the LLM to use a tool (but it can intelligently decide which tool).

none, which requires the LLM to not use a tool.

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. 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.

Running agents

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

Runner.run(), which runs async and returns a RunResult.

Runner.run\_sync(), which is a sync method and just runs .run() under the hood.

Runner.run\_streamed(), which runs async and returns a 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.

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:

We call the LLM for the current agent, with the current input.

The LLM produces its output.

If the LLM returns a final\_output, the loop ends and we return the result.

If the LLM does a handoff, we update the current agent and input, and re-run the loop.

If the LLM produces tool calls, we run those tool calls, append the results, and re-run the loop.

If we exceed the max\_turns passed, we raise a 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 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.

Run config

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

model: Allows setting a global LLM model to use, irrespective of what model each Agent has.

model\_provider: A model provider for looking up model names, which defaults to OpenAI.

model\_settings: Overrides agent-specific settings. For example, you can set a global temperature or top\_p.

input\_guardrails, output\_guardrails: A list of input or output guardrails to include on all runs.

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 for more details.

tracing\_disabled: Allows you to disable tracing for the entire run.

trace\_include\_sensitive\_data: Configures whether traces will include potentially sensitive data, such as LLM and tool call inputs/outputs.

workflow\_name, trace\_id, group\_id: Sets the tracing workflow name, trace ID and trace group ID for the run. We recommend at least setting workflow\_name. The session ID is an optional field that lets you link traces across multiple runs.

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:

User turn: user enter text

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() 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 is the base class for all exceptions raised in the SDK.

MaxTurnsExceeded is raised when the run exceeds the max\_turns passed to the run methods.

ModelBehaviorError is raised when the model produces invalid outputs, e.g. malformed JSON or using non-existent tools.

UserError is raised when you (the person writing code using the SDK) make an error using the SDK.

InputGuardrailTripwireTriggered, OutputGuardrailTripwireTriggered is raised when a guardrail is tripped.

Results

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

RunResult if you call run or run\_sync

RunResultStreaming if you call run\_streamed

Both of these inherit from RunResultBase, which is where most useful information is present.

Final output

The 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() 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 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 property contains the new items generated during the run. The items are RunItems. A run item wraps the raw item generated by the LLM.

MessageOutputItem indicates a message from the LLM. The raw item is the message generated.

HandoffCallItem indicates that the LLM called the handoff tool. The raw item is the tool call item from the LLM.

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 indicates that the LLM invoked a tool.

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 indicates a reasoning item from the LLM. The raw item is the reasoning generated.

Other information

Guardrail results

The input\_guardrail\_results and 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 property contains the ModelResponses generated by the LLM.

Original input

The 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.

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(), which will give you a RunResultStreaming. Calling result.stream\_events() gives you an async stream of StreamEvent objects, which are described below.

Raw response 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

RunItemStreamEvents 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 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:

The WebSearchTool lets an agent search the web.

The FileSearchTool allows retrieving information from your OpenAI Vector Stores.

The 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 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

async def fetch\_weather(location: Location) -> str:

"""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")

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],

)

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()

Expand to see output

Custom function tools

Sometimes, you don't want to use a Python function as a tool. You can directly create a 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:

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.

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)

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.

Model context protocol (MCP)

The Model context protocol (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:

stdio servers run as a subprocess of your application. You can think of them as running "locally".

HTTP over SSE servers run remotely. You connect to them via a URL.

You can use the MCPServerStdio and MCPServerSse classes to connect to these servers.

For example, this is how you'd use the official MCP filesystem server.

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 and 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.

Tracing

Tracing automatically captures MCP operations, including:

Calls to the MCP server to list tools

MCP-related info on function calls

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 param, which can either take an Agent directly, or a Handoff object that customizes the Handoff.

You can create a handoff using the 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")

triage\_agent = Agent(name="Triage agent", handoffs=[billing\_agent, handoff(refund\_agent)])

Customizing handoffs via the handoff() function

The 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. An input filter is a function that receives the existing input via a 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,

)

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>.""",

)

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, 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:

You can globally disable tracing by setting the env var OPENAI\_AGENTS\_DISABLE\_TRACING=1

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.

In addition, you can set up custom trace 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"):

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}")

Creating traces

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

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.

You can also manually call trace.start() and trace.finish().

The current trace is tracked via a Python contextvar. 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() methods to create a span. In general, you don't need to manually create spans. A 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.

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.

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.

Custom tracing processors

The high level architecture for tracing is:

At initialization, we create a global TraceProvider, which is responsible for creating traces.

We configure the TraceProvider with a BatchTraceProcessor that sends traces/spans in batches to a 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:

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.

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

Arize-Phoenix

MLflow

Braintrust

Pydantic Logfire

AgentOps

Scorecard

Keywords AI

LangSmith

Maxim AI

Comet Opik

Langfuse

Context management

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

Context available locally to your code: this is data and dependencies you might need when tool functions run, during callbacks like on\_handoff, in lifecycle hooks, etc.

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

Local context

This is represented via the RunContextWrapper class and the context property within it. The way this works is:

You create any Python object you want. A common pattern is to use a dataclass or a Pydantic object.

You pass that object to the various run methods (e.g. Runner.run(..., \*\*context=whatever\*\*)).

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

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:

name: str

uid: int

@function\_tool

async def fetch\_user\_age(wrapper: RunContextWrapper[UserInfo]) -> str:

return f"User {wrapper.context.name} is 47 years old"

async def main():

user\_info = UserInfo(name="John", uid=123)

agent = Agent[UserInfo](

name="Assistant",

tools=[fetch\_user\_age],

)

result = await Runner.run(

starting\_agent=agent,

input="What is the age of the user?",

context=user\_info,

)

print(result.final\_output)

# The user John is 47 years old.

if \_\_name\_\_ == "\_\_main\_\_":

asyncio.run(main())

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:

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).

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.

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.

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.

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:

Input guardrails run on the initial user input

Output guardrails run on the final agent output

Input guardrails

Input guardrails run in 3 steps:

First, the guardrail receives the same input passed to the agent.

Next, the guardrail function runs to produce a GuardrailFunctionOutput, which is then wrapped in an InputGuardrailResult

Finally, we check if .tripwire\_triggered is true. If true, an 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:

First, the guardrail receives the same input passed to the agent.

Next, the guardrail function runs to produce a GuardrailFunctionOutput, which is then wrapped in an OutputGuardrailResult

Finally, we check if .tripwire\_triggered is true. If true, an 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. 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(

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(

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,

tripwire\_triggered=result.final\_output.is\_math\_homework,

)

agent = Agent(

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")

Output guardrails are similar.

from pydantic import BaseModel

from agents import (

Agent,

GuardrailFunctionOutput,

OutputGuardrailTripwireTriggered,

RunContextWrapper,

Runner,

output\_guardrail,

)

class MessageOutput(BaseModel):

response: str

class MathOutput(BaseModel):

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(

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(

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")

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:

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.

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:

Invest in good prompts. Make it clear what tools are available, how to use them, and what parameters it must operate within.

Monitor your app and iterate on it. See where things go wrong, and iterate on your prompts.

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.

Have specialized agents that excel in one task, rather than having a general purpose agent that is expected to be good at anything.

Invest in 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 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.

Models

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

Recommended: the OpenAIResponsesModel, which calls OpenAI APIs using the new Responses API.

The OpenAIChatCompletionsModel, which calls OpenAI APIs using the Chat Completions API.

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, you can select a specific model by either:

Passing the name of an OpenAI model.

Passing any model name + a ModelProvider that can map that name to a Model instance.

Directly providing a Model implementation.

Note

While our SDK supports both the OpenAIResponsesModel and the 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",

)

english\_agent = Agent(

name="English agent",

instructions="You only speak English",

model=OpenAIChatCompletionsModel(

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)

Using other LLM providers

You can use other LLM providers in 3 ways (examples here):

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.

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.

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.

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.

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.

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:

Disable tracing entirely: set\_tracing\_disabled(True).

Set an OpenAI key for tracing: set\_tracing\_export\_api\_key(...). This API key will only be used for uploading traces, and must be from platform.openai.com.

Use a non-OpenAI trace processor. See the tracing docs.

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:

Call set\_default\_openai\_api("chat\_completions"). This works if you are setting OPENAI\_API\_KEY and OPENAI\_BASE\_URL via environment vars.

Use OpenAIChatCompletionsModel. There are examples here.

Structured outputs support

Some model providers don't have support for 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.

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() 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() 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() 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 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() 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() 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.

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

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.png")

This will generate agent\_graph.png in the working directory.

Voice Agents:  
Quickstart

Prerequisites

Make sure you've followed the base quickstart instructions 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, which is a 3 step process:

Run a speech-to-text model to turn audio into text.

Run your code, which is usually an agentic workflow, to produce a result.

Run a text-to-speech model to turn the result text back into audio.

Voice Pipeline

Transcribe (speech-to-text)

Your Code

Text-to-speech

🎤 Audio Input

🎧 Audio Output

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 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 to see a demo where you can speak to the agent yourself.

Pipelines and workflows

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.

Voice Pipeline

Transcribe (speech-to-text)

Your Code

Text-to-speech

🎤 Audio Input

🎧 Audio Output

Configuring a pipeline

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

The workflow, which is the code that runs each time new audio is transcribed.

The speech-to-text and text-to-speech models used

The config, 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() method, which lets you pass in audio input in two forms:

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.

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. This is an object that lets you stream events as they occur. There are a few kinds of VoiceStreamEvent, including:

VoiceStreamEventAudio, which contains a chunk of audio.

VoiceStreamEventLifecycle, which informs you of lifecycle events like a turn starting or ending.

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. 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 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.

Tracing

Just like the way agents are traced, 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.

Key tracing related fields are:

tracing\_disabled: controls whether tracing is disabled. By default, tracing is enabled.

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: controls whether traces include audio data.

workflow\_name: The name of the trace workflow.

group\_id: The group\_id of the trace, which lets you link multiple traces.

trace\_metadata: Additional metadata to include with the trace.

Agents:

Agents module

set\_default\_openai\_key

set\_default\_openai\_key(

key: str, use\_for\_tracing: bool = True

) -> None

Set the default OpenAI API key to use for LLM requests (and optionally tracing(). This is only necessary if the OPENAI\_API\_KEY environment variable is not already set.

If provided, this key will be used instead of the OPENAI\_API\_KEY environment variable.

Parameters:

Name Type Description Default

key str The OpenAI key to use. required

use\_for\_tracing bool Whether to also use this key to send traces to OpenAI. Defaults to True If False, you'll either need to set the OPENAI\_API\_KEY environment variable or call set\_tracing\_export\_api\_key() with the API key you want to use for tracing. True

Source code in src/agents/\_\_init\_\_.py

def set\_default\_openai\_key(key: str, use\_for\_tracing: bool = True) -> None:

"""Set the default OpenAI API key to use for LLM requests (and optionally tracing(). This is

only necessary if the OPENAI\_API\_KEY environment variable is not already set.

If provided, this key will be used instead of the OPENAI\_API\_KEY environment variable.

Args:

key: The OpenAI key to use.

use\_for\_tracing: Whether to also use this key to send traces to OpenAI. Defaults to True

If False, you'll either need to set the OPENAI\_API\_KEY environment variable or call

set\_tracing\_export\_api\_key() with the API key you want to use for tracing.

"""

\_config.set\_default\_openai\_key(key, use\_for\_tracing)

set\_default\_openai\_client

set\_default\_openai\_client(

client: AsyncOpenAI, use\_for\_tracing: bool = True

) -> None

Set the default OpenAI client to use for LLM requests and/or tracing. If provided, this client will be used instead of the default OpenAI client.

Parameters:

Name Type Description Default

client AsyncOpenAI The OpenAI client to use. required

use\_for\_tracing bool Whether to use the API key from this client for uploading traces. If False, you'll either need to set the OPENAI\_API\_KEY environment variable or call set\_tracing\_export\_api\_key() with the API key you want to use for tracing. True

Source code in src/agents/\_\_init\_\_.py

def set\_default\_openai\_client(client: AsyncOpenAI, use\_for\_tracing: bool = True) -> None:

"""Set the default OpenAI client to use for LLM requests and/or tracing. If provided, this

client will be used instead of the default OpenAI client.

Args:

client: The OpenAI client to use.

use\_for\_tracing: Whether to use the API key from this client for uploading traces. If False,

you'll either need to set the OPENAI\_API\_KEY environment variable or call

set\_tracing\_export\_api\_key() with the API key you want to use for tracing.

"""

\_config.set\_default\_openai\_client(client, use\_for\_tracing)

set\_default\_openai\_api

set\_default\_openai\_api(

api: Literal["chat\_completions", "responses"],

) -> None

Set the default API to use for OpenAI LLM requests. By default, we will use the responses API but you can set this to use the chat completions API instead.

Source code in src/agents/\_\_init\_\_.py

def set\_default\_openai\_api(api: Literal["chat\_completions", "responses"]) -> None:

"""Set the default API to use for OpenAI LLM requests. By default, we will use the responses API

but you can set this to use the chat completions API instead.

"""

\_config.set\_default\_openai\_api(api)

set\_tracing\_export\_api\_key

set\_tracing\_export\_api\_key(api\_key: str) -> None

Set the OpenAI API key for the backend exporter.

Source code in src/agents/tracing/\_\_init\_\_.py

def set\_tracing\_export\_api\_key(api\_key: str) -> None:

"""

Set the OpenAI API key for the backend exporter.

"""

default\_exporter().set\_api\_key(api\_key)

set\_tracing\_disabled

set\_tracing\_disabled(disabled: bool) -> None

Set whether tracing is globally disabled.

Source code in src/agents/tracing/\_\_init\_\_.py

def set\_tracing\_disabled(disabled: bool) -> None:

"""

Set whether tracing is globally disabled.

"""

GLOBAL\_TRACE\_PROVIDER.set\_disabled(disabled)

set\_trace\_processors

set\_trace\_processors(

processors: list[TracingProcessor],

) -> None

Set the list of trace processors. This will replace the current list of processors.

Source code in src/agents/tracing/\_\_init\_\_.py

def set\_trace\_processors(processors: list[TracingProcessor]) -> None:

"""

Set the list of trace processors. This will replace the current list of processors.

"""

GLOBAL\_TRACE\_PROVIDER.set\_processors(processors)

enable\_verbose\_stdout\_logging

enable\_verbose\_stdout\_logging()

Enables verbose logging to stdout. This is useful for debugging.

Source code in src/agents/\_\_init\_\_.py

def enable\_verbose\_stdout\_logging():

"""Enables verbose logging to stdout. This is useful for debugging."""

logger = logging.getLogger("openai.agents")

logger.setLevel(logging.DEBUG)

logger.addHandler(logging.StreamHandler(sys.stdout))

Agents

ToolsToFinalOutputFunction module-attribute

ToolsToFinalOutputFunction: TypeAlias = Callable[

[RunContextWrapper[TContext], list[FunctionToolResult]],

MaybeAwaitable[ToolsToFinalOutputResult],

]

A function that takes a run context and a list of tool results, and returns a ToolToFinalOutputResult.

ToolsToFinalOutputResult dataclass

Source code in src/agents/agent.py

@dataclass

class ToolsToFinalOutputResult:

is\_final\_output: bool

"""Whether this is the final output. If False, the LLM will run again and receive the tool call

output.

"""

final\_output: Any | None = None

"""The final output. Can be None if `is\_final\_output` is False, otherwise must match the

`output\_type` of the agent.

"""

is\_final\_output instance-attribute

is\_final\_output: bool

Whether this is the final output. If False, the LLM will run again and receive the tool call output.

final\_output class-attribute instance-attribute

final\_output: Any | None = None

The final output. Can be None if is\_final\_output is False, otherwise must match the output\_type of the agent.

StopAtTools

Bases: TypedDict

Source code in src/agents/agent.py

class StopAtTools(TypedDict):

stop\_at\_tool\_names: list[str]

"""A list of tool names, any of which will stop the agent from running further."""

stop\_at\_tool\_names instance-attribute

stop\_at\_tool\_names: list[str]

A list of tool names, any of which will stop the agent from running further.

Agent dataclass

Bases: Generic[TContext]

An agent is an AI model configured with instructions, tools, guardrails, handoffs and more.

We strongly recommend passing instructions, which is the "system prompt" for the agent. In addition, you can pass handoff\_description, which is a human-readable description of the agent, used when the agent is used inside tools/handoffs.

Agents are generic on the context type. The context is a (mutable) object you create. It is passed to tool functions, handoffs, guardrails, etc.

Source code in src/agents/agent.py

@dataclass

class Agent(Generic[TContext]):

"""An agent is an AI model configured with instructions, tools, guardrails, handoffs and more.

We strongly recommend passing `instructions`, which is the "system prompt" for the agent. In

addition, you can pass `handoff\_description`, which is a human-readable description of the

agent, used when the agent is used inside tools/handoffs.

Agents are generic on the context type. The context is a (mutable) object you create. It is

passed to tool functions, handoffs, guardrails, etc.

"""

name: str

"""The name of the agent."""

instructions: (

str

| Callable[

[RunContextWrapper[TContext], Agent[TContext]],

MaybeAwaitable[str],

]

| None

) = None

"""The instructions for the agent. Will be used as the "system prompt" when this agent is

invoked. Describes what the agent should do, and how it responds.

Can either be a string, or a function that dynamically generates instructions for the agent. If

you provide a function, it will be called with the context and the agent instance. It must

return a string.

"""

handoff\_description: str | None = None

"""A description of the agent. This is used when the agent is used as a handoff, so that an

LLM knows what it does and when to invoke it.

"""

handoffs: list[Agent[Any] | Handoff[TContext]] = field(default\_factory=list)

"""Handoffs are sub-agents that the agent can delegate to. You can provide a list of handoffs,

and the agent can choose to delegate to them if relevant. Allows for separation of concerns and

modularity.

"""

model: str | Model | None = None

"""The model implementation to use when invoking the LLM.

By default, if not set, the agent will use the default model configured in

`model\_settings.DEFAULT\_MODEL`.

"""

model\_settings: ModelSettings = field(default\_factory=ModelSettings)

"""Configures model-specific tuning parameters (e.g. temperature, top\_p).

"""

tools: list[Tool] = field(default\_factory=list)

"""A list of tools that the agent can use."""

mcp\_servers: list[MCPServer] = field(default\_factory=list)

"""A list of [Model Context Protocol](https://modelcontextprotocol.io/) servers that

the agent can use. Every time the agent runs, it will include tools from these servers in the

list of available tools.

NOTE: You are expected to manage the lifecycle of these servers. Specifically, you must call

`server.connect()` before passing it to the agent, and `server.cleanup()` when the server is no

longer needed.

"""

input\_guardrails: list[InputGuardrail[TContext]] = field(default\_factory=list)

"""A list of checks that run in parallel to the agent's execution, before generating a

response. Runs only if the agent is the first agent in the chain.

"""

output\_guardrails: list[OutputGuardrail[TContext]] = field(default\_factory=list)

"""A list of checks that run on the final output of the agent, after generating a response.

Runs only if the agent produces a final output.

"""

output\_type: type[Any] | None = None

"""The type of the output object. If not provided, the output will be `str`."""

hooks: AgentHooks[TContext] | None = None

"""A class that receives callbacks on various lifecycle events for this agent.

"""

tool\_use\_behavior: (

Literal["run\_llm\_again", "stop\_on\_first\_tool"] | StopAtTools | ToolsToFinalOutputFunction

) = "run\_llm\_again"

"""This lets you configure how tool use is handled.

- "run\_llm\_again": The default behavior. Tools are run, and then the LLM receives the results

and gets to respond.

- "stop\_on\_first\_tool": The output of the first tool call is used as the final output. This

means that the LLM does not process the result of the tool call.

- A list of tool names: The agent will stop running if any of the tools in the list are called.

The final output will be the output of the first matching tool call. The LLM does not

process the result of the tool call.

- A function: If you pass a function, it will be called with the run context and the list of

tool results. It must return a `ToolToFinalOutputResult`, which determines whether the tool

calls result in a final output.

NOTE: This configuration is specific to FunctionTools. Hosted tools, such as file search,

web search, etc are always processed by the LLM.

"""

reset\_tool\_choice: bool = True

"""Whether to reset the tool choice to the default value after a tool has been called. Defaults

to True. This ensures that the agent doesn't enter an infinite loop of tool usage."""

def clone(self, \*\*kwargs: Any) -> Agent[TContext]:

"""Make a copy of the agent, with the given arguments changed. For example, you could do:

```

new\_agent = agent.clone(instructions="New instructions")

```

"""

return dataclasses.replace(self, \*\*kwargs)

def as\_tool(

self,

tool\_name: str | None,

tool\_description: str | None,

custom\_output\_extractor: Callable[[RunResult], Awaitable[str]] | None = None,

) -> Tool:

"""Transform this agent into a tool, callable by other agents.

This is different from handoffs in two ways:

1. In handoffs, the new agent receives the conversation history. In this tool, the new agent

receives generated input.

2. In handoffs, the new agent takes over the conversation. In this tool, the new agent is

called as a tool, and the conversation is continued by the original agent.

Args:

tool\_name: The name of the tool. If not provided, the agent's name will be used.

tool\_description: The description of the tool, which should indicate what it does and

when to use it.

custom\_output\_extractor: A function that extracts the output from the agent. If not

provided, the last message from the agent will be used.

"""

@function\_tool(

name\_override=tool\_name or \_transforms.transform\_string\_function\_style(self.name),

description\_override=tool\_description or "",

)

async def run\_agent(context: RunContextWrapper, input: str) -> str:

from .run import Runner

output = await Runner.run(

starting\_agent=self,

input=input,

context=context.context,

)

if custom\_output\_extractor:

return await custom\_output\_extractor(output)

return ItemHelpers.text\_message\_outputs(output.new\_items)

return run\_agent

async def get\_system\_prompt(self, run\_context: RunContextWrapper[TContext]) -> str | None:

"""Get the system prompt for the agent."""

if isinstance(self.instructions, str):

return self.instructions

elif callable(self.instructions):

if inspect.iscoroutinefunction(self.instructions):

return await cast(Awaitable[str], self.instructions(run\_context, self))

else:

return cast(str, self.instructions(run\_context, self))

elif self.instructions is not None:

logger.error(f"Instructions must be a string or a function, got {self.instructions}")

return None

async def get\_mcp\_tools(self) -> list[Tool]:

"""Fetches the available tools from the MCP servers."""

return await MCPUtil.get\_all\_function\_tools(self.mcp\_servers)

async def get\_all\_tools(self) -> list[Tool]:

"""All agent tools, including MCP tools and function tools."""

mcp\_tools = await self.get\_mcp\_tools()

return mcp\_tools + self.tools

name instance-attribute

name: str

The name of the agent.

instructions class-attribute instance-attribute

instructions: (

str

| Callable[

[RunContextWrapper[TContext], Agent[TContext]],

MaybeAwaitable[str],

]

| None

) = None

The instructions for the agent. Will be used as the "system prompt" when this agent is invoked. Describes what the agent should do, and how it responds.

Can either be a string, or a function that dynamically generates instructions for the agent. If you provide a function, it will be called with the context and the agent instance. It must return a string.

handoff\_description class-attribute instance-attribute

handoff\_description: str | None = None

A description of the agent. This is used when the agent is used as a handoff, so that an LLM knows what it does and when to invoke it.

handoffs class-attribute instance-attribute

handoffs: list[Agent[Any] | Handoff[TContext]] = field(

default\_factory=list

)

Handoffs are sub-agents that the agent can delegate to. You can provide a list of handoffs, and the agent can choose to delegate to them if relevant. Allows for separation of concerns and modularity.

model class-attribute instance-attribute

model: str | Model | None = None

The model implementation to use when invoking the LLM.

By default, if not set, the agent will use the default model configured in model\_settings.DEFAULT\_MODEL.

model\_settings class-attribute instance-attribute

model\_settings: ModelSettings = field(

default\_factory=ModelSettings

)

Configures model-specific tuning parameters (e.g. temperature, top\_p).

tools class-attribute instance-attribute

tools: list[Tool] = field(default\_factory=list)

A list of tools that the agent can use.

mcp\_servers class-attribute instance-attribute

mcp\_servers: list[MCPServer] = field(default\_factory=list)

A list of Model Context Protocol servers that the agent can use. Every time the agent runs, it will include tools from these servers in the list of available tools.

NOTE: You are expected to manage the lifecycle of these servers. Specifically, you must call server.connect() before passing it to the agent, and server.cleanup() when the server is no longer needed.

input\_guardrails class-attribute instance-attribute

input\_guardrails: list[InputGuardrail[TContext]] = field(

default\_factory=list

)

A list of checks that run in parallel to the agent's execution, before generating a response. Runs only if the agent is the first agent in the chain.

output\_guardrails class-attribute instance-attribute

output\_guardrails: list[OutputGuardrail[TContext]] = field(

default\_factory=list

)

A list of checks that run on the final output of the agent, after generating a response. Runs only if the agent produces a final output.

output\_type class-attribute instance-attribute

output\_type: type[Any] | None = None

The type of the output object. If not provided, the output will be str.

hooks class-attribute instance-attribute

hooks: AgentHooks[TContext] | None = None

A class that receives callbacks on various lifecycle events for this agent.

tool\_use\_behavior class-attribute instance-attribute

tool\_use\_behavior: (

Literal["run\_llm\_again", "stop\_on\_first\_tool"]

| StopAtTools

| ToolsToFinalOutputFunction

) = "run\_llm\_again"

This lets you configure how tool use is handled. - "run\_llm\_again": The default behavior. Tools are run, and then the LLM receives the results and gets to respond. - "stop\_on\_first\_tool": The output of the first tool call is used as the final output. This means that the LLM does not process the result of the tool call. - A list of tool names: The agent will stop running if any of the tools in the list are called. The final output will be the output of the first matching tool call. The LLM does not process the result of the tool call. - A function: If you pass a function, it will be called with the run context and the list of tool results. It must return a ToolToFinalOutputResult, which determines whether the tool calls result in a final output.

NOTE: This configuration is specific to FunctionTools. Hosted tools, such as file search, web search, etc are always processed by the LLM.

reset\_tool\_choice class-attribute instance-attribute

reset\_tool\_choice: bool = True

Whether to reset the tool choice to the default value after a tool has been called. Defaults to True. This ensures that the agent doesn't enter an infinite loop of tool usage.

clone

clone(\*\*kwargs: Any) -> Agent[TContext]

Make a copy of the agent, with the given arguments changed. For example, you could do:

new\_agent = agent.clone(instructions="New instructions")

Source code in src/agents/agent.py

def clone(self, \*\*kwargs: Any) -> Agent[TContext]:

"""Make a copy of the agent, with the given arguments changed. For example, you could do:

```

new\_agent = agent.clone(instructions="New instructions")

```

"""

return dataclasses.replace(self, \*\*kwargs)

as\_tool

as\_tool(

tool\_name: str | None,

tool\_description: str | None,

custom\_output\_extractor: Callable[

[RunResult], Awaitable[str]

]

| None = None,

) -> Tool

Transform this agent into a tool, callable by other agents.

This is different from handoffs in two ways: 1. In handoffs, the new agent receives the conversation history. In this tool, the new agent receives generated input. 2. In handoffs, the new agent takes over the conversation. In this tool, the new agent is called as a tool, and the conversation is continued by the original agent.

Parameters:

Name Type Description Default

tool\_name str | None The name of the tool. If not provided, the agent's name will be used. required

tool\_description str | None The description of the tool, which should indicate what it does and when to use it. required

custom\_output\_extractor Callable[[RunResult], Awaitable[str]] | None A function that extracts the output from the agent. If not provided, the last message from the agent will be used. None

Source code in src/agents/agent.py

def as\_tool(

self,

tool\_name: str | None,

tool\_description: str | None,

custom\_output\_extractor: Callable[[RunResult], Awaitable[str]] | None = None,

) -> Tool:

"""Transform this agent into a tool, callable by other agents.

This is different from handoffs in two ways:

1. In handoffs, the new agent receives the conversation history. In this tool, the new agent

receives generated input.

2. In handoffs, the new agent takes over the conversation. In this tool, the new agent is

called as a tool, and the conversation is continued by the original agent.

Args:

tool\_name: The name of the tool. If not provided, the agent's name will be used.

tool\_description: The description of the tool, which should indicate what it does and

when to use it.

custom\_output\_extractor: A function that extracts the output from the agent. If not

provided, the last message from the agent will be used.

"""

@function\_tool(

name\_override=tool\_name or \_transforms.transform\_string\_function\_style(self.name),

description\_override=tool\_description or "",

)

async def run\_agent(context: RunContextWrapper, input: str) -> str:

from .run import Runner

output = await Runner.run(

starting\_agent=self,

input=input,

context=context.context,

)

if custom\_output\_extractor:

return await custom\_output\_extractor(output)

return ItemHelpers.text\_message\_outputs(output.new\_items)

return run\_agent

get\_system\_prompt async

get\_system\_prompt(

run\_context: RunContextWrapper[TContext],

) -> str | None

Get the system prompt for the agent.

Source code in src/agents/agent.py

async def get\_system\_prompt(self, run\_context: RunContextWrapper[TContext]) -> str | None:

"""Get the system prompt for the agent."""

if isinstance(self.instructions, str):

return self.instructions

elif callable(self.instructions):

if inspect.iscoroutinefunction(self.instructions):

return await cast(Awaitable[str], self.instructions(run\_context, self))

else:

return cast(str, self.instructions(run\_context, self))

elif self.instructions is not None:

logger.error(f"Instructions must be a string or a function, got {self.instructions}")

return None

get\_mcp\_tools async

get\_mcp\_tools() -> list[Tool]

Fetches the available tools from the MCP servers.

Source code in src/agents/agent.py

async def get\_mcp\_tools(self) -> list[Tool]:

"""Fetches the available tools from the MCP servers."""

return await MCPUtil.get\_all\_function\_tools(self.mcp\_servers)

get\_all\_tools async

get\_all\_tools() -> list[Tool]

All agent tools, including MCP tools and function tools.

Source code in src/agents/agent.py

async def get\_all\_tools(self) -> list[Tool]:

"""All agent tools, including MCP tools and function tools."""

mcp\_tools = await self.get\_mcp\_tools()

return mcp\_tools + self.tools

Runner

Runner

Source code in src/agents/run.py

class Runner:

@classmethod

async def run(

cls,

starting\_agent: Agent[TContext],

input: str | list[TResponseInputItem],

\*,

context: TContext | None = None,

max\_turns: int = DEFAULT\_MAX\_TURNS,

hooks: RunHooks[TContext] | None = None,

run\_config: RunConfig | None = None,

) -> RunResult:

"""Run a workflow starting at the given agent. The agent will run in a loop until a final

output is generated. The loop runs like so:

1. The agent is invoked with the given input.

2. If there is a final output (i.e. the agent produces something of type

`agent.output\_type`, the loop terminates.

3. If there's a handoff, we run the loop again, with the new agent.

4. Else, we run tool calls (if any), and re-run the loop.

In two cases, the agent may raise an exception:

1. If the max\_turns is exceeded, a MaxTurnsExceeded exception is raised.

2. If a guardrail tripwire is triggered, a GuardrailTripwireTriggered exception is raised.

Note that only the first agent's input guardrails are run.

Args:

starting\_agent: The starting agent to run.

input: The initial input to the agent. You can pass a single string for a user message,

or a list of input items.

context: The context to run the agent with.

max\_turns: The maximum number of turns to run the agent for. A turn is defined as one

AI invocation (including any tool calls that might occur).

hooks: An object that receives callbacks on various lifecycle events.

run\_config: Global settings for the entire agent run.

Returns:

A run result containing all the inputs, guardrail results and the output of the last

agent. Agents may perform handoffs, so we don't know the specific type of the output.

"""

if hooks is None:

hooks = RunHooks[Any]()

if run\_config is None:

run\_config = RunConfig()

tool\_use\_tracker = AgentToolUseTracker()

with TraceCtxManager(

workflow\_name=run\_config.workflow\_name,

trace\_id=run\_config.trace\_id,

group\_id=run\_config.group\_id,

metadata=run\_config.trace\_metadata,

disabled=run\_config.tracing\_disabled,

):

current\_turn = 0

original\_input: str | list[TResponseInputItem] = copy.deepcopy(input)

generated\_items: list[RunItem] = []

model\_responses: list[ModelResponse] = []

context\_wrapper: RunContextWrapper[TContext] = RunContextWrapper(

context=context, # type: ignore

)

input\_guardrail\_results: list[InputGuardrailResult] = []

current\_span: Span[AgentSpanData] | None = None

current\_agent = starting\_agent

should\_run\_agent\_start\_hooks = True

try:

while True:

# Start an agent span if we don't have one. This span is ended if the current

# agent changes, or if the agent loop ends.

if current\_span is None:

handoff\_names = [h.agent\_name for h in cls.\_get\_handoffs(current\_agent)]

if output\_schema := cls.\_get\_output\_schema(current\_agent):

output\_type\_name = output\_schema.output\_type\_name()

else:

output\_type\_name = "str"

current\_span = agent\_span(

name=current\_agent.name,

handoffs=handoff\_names,

output\_type=output\_type\_name,

)

current\_span.start(mark\_as\_current=True)

all\_tools = await cls.\_get\_all\_tools(current\_agent)

current\_span.span\_data.tools = [t.name for t in all\_tools]

current\_turn += 1

if current\_turn > max\_turns:

\_error\_tracing.attach\_error\_to\_span(

current\_span,

SpanError(

message="Max turns exceeded",

data={"max\_turns": max\_turns},

),

)

raise MaxTurnsExceeded(f"Max turns ({max\_turns}) exceeded")

logger.debug(

f"Running agent {current\_agent.name} (turn {current\_turn})",

)

if current\_turn == 1:

input\_guardrail\_results, turn\_result = await asyncio.gather(

cls.\_run\_input\_guardrails(

starting\_agent,

starting\_agent.input\_guardrails

+ (run\_config.input\_guardrails or []),

copy.deepcopy(input),

context\_wrapper,

),

cls.\_run\_single\_turn(

agent=current\_agent,

all\_tools=all\_tools,

original\_input=original\_input,

generated\_items=generated\_items,

hooks=hooks,

context\_wrapper=context\_wrapper,

run\_config=run\_config,

should\_run\_agent\_start\_hooks=should\_run\_agent\_start\_hooks,

tool\_use\_tracker=tool\_use\_tracker,

),

)

else:

turn\_result = await cls.\_run\_single\_turn(

agent=current\_agent,

all\_tools=all\_tools,

original\_input=original\_input,

generated\_items=generated\_items,

hooks=hooks,

context\_wrapper=context\_wrapper,

run\_config=run\_config,

should\_run\_agent\_start\_hooks=should\_run\_agent\_start\_hooks,

tool\_use\_tracker=tool\_use\_tracker,

)

should\_run\_agent\_start\_hooks = False

model\_responses.append(turn\_result.model\_response)

original\_input = turn\_result.original\_input

generated\_items = turn\_result.generated\_items

if isinstance(turn\_result.next\_step, NextStepFinalOutput):

output\_guardrail\_results = await cls.\_run\_output\_guardrails(

current\_agent.output\_guardrails + (run\_config.output\_guardrails or []),

current\_agent,

turn\_result.next\_step.output,

context\_wrapper,

)

return RunResult(

input=original\_input,

new\_items=generated\_items,

raw\_responses=model\_responses,

final\_output=turn\_result.next\_step.output,

\_last\_agent=current\_agent,

input\_guardrail\_results=input\_guardrail\_results,

output\_guardrail\_results=output\_guardrail\_results,

)

elif isinstance(turn\_result.next\_step, NextStepHandoff):

current\_agent = cast(Agent[TContext], turn\_result.next\_step.new\_agent)

current\_span.finish(reset\_current=True)

current\_span = None

should\_run\_agent\_start\_hooks = True

elif isinstance(turn\_result.next\_step, NextStepRunAgain):

pass

else:

raise AgentsException(

f"Unknown next step type: {type(turn\_result.next\_step)}"

)

finally:

if current\_span:

current\_span.finish(reset\_current=True)

@classmethod

def run\_sync(

cls,

starting\_agent: Agent[TContext],

input: str | list[TResponseInputItem],

\*,

context: TContext | None = None,

max\_turns: int = DEFAULT\_MAX\_TURNS,

hooks: RunHooks[TContext] | None = None,

run\_config: RunConfig | None = None,

) -> RunResult:

"""Run a workflow synchronously, starting at the given agent. Note that this just wraps the

`run` method, so it will not work if there's already an event loop (e.g. inside an async

function, or in a Jupyter notebook or async context like FastAPI). For those cases, use

the `run` method instead.

The agent will run in a loop until a final output is generated. The loop runs like so:

1. The agent is invoked with the given input.

2. If there is a final output (i.e. the agent produces something of type

`agent.output\_type`, the loop terminates.

3. If there's a handoff, we run the loop again, with the new agent.

4. Else, we run tool calls (if any), and re-run the loop.

In two cases, the agent may raise an exception:

1. If the max\_turns is exceeded, a MaxTurnsExceeded exception is raised.

2. If a guardrail tripwire is triggered, a GuardrailTripwireTriggered exception is raised.

Note that only the first agent's input guardrails are run.

Args:

starting\_agent: The starting agent to run.

input: The initial input to the agent. You can pass a single string for a user message,

or a list of input items.

context: The context to run the agent with.

max\_turns: The maximum number of turns to run the agent for. A turn is defined as one

AI invocation (including any tool calls that might occur).

hooks: An object that receives callbacks on various lifecycle events.

run\_config: Global settings for the entire agent run.

Returns:

A run result containing all the inputs, guardrail results and the output of the last

agent. Agents may perform handoffs, so we don't know the specific type of the output.

"""

return asyncio.get\_event\_loop().run\_until\_complete(

cls.run(

starting\_agent,

input,

context=context,

max\_turns=max\_turns,

hooks=hooks,

run\_config=run\_config,

)

)

@classmethod

def run\_streamed(

cls,

starting\_agent: Agent[TContext],

input: str | list[TResponseInputItem],

context: TContext | None = None,

max\_turns: int = DEFAULT\_MAX\_TURNS,

hooks: RunHooks[TContext] | None = None,

run\_config: RunConfig | None = None,

) -> RunResultStreaming:

"""Run a workflow starting at the given agent in streaming mode. The returned result object

contains a method you can use to stream semantic events as they are generated.

The agent will run in a loop until a final output is generated. The loop runs like so:

1. The agent is invoked with the given input.

2. If there is a final output (i.e. the agent produces something of type

`agent.output\_type`, the loop terminates.

3. If there's a handoff, we run the loop again, with the new agent.

4. Else, we run tool calls (if any), and re-run the loop.

In two cases, the agent may raise an exception:

1. If the max\_turns is exceeded, a MaxTurnsExceeded exception is raised.

2. If a guardrail tripwire is triggered, a GuardrailTripwireTriggered exception is raised.

Note that only the first agent's input guardrails are run.

Args:

starting\_agent: The starting agent to run.

input: The initial input to the agent. You can pass a single string for a user message,

or a list of input items.

context: The context to run the agent with.

max\_turns: The maximum number of turns to run the agent for. A turn is defined as one

AI invocation (including any tool calls that might occur).

hooks: An object that receives callbacks on various lifecycle events.

run\_config: Global settings for the entire agent run.

Returns:

A result object that contains data about the run, as well as a method to stream events.

"""

if hooks is None:

hooks = RunHooks[Any]()

if run\_config is None:

run\_config = RunConfig()

# If there's already a trace, we don't create a new one. In addition, we can't end the

# trace here, because the actual work is done in `stream\_events` and this method ends

# before that.

new\_trace = (

None

if get\_current\_trace()

else trace(

workflow\_name=run\_config.workflow\_name,

trace\_id=run\_config.trace\_id,

group\_id=run\_config.group\_id,

metadata=run\_config.trace\_metadata,

disabled=run\_config.tracing\_disabled,

)

)

# Need to start the trace here, because the current trace contextvar is captured at

# asyncio.create\_task time

if new\_trace:

new\_trace.start(mark\_as\_current=True)

output\_schema = cls.\_get\_output\_schema(starting\_agent)

context\_wrapper: RunContextWrapper[TContext] = RunContextWrapper(

context=context # type: ignore

)

streamed\_result = RunResultStreaming(

input=copy.deepcopy(input),

new\_items=[],

current\_agent=starting\_agent,

raw\_responses=[],

final\_output=None,

is\_complete=False,

current\_turn=0,

max\_turns=max\_turns,

input\_guardrail\_results=[],

output\_guardrail\_results=[],

\_current\_agent\_output\_schema=output\_schema,

\_trace=new\_trace,

)

# Kick off the actual agent loop in the background and return the streamed result object.

streamed\_result.\_run\_impl\_task = asyncio.create\_task(

cls.\_run\_streamed\_impl(

starting\_input=input,

streamed\_result=streamed\_result,

starting\_agent=starting\_agent,

max\_turns=max\_turns,

hooks=hooks,

context\_wrapper=context\_wrapper,

run\_config=run\_config,

)

)

return streamed\_result

@classmethod

async def \_run\_input\_guardrails\_with\_queue(

cls,

agent: Agent[Any],

guardrails: list[InputGuardrail[TContext]],

input: str | list[TResponseInputItem],

context: RunContextWrapper[TContext],

streamed\_result: RunResultStreaming,

parent\_span: Span[Any],

):

queue = streamed\_result.\_input\_guardrail\_queue

# We'll run the guardrails and push them onto the queue as they complete

guardrail\_tasks = [

asyncio.create\_task(

RunImpl.run\_single\_input\_guardrail(agent, guardrail, input, context)

)

for guardrail in guardrails

]

guardrail\_results = []

try:

for done in asyncio.as\_completed(guardrail\_tasks):

result = await done

if result.output.tripwire\_triggered:

\_error\_tracing.attach\_error\_to\_span(

parent\_span,

SpanError(

message="Guardrail tripwire triggered",

data={

"guardrail": result.guardrail.get\_name(),

"type": "input\_guardrail",

},

),

)

queue.put\_nowait(result)

guardrail\_results.append(result)

except Exception:

for t in guardrail\_tasks:

t.cancel()

raise

streamed\_result.input\_guardrail\_results = guardrail\_results

@classmethod

async def \_run\_streamed\_impl(

cls,

starting\_input: str | list[TResponseInputItem],

streamed\_result: RunResultStreaming,

starting\_agent: Agent[TContext],

max\_turns: int,

hooks: RunHooks[TContext],

context\_wrapper: RunContextWrapper[TContext],

run\_config: RunConfig,

):

current\_span: Span[AgentSpanData] | None = None

current\_agent = starting\_agent

current\_turn = 0

should\_run\_agent\_start\_hooks = True

tool\_use\_tracker = AgentToolUseTracker()

streamed\_result.\_event\_queue.put\_nowait(AgentUpdatedStreamEvent(new\_agent=current\_agent))

try:

while True:

if streamed\_result.is\_complete:

break

# Start an agent span if we don't have one. This span is ended if the current

# agent changes, or if the agent loop ends.

if current\_span is None:

handoff\_names = [h.agent\_name for h in cls.\_get\_handoffs(current\_agent)]

if output\_schema := cls.\_get\_output\_schema(current\_agent):

output\_type\_name = output\_schema.output\_type\_name()

else:

output\_type\_name = "str"

current\_span = agent\_span(

name=current\_agent.name,

handoffs=handoff\_names,

output\_type=output\_type\_name,

)

current\_span.start(mark\_as\_current=True)

all\_tools = await cls.\_get\_all\_tools(current\_agent)

tool\_names = [t.name for t in all\_tools]

current\_span.span\_data.tools = tool\_names

current\_turn += 1

streamed\_result.current\_turn = current\_turn

if current\_turn > max\_turns:

\_error\_tracing.attach\_error\_to\_span(

current\_span,

SpanError(

message="Max turns exceeded",

data={"max\_turns": max\_turns},

),

)

streamed\_result.\_event\_queue.put\_nowait(QueueCompleteSentinel())

break

if current\_turn == 1:

# Run the input guardrails in the background and put the results on the queue

streamed\_result.\_input\_guardrails\_task = asyncio.create\_task(

cls.\_run\_input\_guardrails\_with\_queue(

starting\_agent,

starting\_agent.input\_guardrails + (run\_config.input\_guardrails or []),

copy.deepcopy(ItemHelpers.input\_to\_new\_input\_list(starting\_input)),

context\_wrapper,

streamed\_result,

current\_span,

)

)

try:

turn\_result = await cls.\_run\_single\_turn\_streamed(

streamed\_result,

current\_agent,

hooks,

context\_wrapper,

run\_config,

should\_run\_agent\_start\_hooks,

tool\_use\_tracker,

all\_tools,

)

should\_run\_agent\_start\_hooks = False

streamed\_result.raw\_responses = streamed\_result.raw\_responses + [

turn\_result.model\_response

]

streamed\_result.input = turn\_result.original\_input

streamed\_result.new\_items = turn\_result.generated\_items

if isinstance(turn\_result.next\_step, NextStepHandoff):

current\_agent = turn\_result.next\_step.new\_agent

current\_span.finish(reset\_current=True)

current\_span = None

should\_run\_agent\_start\_hooks = True

streamed\_result.\_event\_queue.put\_nowait(

AgentUpdatedStreamEvent(new\_agent=current\_agent)

)

elif isinstance(turn\_result.next\_step, NextStepFinalOutput):

streamed\_result.\_output\_guardrails\_task = asyncio.create\_task(

cls.\_run\_output\_guardrails(

current\_agent.output\_guardrails

+ (run\_config.output\_guardrails or []),

current\_agent,

turn\_result.next\_step.output,

context\_wrapper,

)

)

try:

output\_guardrail\_results = await streamed\_result.\_output\_guardrails\_task

except Exception:

# Exceptions will be checked in the stream\_events loop

output\_guardrail\_results = []

streamed\_result.output\_guardrail\_results = output\_guardrail\_results

streamed\_result.final\_output = turn\_result.next\_step.output

streamed\_result.is\_complete = True

streamed\_result.\_event\_queue.put\_nowait(QueueCompleteSentinel())

elif isinstance(turn\_result.next\_step, NextStepRunAgain):

pass

except Exception as e:

if current\_span:

\_error\_tracing.attach\_error\_to\_span(

current\_span,

SpanError(

message="Error in agent run",

data={"error": str(e)},

),

)

streamed\_result.is\_complete = True

streamed\_result.\_event\_queue.put\_nowait(QueueCompleteSentinel())

raise

streamed\_result.is\_complete = True

finally:

if current\_span:

current\_span.finish(reset\_current=True)

@classmethod

async def \_run\_single\_turn\_streamed(

cls,

streamed\_result: RunResultStreaming,

agent: Agent[TContext],

hooks: RunHooks[TContext],

context\_wrapper: RunContextWrapper[TContext],

run\_config: RunConfig,

should\_run\_agent\_start\_hooks: bool,

tool\_use\_tracker: AgentToolUseTracker,

all\_tools: list[Tool],

) -> SingleStepResult:

if should\_run\_agent\_start\_hooks:

await asyncio.gather(

hooks.on\_agent\_start(context\_wrapper, agent),

(

agent.hooks.on\_start(context\_wrapper, agent)

if agent.hooks

else \_coro.noop\_coroutine()

),

)

output\_schema = cls.\_get\_output\_schema(agent)

streamed\_result.current\_agent = agent

streamed\_result.\_current\_agent\_output\_schema = output\_schema

system\_prompt = await agent.get\_system\_prompt(context\_wrapper)

handoffs = cls.\_get\_handoffs(agent)

model = cls.\_get\_model(agent, run\_config)

model\_settings = agent.model\_settings.resolve(run\_config.model\_settings)

model\_settings = RunImpl.maybe\_reset\_tool\_choice(agent, tool\_use\_tracker, model\_settings)

final\_response: ModelResponse | None = None

input = ItemHelpers.input\_to\_new\_input\_list(streamed\_result.input)

input.extend([item.to\_input\_item() for item in streamed\_result.new\_items])

# 1. Stream the output events

async for event in model.stream\_response(

system\_prompt,

input,

model\_settings,

all\_tools,

output\_schema,

handoffs,

get\_model\_tracing\_impl(

run\_config.tracing\_disabled, run\_config.trace\_include\_sensitive\_data

),

):

if isinstance(event, ResponseCompletedEvent):

usage = (

Usage(

requests=1,

input\_tokens=event.response.usage.input\_tokens,

output\_tokens=event.response.usage.output\_tokens,

total\_tokens=event.response.usage.total\_tokens,

)

if event.response.usage

else Usage()

)

final\_response = ModelResponse(

output=event.response.output,

usage=usage,

referenceable\_id=event.response.id,

)

streamed\_result.\_event\_queue.put\_nowait(RawResponsesStreamEvent(data=event))

# 2. At this point, the streaming is complete for this turn of the agent loop.

if not final\_response:

raise ModelBehaviorError("Model did not produce a final response!")

# 3. Now, we can process the turn as we do in the non-streaming case

single\_step\_result = await cls.\_get\_single\_step\_result\_from\_response(

agent=agent,

original\_input=streamed\_result.input,

pre\_step\_items=streamed\_result.new\_items,

new\_response=final\_response,

output\_schema=output\_schema,

all\_tools=all\_tools,

handoffs=handoffs,

hooks=hooks,

context\_wrapper=context\_wrapper,

run\_config=run\_config,

tool\_use\_tracker=tool\_use\_tracker,

)

RunImpl.stream\_step\_result\_to\_queue(single\_step\_result, streamed\_result.\_event\_queue)

return single\_step\_result

@classmethod

async def \_run\_single\_turn(

cls,

\*,

agent: Agent[TContext],

all\_tools: list[Tool],

original\_input: str | list[TResponseInputItem],

generated\_items: list[RunItem],

hooks: RunHooks[TContext],

context\_wrapper: RunContextWrapper[TContext],

run\_config: RunConfig,

should\_run\_agent\_start\_hooks: bool,

tool\_use\_tracker: AgentToolUseTracker,

) -> SingleStepResult:

# Ensure we run the hooks before anything else

if should\_run\_agent\_start\_hooks:

await asyncio.gather(

hooks.on\_agent\_start(context\_wrapper, agent),

(

agent.hooks.on\_start(context\_wrapper, agent)

if agent.hooks

else \_coro.noop\_coroutine()

),

)

system\_prompt = await agent.get\_system\_prompt(context\_wrapper)

output\_schema = cls.\_get\_output\_schema(agent)

handoffs = cls.\_get\_handoffs(agent)

input = ItemHelpers.input\_to\_new\_input\_list(original\_input)

input.extend([generated\_item.to\_input\_item() for generated\_item in generated\_items])

new\_response = await cls.\_get\_new\_response(

agent,

system\_prompt,

input,

output\_schema,

all\_tools,

handoffs,

context\_wrapper,

run\_config,

tool\_use\_tracker,

)

return await cls.\_get\_single\_step\_result\_from\_response(

agent=agent,

original\_input=original\_input,

pre\_step\_items=generated\_items,

new\_response=new\_response,

output\_schema=output\_schema,

all\_tools=all\_tools,

handoffs=handoffs,

hooks=hooks,

context\_wrapper=context\_wrapper,

run\_config=run\_config,

tool\_use\_tracker=tool\_use\_tracker,

)

@classmethod

async def \_get\_single\_step\_result\_from\_response(

cls,

\*,

agent: Agent[TContext],

all\_tools: list[Tool],

original\_input: str | list[TResponseInputItem],

pre\_step\_items: list[RunItem],

new\_response: ModelResponse,

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

hooks: RunHooks[TContext],

context\_wrapper: RunContextWrapper[TContext],

run\_config: RunConfig,

tool\_use\_tracker: AgentToolUseTracker,

) -> SingleStepResult:

processed\_response = RunImpl.process\_model\_response(

agent=agent,

all\_tools=all\_tools,

response=new\_response,

output\_schema=output\_schema,

handoffs=handoffs,

)

tool\_use\_tracker.add\_tool\_use(agent, processed\_response.tools\_used)

return await RunImpl.execute\_tools\_and\_side\_effects(

agent=agent,

original\_input=original\_input,

pre\_step\_items=pre\_step\_items,

new\_response=new\_response,

processed\_response=processed\_response,

output\_schema=output\_schema,

hooks=hooks,

context\_wrapper=context\_wrapper,

run\_config=run\_config,

)

@classmethod

async def \_run\_input\_guardrails(

cls,

agent: Agent[Any],

guardrails: list[InputGuardrail[TContext]],

input: str | list[TResponseInputItem],

context: RunContextWrapper[TContext],

) -> list[InputGuardrailResult]:

if not guardrails:

return []

guardrail\_tasks = [

asyncio.create\_task(

RunImpl.run\_single\_input\_guardrail(agent, guardrail, input, context)

)

for guardrail in guardrails

]

guardrail\_results = []

for done in asyncio.as\_completed(guardrail\_tasks):

result = await done

if result.output.tripwire\_triggered:

# Cancel all guardrail tasks if a tripwire is triggered.

for t in guardrail\_tasks:

t.cancel()

\_error\_tracing.attach\_error\_to\_current\_span(

SpanError(

message="Guardrail tripwire triggered",

data={"guardrail": result.guardrail.get\_name()},

)

)

raise InputGuardrailTripwireTriggered(result)

else:

guardrail\_results.append(result)

return guardrail\_results

@classmethod

async def \_run\_output\_guardrails(

cls,

guardrails: list[OutputGuardrail[TContext]],

agent: Agent[TContext],

agent\_output: Any,

context: RunContextWrapper[TContext],

) -> list[OutputGuardrailResult]:

if not guardrails:

return []

guardrail\_tasks = [

asyncio.create\_task(

RunImpl.run\_single\_output\_guardrail(guardrail, agent, agent\_output, context)

)

for guardrail in guardrails

]

guardrail\_results = []

for done in asyncio.as\_completed(guardrail\_tasks):

result = await done

if result.output.tripwire\_triggered:

# Cancel all guardrail tasks if a tripwire is triggered.

for t in guardrail\_tasks:

t.cancel()

\_error\_tracing.attach\_error\_to\_current\_span(

SpanError(

message="Guardrail tripwire triggered",

data={"guardrail": result.guardrail.get\_name()},

)

)

raise OutputGuardrailTripwireTriggered(result)

else:

guardrail\_results.append(result)

return guardrail\_results

@classmethod

async def \_get\_new\_response(

cls,

agent: Agent[TContext],

system\_prompt: str | None,

input: list[TResponseInputItem],

output\_schema: AgentOutputSchema | None,

all\_tools: list[Tool],

handoffs: list[Handoff],

context\_wrapper: RunContextWrapper[TContext],

run\_config: RunConfig,

tool\_use\_tracker: AgentToolUseTracker,

) -> ModelResponse:

model = cls.\_get\_model(agent, run\_config)

model\_settings = agent.model\_settings.resolve(run\_config.model\_settings)

model\_settings = RunImpl.maybe\_reset\_tool\_choice(agent, tool\_use\_tracker, model\_settings)

new\_response = await model.get\_response(

system\_instructions=system\_prompt,

input=input,

model\_settings=model\_settings,

tools=all\_tools,

output\_schema=output\_schema,

handoffs=handoffs,

tracing=get\_model\_tracing\_impl(

run\_config.tracing\_disabled, run\_config.trace\_include\_sensitive\_data

),

)

context\_wrapper.usage.add(new\_response.usage)

return new\_response

@classmethod

def \_get\_output\_schema(cls, agent: Agent[Any]) -> AgentOutputSchema | None:

if agent.output\_type is None or agent.output\_type is str:

return None

return AgentOutputSchema(agent.output\_type)

@classmethod

def \_get\_handoffs(cls, agent: Agent[Any]) -> list[Handoff]:

handoffs = []

for handoff\_item in agent.handoffs:

if isinstance(handoff\_item, Handoff):

handoffs.append(handoff\_item)

elif isinstance(handoff\_item, Agent):

handoffs.append(handoff(handoff\_item))

return handoffs

@classmethod

async def \_get\_all\_tools(cls, agent: Agent[Any]) -> list[Tool]:

return await agent.get\_all\_tools()

@classmethod

def \_get\_model(cls, agent: Agent[Any], run\_config: RunConfig) -> Model:

if isinstance(run\_config.model, Model):

return run\_config.model

elif isinstance(run\_config.model, str):

return run\_config.model\_provider.get\_model(run\_config.model)

elif isinstance(agent.model, Model):

return agent.model

return run\_config.model\_provider.get\_model(agent.model)

run async classmethod

run(

starting\_agent: Agent[TContext],

input: str | list[TResponseInputItem],

\*,

context: TContext | None = None,

max\_turns: int = DEFAULT\_MAX\_TURNS,

hooks: RunHooks[TContext] | None = None,

run\_config: RunConfig | None = None,

) -> RunResult

Run a workflow starting at the given agent. The agent will run in a loop until a final output is generated. The loop runs like so: 1. The agent is invoked with the given input. 2. If there is a final output (i.e. the agent produces something of type agent.output\_type, the loop terminates. 3. If there's a handoff, we run the loop again, with the new agent. 4. Else, we run tool calls (if any), and re-run the loop.

In two cases, the agent may raise an exception: 1. If the max\_turns is exceeded, a MaxTurnsExceeded exception is raised. 2. If a guardrail tripwire is triggered, a GuardrailTripwireTriggered exception is raised.

Note that only the first agent's input guardrails are run.

Parameters:

Name Type Description Default

starting\_agent Agent[TContext] The starting agent to run. required

input str | list[TResponseInputItem] The initial input to the agent. You can pass a single string for a user message, or a list of input items. required

context TContext | None The context to run the agent with. None

max\_turns int The maximum number of turns to run the agent for. A turn is defined as one AI invocation (including any tool calls that might occur). DEFAULT\_MAX\_TURNS

hooks RunHooks[TContext] | None An object that receives callbacks on various lifecycle events. None

run\_config RunConfig | None Global settings for the entire agent run. None

Returns:

Type Description

RunResult A run result containing all the inputs, guardrail results and the output of the last

RunResult agent. Agents may perform handoffs, so we don't know the specific type of the output.

Source code in src/agents/run.py

@classmethod

async def run(

cls,

starting\_agent: Agent[TContext],

input: str | list[TResponseInputItem],

\*,

context: TContext | None = None,

max\_turns: int = DEFAULT\_MAX\_TURNS,

hooks: RunHooks[TContext] | None = None,

run\_config: RunConfig | None = None,

) -> RunResult:

"""Run a workflow starting at the given agent. The agent will run in a loop until a final

output is generated. The loop runs like so:

1. The agent is invoked with the given input.

2. If there is a final output (i.e. the agent produces something of type

`agent.output\_type`, the loop terminates.

3. If there's a handoff, we run the loop again, with the new agent.

4. Else, we run tool calls (if any), and re-run the loop.

In two cases, the agent may raise an exception:

1. If the max\_turns is exceeded, a MaxTurnsExceeded exception is raised.

2. If a guardrail tripwire is triggered, a GuardrailTripwireTriggered exception is raised.

Note that only the first agent's input guardrails are run.

Args:

starting\_agent: The starting agent to run.

input: The initial input to the agent. You can pass a single string for a user message,

or a list of input items.

context: The context to run the agent with.

max\_turns: The maximum number of turns to run the agent for. A turn is defined as one

AI invocation (including any tool calls that might occur).

hooks: An object that receives callbacks on various lifecycle events.

run\_config: Global settings for the entire agent run.

Returns:

A run result containing all the inputs, guardrail results and the output of the last

agent. Agents may perform handoffs, so we don't know the specific type of the output.

"""

if hooks is None:

hooks = RunHooks[Any]()

if run\_config is None:

run\_config = RunConfig()

tool\_use\_tracker = AgentToolUseTracker()

with TraceCtxManager(

workflow\_name=run\_config.workflow\_name,

trace\_id=run\_config.trace\_id,

group\_id=run\_config.group\_id,

metadata=run\_config.trace\_metadata,

disabled=run\_config.tracing\_disabled,

):

current\_turn = 0

original\_input: str | list[TResponseInputItem] = copy.deepcopy(input)

generated\_items: list[RunItem] = []

model\_responses: list[ModelResponse] = []

context\_wrapper: RunContextWrapper[TContext] = RunContextWrapper(

context=context, # type: ignore

)

input\_guardrail\_results: list[InputGuardrailResult] = []

current\_span: Span[AgentSpanData] | None = None

current\_agent = starting\_agent

should\_run\_agent\_start\_hooks = True

try:

while True:

# Start an agent span if we don't have one. This span is ended if the current

# agent changes, or if the agent loop ends.

if current\_span is None:

handoff\_names = [h.agent\_name for h in cls.\_get\_handoffs(current\_agent)]

if output\_schema := cls.\_get\_output\_schema(current\_agent):

output\_type\_name = output\_schema.output\_type\_name()

else:

output\_type\_name = "str"

current\_span = agent\_span(

name=current\_agent.name,

handoffs=handoff\_names,

output\_type=output\_type\_name,

)

current\_span.start(mark\_as\_current=True)

all\_tools = await cls.\_get\_all\_tools(current\_agent)

current\_span.span\_data.tools = [t.name for t in all\_tools]

current\_turn += 1

if current\_turn > max\_turns:

\_error\_tracing.attach\_error\_to\_span(

current\_span,

SpanError(

message="Max turns exceeded",

data={"max\_turns": max\_turns},

),

)

raise MaxTurnsExceeded(f"Max turns ({max\_turns}) exceeded")

logger.debug(

f"Running agent {current\_agent.name} (turn {current\_turn})",

)

if current\_turn == 1:

input\_guardrail\_results, turn\_result = await asyncio.gather(

cls.\_run\_input\_guardrails(

starting\_agent,

starting\_agent.input\_guardrails

+ (run\_config.input\_guardrails or []),

copy.deepcopy(input),

context\_wrapper,

),

cls.\_run\_single\_turn(

agent=current\_agent,

all\_tools=all\_tools,

original\_input=original\_input,

generated\_items=generated\_items,

hooks=hooks,

context\_wrapper=context\_wrapper,

run\_config=run\_config,

should\_run\_agent\_start\_hooks=should\_run\_agent\_start\_hooks,

tool\_use\_tracker=tool\_use\_tracker,

),

)

else:

turn\_result = await cls.\_run\_single\_turn(

agent=current\_agent,

all\_tools=all\_tools,

original\_input=original\_input,

generated\_items=generated\_items,

hooks=hooks,

context\_wrapper=context\_wrapper,

run\_config=run\_config,

should\_run\_agent\_start\_hooks=should\_run\_agent\_start\_hooks,

tool\_use\_tracker=tool\_use\_tracker,

)

should\_run\_agent\_start\_hooks = False

model\_responses.append(turn\_result.model\_response)

original\_input = turn\_result.original\_input

generated\_items = turn\_result.generated\_items

if isinstance(turn\_result.next\_step, NextStepFinalOutput):

output\_guardrail\_results = await cls.\_run\_output\_guardrails(

current\_agent.output\_guardrails + (run\_config.output\_guardrails or []),

current\_agent,

turn\_result.next\_step.output,

context\_wrapper,

)

return RunResult(

input=original\_input,

new\_items=generated\_items,

raw\_responses=model\_responses,

final\_output=turn\_result.next\_step.output,

\_last\_agent=current\_agent,

input\_guardrail\_results=input\_guardrail\_results,

output\_guardrail\_results=output\_guardrail\_results,

)

elif isinstance(turn\_result.next\_step, NextStepHandoff):

current\_agent = cast(Agent[TContext], turn\_result.next\_step.new\_agent)

current\_span.finish(reset\_current=True)

current\_span = None

should\_run\_agent\_start\_hooks = True

elif isinstance(turn\_result.next\_step, NextStepRunAgain):

pass

else:

raise AgentsException(

f"Unknown next step type: {type(turn\_result.next\_step)}"

)

finally:

if current\_span:

current\_span.finish(reset\_current=True)

run\_sync classmethod

run\_sync(

starting\_agent: Agent[TContext],

input: str | list[TResponseInputItem],

\*,

context: TContext | None = None,

max\_turns: int = DEFAULT\_MAX\_TURNS,

hooks: RunHooks[TContext] | None = None,

run\_config: RunConfig | None = None,

) -> RunResult

Run a workflow synchronously, starting at the given agent. Note that this just wraps the run method, so it will not work if there's already an event loop (e.g. inside an async function, or in a Jupyter notebook or async context like FastAPI). For those cases, use the run method instead.

The agent will run in a loop until a final output is generated. The loop runs like so: 1. The agent is invoked with the given input. 2. If there is a final output (i.e. the agent produces something of type agent.output\_type, the loop terminates. 3. If there's a handoff, we run the loop again, with the new agent. 4. Else, we run tool calls (if any), and re-run the loop.

In two cases, the agent may raise an exception: 1. If the max\_turns is exceeded, a MaxTurnsExceeded exception is raised. 2. If a guardrail tripwire is triggered, a GuardrailTripwireTriggered exception is raised.

Note that only the first agent's input guardrails are run.

Parameters:

Name Type Description Default

starting\_agent Agent[TContext] The starting agent to run. required

input str | list[TResponseInputItem] The initial input to the agent. You can pass a single string for a user message, or a list of input items. required

context TContext | None The context to run the agent with. None

max\_turns int The maximum number of turns to run the agent for. A turn is defined as one AI invocation (including any tool calls that might occur). DEFAULT\_MAX\_TURNS

hooks RunHooks[TContext] | None An object that receives callbacks on various lifecycle events. None

run\_config RunConfig | None Global settings for the entire agent run. None

Returns:

Type Description

RunResult A run result containing all the inputs, guardrail results and the output of the last

RunResult agent. Agents may perform handoffs, so we don't know the specific type of the output.

Source code in src/agents/run.py

@classmethod

def run\_sync(

cls,

starting\_agent: Agent[TContext],

input: str | list[TResponseInputItem],

\*,

context: TContext | None = None,

max\_turns: int = DEFAULT\_MAX\_TURNS,

hooks: RunHooks[TContext] | None = None,

run\_config: RunConfig | None = None,

) -> RunResult:

"""Run a workflow synchronously, starting at the given agent. Note that this just wraps the

`run` method, so it will not work if there's already an event loop (e.g. inside an async

function, or in a Jupyter notebook or async context like FastAPI). For those cases, use

the `run` method instead.

The agent will run in a loop until a final output is generated. The loop runs like so:

1. The agent is invoked with the given input.

2. If there is a final output (i.e. the agent produces something of type

`agent.output\_type`, the loop terminates.

3. If there's a handoff, we run the loop again, with the new agent.

4. Else, we run tool calls (if any), and re-run the loop.

In two cases, the agent may raise an exception:

1. If the max\_turns is exceeded, a MaxTurnsExceeded exception is raised.

2. If a guardrail tripwire is triggered, a GuardrailTripwireTriggered exception is raised.

Note that only the first agent's input guardrails are run.

Args:

starting\_agent: The starting agent to run.

input: The initial input to the agent. You can pass a single string for a user message,

or a list of input items.

context: The context to run the agent with.

max\_turns: The maximum number of turns to run the agent for. A turn is defined as one

AI invocation (including any tool calls that might occur).

hooks: An object that receives callbacks on various lifecycle events.

run\_config: Global settings for the entire agent run.

Returns:

A run result containing all the inputs, guardrail results and the output of the last

agent. Agents may perform handoffs, so we don't know the specific type of the output.

"""

return asyncio.get\_event\_loop().run\_until\_complete(

cls.run(

starting\_agent,

input,

context=context,

max\_turns=max\_turns,

hooks=hooks,

run\_config=run\_config,

)

)

run\_streamed classmethod

run\_streamed(

starting\_agent: Agent[TContext],

input: str | list[TResponseInputItem],

context: TContext | None = None,

max\_turns: int = DEFAULT\_MAX\_TURNS,

hooks: RunHooks[TContext] | None = None,

run\_config: RunConfig | None = None,

) -> RunResultStreaming

Run a workflow starting at the given agent in streaming mode. The returned result object contains a method you can use to stream semantic events as they are generated.

The agent will run in a loop until a final output is generated. The loop runs like so: 1. The agent is invoked with the given input. 2. If there is a final output (i.e. the agent produces something of type agent.output\_type, the loop terminates. 3. If there's a handoff, we run the loop again, with the new agent. 4. Else, we run tool calls (if any), and re-run the loop.

In two cases, the agent may raise an exception: 1. If the max\_turns is exceeded, a MaxTurnsExceeded exception is raised. 2. If a guardrail tripwire is triggered, a GuardrailTripwireTriggered exception is raised.

Note that only the first agent's input guardrails are run.

Parameters:

Name Type Description Default

starting\_agent Agent[TContext] The starting agent to run. required

input str | list[TResponseInputItem] The initial input to the agent. You can pass a single string for a user message, or a list of input items. required

context TContext | None The context to run the agent with. None

max\_turns int The maximum number of turns to run the agent for. A turn is defined as one AI invocation (including any tool calls that might occur). DEFAULT\_MAX\_TURNS

hooks RunHooks[TContext] | None An object that receives callbacks on various lifecycle events. None

run\_config RunConfig | None Global settings for the entire agent run. None

Returns:

Type Description

RunResultStreaming A result object that contains data about the run, as well as a method to stream events.

Source code in src/agents/run.py

@classmethod

def run\_streamed(

cls,

starting\_agent: Agent[TContext],

input: str | list[TResponseInputItem],

context: TContext | None = None,

max\_turns: int = DEFAULT\_MAX\_TURNS,

hooks: RunHooks[TContext] | None = None,

run\_config: RunConfig | None = None,

) -> RunResultStreaming:

"""Run a workflow starting at the given agent in streaming mode. The returned result object

contains a method you can use to stream semantic events as they are generated.

The agent will run in a loop until a final output is generated. The loop runs like so:

1. The agent is invoked with the given input.

2. If there is a final output (i.e. the agent produces something of type

`agent.output\_type`, the loop terminates.

3. If there's a handoff, we run the loop again, with the new agent.

4. Else, we run tool calls (if any), and re-run the loop.

In two cases, the agent may raise an exception:

1. If the max\_turns is exceeded, a MaxTurnsExceeded exception is raised.

2. If a guardrail tripwire is triggered, a GuardrailTripwireTriggered exception is raised.

Note that only the first agent's input guardrails are run.

Args:

starting\_agent: The starting agent to run.

input: The initial input to the agent. You can pass a single string for a user message,

or a list of input items.

context: The context to run the agent with.

max\_turns: The maximum number of turns to run the agent for. A turn is defined as one

AI invocation (including any tool calls that might occur).

hooks: An object that receives callbacks on various lifecycle events.

run\_config: Global settings for the entire agent run.

Returns:

A result object that contains data about the run, as well as a method to stream events.

"""

if hooks is None:

hooks = RunHooks[Any]()

if run\_config is None:

run\_config = RunConfig()

# If there's already a trace, we don't create a new one. In addition, we can't end the

# trace here, because the actual work is done in `stream\_events` and this method ends

# before that.

new\_trace = (

None

if get\_current\_trace()

else trace(

workflow\_name=run\_config.workflow\_name,

trace\_id=run\_config.trace\_id,

group\_id=run\_config.group\_id,

metadata=run\_config.trace\_metadata,

disabled=run\_config.tracing\_disabled,

)

)

# Need to start the trace here, because the current trace contextvar is captured at

# asyncio.create\_task time

if new\_trace:

new\_trace.start(mark\_as\_current=True)

output\_schema = cls.\_get\_output\_schema(starting\_agent)

context\_wrapper: RunContextWrapper[TContext] = RunContextWrapper(

context=context # type: ignore

)

streamed\_result = RunResultStreaming(

input=copy.deepcopy(input),

new\_items=[],

current\_agent=starting\_agent,

raw\_responses=[],

final\_output=None,

is\_complete=False,

current\_turn=0,

max\_turns=max\_turns,

input\_guardrail\_results=[],

output\_guardrail\_results=[],

\_current\_agent\_output\_schema=output\_schema,

\_trace=new\_trace,

)

# Kick off the actual agent loop in the background and return the streamed result object.

streamed\_result.\_run\_impl\_task = asyncio.create\_task(

cls.\_run\_streamed\_impl(

starting\_input=input,

streamed\_result=streamed\_result,

starting\_agent=starting\_agent,

max\_turns=max\_turns,

hooks=hooks,

context\_wrapper=context\_wrapper,

run\_config=run\_config,

)

)

return streamed\_result

RunConfig dataclass

Configures settings for the entire agent run.

Source code in src/agents/run.py

@dataclass

class RunConfig:

"""Configures settings for the entire agent run."""

model: str | Model | None = None

"""The model to use for the entire agent run. If set, will override the model set on every

agent. The model\_provider passed in below must be able to resolve this model name.

"""

model\_provider: ModelProvider = field(default\_factory=OpenAIProvider)

"""The model provider to use when looking up string model names. Defaults to OpenAI."""

model\_settings: ModelSettings | None = None

"""Configure global model settings. Any non-null values will override the agent-specific model

settings.

"""

handoff\_input\_filter: HandoffInputFilter | None = None

"""A global input filter to apply to all handoffs. If `Handoff.input\_filter` is set, then that

will take precedence. The input filter allows you to edit the inputs that are sent to the new

agent. See the documentation in `Handoff.input\_filter` for more details.

"""

input\_guardrails: list[InputGuardrail[Any]] | None = None

"""A list of input guardrails to run on the initial run input."""

output\_guardrails: list[OutputGuardrail[Any]] | None = None

"""A list of output guardrails to run on the final output of the run."""

tracing\_disabled: bool = False

"""Whether tracing is disabled for the agent run. If disabled, we will not trace the agent run.

"""

trace\_include\_sensitive\_data: bool = True

"""Whether we include potentially sensitive data (for example: inputs/outputs of tool calls or

LLM generations) in traces. If False, we'll still create spans for these events, but the

sensitive data will not be included.

"""

workflow\_name: str = "Agent workflow"

"""The name of the run, used for tracing. Should be a logical name for the run, like

"Code generation workflow" or "Customer support agent".

"""

trace\_id: str | None = None

"""A custom trace ID to use for tracing. If not provided, we will generate a new trace ID."""

group\_id: str | None = None

"""

A grouping identifier to use for tracing, to link multiple traces from the same conversation

or process. For example, you might use a chat thread ID.

"""

trace\_metadata: dict[str, Any] | None = None

"""

An optional dictionary of additional metadata to include with the trace.

"""

model class-attribute instance-attribute

model: str | Model | None = None

The model to use for the entire agent run. If set, will override the model set on every agent. The model\_provider passed in below must be able to resolve this model name.

model\_provider class-attribute instance-attribute

model\_provider: ModelProvider = field(

default\_factory=OpenAIProvider

)

The model provider to use when looking up string model names. Defaults to OpenAI.

model\_settings class-attribute instance-attribute

model\_settings: ModelSettings | None = None

Configure global model settings. Any non-null values will override the agent-specific model settings.

handoff\_input\_filter class-attribute instance-attribute

handoff\_input\_filter: HandoffInputFilter | None = None

A global input filter to apply to all handoffs. If Handoff.input\_filter is set, then that will take precedence. The input filter allows you to edit the inputs that are sent to the new agent. See the documentation in Handoff.input\_filter for more details.

input\_guardrails class-attribute instance-attribute

input\_guardrails: list[InputGuardrail[Any]] | None = None

A list of input guardrails to run on the initial run input.

output\_guardrails class-attribute instance-attribute

output\_guardrails: list[OutputGuardrail[Any]] | None = None

A list of output guardrails to run on the final output of the run.

tracing\_disabled class-attribute instance-attribute

tracing\_disabled: bool = False

Whether tracing is disabled for the agent run. If disabled, we will not trace the agent run.

trace\_include\_sensitive\_data class-attribute instance-attribute

trace\_include\_sensitive\_data: bool = True

Whether we include potentially sensitive data (for example: inputs/outputs of tool calls or LLM generations) in traces. If False, we'll still create spans for these events, but the sensitive data will not be included.

workflow\_name class-attribute instance-attribute

workflow\_name: str = 'Agent workflow'

The name of the run, used for tracing. Should be a logical name for the run, like "Code generation workflow" or "Customer support agent".

trace\_id class-attribute instance-attribute

trace\_id: str | None = None

A custom trace ID to use for tracing. If not provided, we will generate a new trace ID.

group\_id class-attribute instance-attribute

group\_id: str | None = None

A grouping identifier to use for tracing, to link multiple traces from the same conversation or process. For example, you might use a chat thread ID.

trace\_metadata class-attribute instance-attribute

trace\_metadata: dict[str, Any] | None = None

An optional dictionary of additional metadata to include with the trace.

Tools

Tool module-attribute

Tool = Union[

FunctionTool,

FileSearchTool,

WebSearchTool,

ComputerTool,

]

A tool that can be used in an agent.

FunctionToolResult dataclass

Source code in src/agents/tool.py

@dataclass

class FunctionToolResult:

tool: FunctionTool

"""The tool that was run."""

output: Any

"""The output of the tool."""

run\_item: RunItem

"""The run item that was produced as a result of the tool call."""

tool instance-attribute

tool: FunctionTool

The tool that was run.

output instance-attribute

output: Any

The output of the tool.

run\_item instance-attribute

run\_item: RunItem

The run item that was produced as a result of the tool call.

FunctionTool dataclass

A tool that wraps a function. In most cases, you should use the function\_tool helpers to create a FunctionTool, as they let you easily wrap a Python function.

Source code in src/agents/tool.py

@dataclass

class FunctionTool:

"""A tool that wraps a function. In most cases, you should use the `function\_tool` helpers to

create a FunctionTool, as they let you easily wrap a Python function.

"""

name: str

"""The name of the tool, as shown to the LLM. Generally the name of the function."""

description: str

"""A description of the tool, as shown to the LLM."""

params\_json\_schema: dict[str, Any]

"""The JSON schema for the tool's parameters."""

on\_invoke\_tool: Callable[[RunContextWrapper[Any], str], Awaitable[Any]]

"""A function that invokes the tool with the given context and parameters. The params passed

are:

1. The tool run context.

2. The arguments from the LLM, as a JSON string.

You must return a string representation of the tool output, or something we can call `str()` on.

In case of errors, you can either raise an Exception (which will cause the run to fail) or

return a string error message (which will be sent back to the LLM).

"""

strict\_json\_schema: bool = True

"""Whether the JSON schema is in strict mode. We \*\*strongly\*\* recommend setting this to True,

as it increases the likelihood of correct JSON input."""

name instance-attribute

name: str

The name of the tool, as shown to the LLM. Generally the name of the function.

description instance-attribute

description: str

A description of the tool, as shown to the LLM.

params\_json\_schema instance-attribute

params\_json\_schema: dict[str, Any]

The JSON schema for the tool's parameters.

on\_invoke\_tool instance-attribute

on\_invoke\_tool: Callable[

[RunContextWrapper[Any], str], Awaitable[Any]

]

A function that invokes the tool with the given context and parameters. The params passed are: 1. The tool run context. 2. The arguments from the LLM, as a JSON string.

You must return a string representation of the tool output, or something we can call str() on. In case of errors, you can either raise an Exception (which will cause the run to fail) or return a string error message (which will be sent back to the LLM).

strict\_json\_schema class-attribute instance-attribute

strict\_json\_schema: bool = True

Whether the JSON schema is in strict mode. We strongly recommend setting this to True, as it increases the likelihood of correct JSON input.

FileSearchTool dataclass

A hosted tool that lets the LLM search through a vector store. Currently only supported with OpenAI models, using the Responses API.

Source code in src/agents/tool.py

@dataclass

class FileSearchTool:

"""A hosted tool that lets the LLM search through a vector store. Currently only supported with

OpenAI models, using the Responses API.

"""

vector\_store\_ids: list[str]

"""The IDs of the vector stores to search."""

max\_num\_results: int | None = None

"""The maximum number of results to return."""

include\_search\_results: bool = False

"""Whether to include the search results in the output produced by the LLM."""

ranking\_options: RankingOptions | None = None

"""Ranking options for search."""

filters: Filters | None = None

"""A filter to apply based on file attributes."""

@property

def name(self):

return "file\_search"

vector\_store\_ids instance-attribute

vector\_store\_ids: list[str]

The IDs of the vector stores to search.

max\_num\_results class-attribute instance-attribute

max\_num\_results: int | None = None

The maximum number of results to return.

include\_search\_results class-attribute instance-attribute

include\_search\_results: bool = False

Whether to include the search results in the output produced by the LLM.

ranking\_options class-attribute instance-attribute

ranking\_options: RankingOptions | None = None

Ranking options for search.

filters class-attribute instance-attribute

filters: Filters | None = None

A filter to apply based on file attributes.

WebSearchTool dataclass

A hosted tool that lets the LLM search the web. Currently only supported with OpenAI models, using the Responses API.

Source code in src/agents/tool.py

@dataclass

class WebSearchTool:

"""A hosted tool that lets the LLM search the web. Currently only supported with OpenAI models,

using the Responses API.

"""

user\_location: UserLocation | None = None

"""Optional location for the search. Lets you customize results to be relevant to a location."""

search\_context\_size: Literal["low", "medium", "high"] = "medium"

"""The amount of context to use for the search."""

@property

def name(self):

return "web\_search\_preview"

user\_location class-attribute instance-attribute

user\_location: UserLocation | None = None

Optional location for the search. Lets you customize results to be relevant to a location.

search\_context\_size class-attribute instance-attribute

search\_context\_size: Literal["low", "medium", "high"] = (

"medium"

)

The amount of context to use for the search.

ComputerTool dataclass

A hosted tool that lets the LLM control a computer.

Source code in src/agents/tool.py

@dataclass

class ComputerTool:

"""A hosted tool that lets the LLM control a computer."""

computer: Computer | AsyncComputer

"""The computer implementation, which describes the environment and dimensions of the computer,

as well as implements the computer actions like click, screenshot, etc.

"""

@property

def name(self):

return "computer\_use\_preview"

computer instance-attribute

computer: Computer | AsyncComputer

The computer implementation, which describes the environment and dimensions of the computer, as well as implements the computer actions like click, screenshot, etc.

default\_tool\_error\_function

default\_tool\_error\_function(

ctx: RunContextWrapper[Any], error: Exception

) -> str

The default tool error function, which just returns a generic error message.

Source code in src/agents/tool.py

def default\_tool\_error\_function(ctx: RunContextWrapper[Any], error: Exception) -> str:

"""The default tool error function, which just returns a generic error message."""

return f"An error occurred while running the tool. Please try again. Error: {str(error)}"

function\_tool

function\_tool(

func: ToolFunction[...],

\*,

name\_override: str | None = None,

description\_override: str | None = None,

docstring\_style: DocstringStyle | None = None,

use\_docstring\_info: bool = True,

failure\_error\_function: ToolErrorFunction | None = None,

strict\_mode: bool = True,

) -> FunctionTool

function\_tool(

\*,

name\_override: str | None = None,

description\_override: str | None = None,

docstring\_style: DocstringStyle | None = None,

use\_docstring\_info: bool = True,

failure\_error\_function: ToolErrorFunction | None = None,

strict\_mode: bool = True,

) -> Callable[[ToolFunction[...]], FunctionTool]

function\_tool(

func: ToolFunction[...] | None = None,

\*,

name\_override: str | None = None,

description\_override: str | None = None,

docstring\_style: DocstringStyle | None = None,

use\_docstring\_info: bool = True,

failure\_error\_function: ToolErrorFunction

| None = default\_tool\_error\_function,

strict\_mode: bool = True,

) -> (

FunctionTool

| Callable[[ToolFunction[...]], FunctionTool]

)

Decorator to create a FunctionTool from a function. By default, we will: 1. Parse the function signature to create a JSON schema for the tool's parameters. 2. Use the function's docstring to populate the tool's description. 3. Use the function's docstring to populate argument descriptions. The docstring style is detected automatically, but you can override it.

If the function takes a RunContextWrapper as the first argument, it must match the context type of the agent that uses the tool.

Parameters:

Name Type Description Default

func ToolFunction[...] | None The function to wrap. None

name\_override str | None If provided, use this name for the tool instead of the function's name. None

description\_override str | None If provided, use this description for the tool instead of the function's docstring. None

docstring\_style DocstringStyle | None If provided, use this style for the tool's docstring. If not provided, we will attempt to auto-detect the style. None

use\_docstring\_info bool If True, use the function's docstring to populate the tool's description and argument descriptions. True

failure\_error\_function ToolErrorFunction | None If provided, use this function to generate an error message when the tool call fails. The error message is sent to the LLM. If you pass None, then no error message will be sent and instead an Exception will be raised. default\_tool\_error\_function

strict\_mode bool Whether to enable strict mode for the tool's JSON schema. We strongly recommend setting this to True, as it increases the likelihood of correct JSON input. If False, it allows non-strict JSON schemas. For example, if a parameter has a default value, it will be optional, additional properties are allowed, etc. See here for more: https://platform.openai.com/docs/guides/structured-outputs?api-mode=responses#supported-schemas True

Source code in src/agents/tool.py

def function\_tool(

func: ToolFunction[...] | None = None,

\*,

name\_override: str | None = None,

description\_override: str | None = None,

docstring\_style: DocstringStyle | None = None,

use\_docstring\_info: bool = True,

failure\_error\_function: ToolErrorFunction | None = default\_tool\_error\_function,

strict\_mode: bool = True,

) -> FunctionTool | Callable[[ToolFunction[...]], FunctionTool]:

"""

Decorator to create a FunctionTool from a function. By default, we will:

1. Parse the function signature to create a JSON schema for the tool's parameters.

2. Use the function's docstring to populate the tool's description.

3. Use the function's docstring to populate argument descriptions.

The docstring style is detected automatically, but you can override it.

If the function takes a `RunContextWrapper` as the first argument, it \*must\* match the

context type of the agent that uses the tool.

Args:

func: The function to wrap.

name\_override: If provided, use this name for the tool instead of the function's name.

description\_override: If provided, use this description for the tool instead of the

function's docstring.

docstring\_style: If provided, use this style for the tool's docstring. If not provided,

we will attempt to auto-detect the style.

use\_docstring\_info: If True, use the function's docstring to populate the tool's

description and argument descriptions.

failure\_error\_function: If provided, use this function to generate an error message when

the tool call fails. The error message is sent to the LLM. If you pass None, then no

error message will be sent and instead an Exception will be raised.

strict\_mode: Whether to enable strict mode for the tool's JSON schema. We \*strongly\*

recommend setting this to True, as it increases the likelihood of correct JSON input.

If False, it allows non-strict JSON schemas. For example, if a parameter has a default

value, it will be optional, additional properties are allowed, etc. See here for more:

https://platform.openai.com/docs/guides/structured-outputs?api-mode=responses#supported-schemas

"""

def \_create\_function\_tool(the\_func: ToolFunction[...]) -> FunctionTool:

schema = function\_schema(

func=the\_func,

name\_override=name\_override,

description\_override=description\_override,

docstring\_style=docstring\_style,

use\_docstring\_info=use\_docstring\_info,

strict\_json\_schema=strict\_mode,

)

async def \_on\_invoke\_tool\_impl(ctx: RunContextWrapper[Any], input: str) -> Any:

try:

json\_data: dict[str, Any] = json.loads(input) if input else {}

except Exception as e:

if \_debug.DONT\_LOG\_TOOL\_DATA:

logger.debug(f"Invalid JSON input for tool {schema.name}")

else:

logger.debug(f"Invalid JSON input for tool {schema.name}: {input}")

raise ModelBehaviorError(

f"Invalid JSON input for tool {schema.name}: {input}"

) from e

if \_debug.DONT\_LOG\_TOOL\_DATA:

logger.debug(f"Invoking tool {schema.name}")

else:

logger.debug(f"Invoking tool {schema.name} with input {input}")

try:

parsed = (

schema.params\_pydantic\_model(\*\*json\_data)

if json\_data

else schema.params\_pydantic\_model()

)

except ValidationError as e:

raise ModelBehaviorError(f"Invalid JSON input for tool {schema.name}: {e}") from e

args, kwargs\_dict = schema.to\_call\_args(parsed)

if not \_debug.DONT\_LOG\_TOOL\_DATA:

logger.debug(f"Tool call args: {args}, kwargs: {kwargs\_dict}")

if inspect.iscoroutinefunction(the\_func):

if schema.takes\_context:

result = await the\_func(ctx, \*args, \*\*kwargs\_dict)

else:

result = await the\_func(\*args, \*\*kwargs\_dict)

else:

if schema.takes\_context:

result = the\_func(ctx, \*args, \*\*kwargs\_dict)

else:

result = the\_func(\*args, \*\*kwargs\_dict)

if \_debug.DONT\_LOG\_TOOL\_DATA:

logger.debug(f"Tool {schema.name} completed.")

else:

logger.debug(f"Tool {schema.name} returned {result}")

return result

async def \_on\_invoke\_tool(ctx: RunContextWrapper[Any], input: str) -> Any:

try:

return await \_on\_invoke\_tool\_impl(ctx, input)

except Exception as e:

if failure\_error\_function is None:

raise

result = failure\_error\_function(ctx, e)

if inspect.isawaitable(result):

return await result

\_error\_tracing.attach\_error\_to\_current\_span(

SpanError(

message="Error running tool (non-fatal)",

data={

"tool\_name": schema.name,

"error": str(e),

},

)

)

return result

return FunctionTool(

name=schema.name,

description=schema.description or "",

params\_json\_schema=schema.params\_json\_schema,

on\_invoke\_tool=\_on\_invoke\_tool,

strict\_json\_schema=strict\_mode,

)

# If func is actually a callable, we were used as @function\_tool with no parentheses

if callable(func):

return \_create\_function\_tool(func)

# Otherwise, we were used as @function\_tool(...), so return a decorator

def decorator(real\_func: ToolFunction[...]) -> FunctionTool:

return \_create\_function\_tool(real\_func)

return decorator

Results

RunResultBase dataclass

Bases: ABC

Source code in src/agents/result.py

@dataclass

class RunResultBase(abc.ABC):

input: str | list[TResponseInputItem]

"""The original input items i.e. the items before run() was called. This may be a mutated

version of the input, if there are handoff input filters that mutate the input.

"""

new\_items: list[RunItem]

"""The new items generated during the agent run. These include things like new messages, tool

calls and their outputs, etc.

"""

raw\_responses: list[ModelResponse]

"""The raw LLM responses generated by the model during the agent run."""

final\_output: Any

"""The output of the last agent."""

input\_guardrail\_results: list[InputGuardrailResult]

"""Guardrail results for the input messages."""

output\_guardrail\_results: list[OutputGuardrailResult]

"""Guardrail results for the final output of the agent."""

@property

@abc.abstractmethod

def last\_agent(self) -> Agent[Any]:

"""The last agent that was run."""

def final\_output\_as(self, cls: type[T], raise\_if\_incorrect\_type: bool = False) -> T:

"""A convenience method to cast the final output to a specific type. By default, the cast

is only for the typechecker. If you set `raise\_if\_incorrect\_type` to True, we'll raise a

TypeError if the final output is not of the given type.

Args:

cls: The type to cast the final output to.

raise\_if\_incorrect\_type: If True, we'll raise a TypeError if the final output is not of

the given type.

Returns:

The final output casted to the given type.

"""

if raise\_if\_incorrect\_type and not isinstance(self.final\_output, cls):

raise TypeError(f"Final output is not of type {cls.\_\_name\_\_}")

return cast(T, self.final\_output)

def to\_input\_list(self) -> list[TResponseInputItem]:

"""Creates a new input list, merging the original input with all the new items generated."""

original\_items: list[TResponseInputItem] = ItemHelpers.input\_to\_new\_input\_list(self.input)

new\_items = [item.to\_input\_item() for item in self.new\_items]

return original\_items + new\_items

input instance-attribute

input: str | list[TResponseInputItem]

The original input items i.e. the items before run() was called. This may be a mutated version of the input, if there are handoff input filters that mutate the input.

new\_items instance-attribute

new\_items: list[RunItem]

The new items generated during the agent run. These include things like new messages, tool calls and their outputs, etc.

raw\_responses instance-attribute

raw\_responses: list[ModelResponse]

The raw LLM responses generated by the model during the agent run.

final\_output instance-attribute

final\_output: Any

The output of the last agent.

input\_guardrail\_results instance-attribute

input\_guardrail\_results: list[InputGuardrailResult]

Guardrail results for the input messages.

output\_guardrail\_results instance-attribute

output\_guardrail\_results: list[OutputGuardrailResult]

Guardrail results for the final output of the agent.

last\_agent abstractmethod property

last\_agent: Agent[Any]

The last agent that was run.

final\_output\_as

final\_output\_as(

cls: type[T], raise\_if\_incorrect\_type: bool = False

) -> T

A convenience method to cast the final output to a specific type. By default, the cast is only for the typechecker. If you set raise\_if\_incorrect\_type to True, we'll raise a TypeError if the final output is not of the given type.

Parameters:

Name Type Description Default

cls type[T] The type to cast the final output to. required

raise\_if\_incorrect\_type bool If True, we'll raise a TypeError if the final output is not of the given type. False

Returns:

Type Description

T The final output casted to the given type.

Source code in src/agents/result.py

def final\_output\_as(self, cls: type[T], raise\_if\_incorrect\_type: bool = False) -> T:

"""A convenience method to cast the final output to a specific type. By default, the cast

is only for the typechecker. If you set `raise\_if\_incorrect\_type` to True, we'll raise a

TypeError if the final output is not of the given type.

Args:

cls: The type to cast the final output to.

raise\_if\_incorrect\_type: If True, we'll raise a TypeError if the final output is not of

the given type.

Returns:

The final output casted to the given type.

"""

if raise\_if\_incorrect\_type and not isinstance(self.final\_output, cls):

raise TypeError(f"Final output is not of type {cls.\_\_name\_\_}")

return cast(T, self.final\_output)

to\_input\_list

to\_input\_list() -> list[TResponseInputItem]

Creates a new input list, merging the original input with all the new items generated.

Source code in src/agents/result.py

def to\_input\_list(self) -> list[TResponseInputItem]:

"""Creates a new input list, merging the original input with all the new items generated."""

original\_items: list[TResponseInputItem] = ItemHelpers.input\_to\_new\_input\_list(self.input)

new\_items = [item.to\_input\_item() for item in self.new\_items]

return original\_items + new\_items

RunResult dataclass

Bases: RunResultBase

Source code in src/agents/result.py

@dataclass

class RunResult(RunResultBase):

\_last\_agent: Agent[Any]

@property

def last\_agent(self) -> Agent[Any]:

"""The last agent that was run."""

return self.\_last\_agent

def \_\_str\_\_(self) -> str:

return pretty\_print\_result(self)

input instance-attribute

input: str | list[TResponseInputItem]

The original input items i.e. the items before run() was called. This may be a mutated version of the input, if there are handoff input filters that mutate the input.

new\_items instance-attribute

new\_items: list[RunItem]

The new items generated during the agent run. These include things like new messages, tool calls and their outputs, etc.

raw\_responses instance-attribute

raw\_responses: list[ModelResponse]

The raw LLM responses generated by the model during the agent run.

final\_output instance-attribute

final\_output: Any

The output of the last agent.

input\_guardrail\_results instance-attribute

input\_guardrail\_results: list[InputGuardrailResult]

Guardrail results for the input messages.

output\_guardrail\_results instance-attribute

output\_guardrail\_results: list[OutputGuardrailResult]

Guardrail results for the final output of the agent.

last\_agent property

last\_agent: Agent[Any]

The last agent that was run.

final\_output\_as

final\_output\_as(

cls: type[T], raise\_if\_incorrect\_type: bool = False

) -> T

A convenience method to cast the final output to a specific type. By default, the cast is only for the typechecker. If you set raise\_if\_incorrect\_type to True, we'll raise a TypeError if the final output is not of the given type.

Parameters:

Name Type Description Default

cls type[T] The type to cast the final output to. required

raise\_if\_incorrect\_type bool If True, we'll raise a TypeError if the final output is not of the given type. False

Returns:

Type Description

T The final output casted to the given type.

Source code in src/agents/result.py

def final\_output\_as(self, cls: type[T], raise\_if\_incorrect\_type: bool = False) -> T:

"""A convenience method to cast the final output to a specific type. By default, the cast

is only for the typechecker. If you set `raise\_if\_incorrect\_type` to True, we'll raise a

TypeError if the final output is not of the given type.

Args:

cls: The type to cast the final output to.

raise\_if\_incorrect\_type: If True, we'll raise a TypeError if the final output is not of

the given type.

Returns:

The final output casted to the given type.

"""

if raise\_if\_incorrect\_type and not isinstance(self.final\_output, cls):

raise TypeError(f"Final output is not of type {cls.\_\_name\_\_}")

return cast(T, self.final\_output)

to\_input\_list

to\_input\_list() -> list[TResponseInputItem]

Creates a new input list, merging the original input with all the new items generated.

Source code in src/agents/result.py

def to\_input\_list(self) -> list[TResponseInputItem]:

"""Creates a new input list, merging the original input with all the new items generated."""

original\_items: list[TResponseInputItem] = ItemHelpers.input\_to\_new\_input\_list(self.input)

new\_items = [item.to\_input\_item() for item in self.new\_items]

return original\_items + new\_items

RunResultStreaming dataclass

Bases: RunResultBase

The result of an agent run in streaming mode. You can use the stream\_events method to receive semantic events as they are generated.

The streaming method will raise: - A MaxTurnsExceeded exception if the agent exceeds the max\_turns limit. - A GuardrailTripwireTriggered exception if a guardrail is tripped.

Source code in src/agents/result.py

@dataclass

class RunResultStreaming(RunResultBase):

"""The result of an agent run in streaming mode. You can use the `stream\_events` method to

receive semantic events as they are generated.

The streaming method will raise:

- A MaxTurnsExceeded exception if the agent exceeds the max\_turns limit.

- A GuardrailTripwireTriggered exception if a guardrail is tripped.

"""

current\_agent: Agent[Any]

"""The current agent that is running."""

current\_turn: int

"""The current turn number."""

max\_turns: int

"""The maximum number of turns the agent can run for."""

final\_output: Any

"""The final output of the agent. This is None until the agent has finished running."""

\_current\_agent\_output\_schema: AgentOutputSchema | None = field(repr=False)

\_trace: Trace | None = field(repr=False)

is\_complete: bool = False

"""Whether the agent has finished running."""

# Queues that the background run\_loop writes to

\_event\_queue: asyncio.Queue[StreamEvent | QueueCompleteSentinel] = field(

default\_factory=asyncio.Queue, repr=False

)

\_input\_guardrail\_queue: asyncio.Queue[InputGuardrailResult] = field(

default\_factory=asyncio.Queue, repr=False

)

# Store the asyncio tasks that we're waiting on

\_run\_impl\_task: asyncio.Task[Any] | None = field(default=None, repr=False)

\_input\_guardrails\_task: asyncio.Task[Any] | None = field(default=None, repr=False)

\_output\_guardrails\_task: asyncio.Task[Any] | None = field(default=None, repr=False)

\_stored\_exception: Exception | None = field(default=None, repr=False)

@property

def last\_agent(self) -> Agent[Any]:

"""The last agent that was run. Updates as the agent run progresses, so the true last agent

is only available after the agent run is complete.

"""

return self.current\_agent

async def stream\_events(self) -> AsyncIterator[StreamEvent]:

"""Stream deltas for new items as they are generated. We're using the types from the

OpenAI Responses API, so these are semantic events: each event has a `type` field that

describes the type of the event, along with the data for that event.

This will raise:

- A MaxTurnsExceeded exception if the agent exceeds the max\_turns limit.

- A GuardrailTripwireTriggered exception if a guardrail is tripped.

"""

while True:

self.\_check\_errors()

if self.\_stored\_exception:

logger.debug("Breaking due to stored exception")

self.is\_complete = True

break

if self.is\_complete and self.\_event\_queue.empty():

break

try:

item = await self.\_event\_queue.get()

except asyncio.CancelledError:

break

if isinstance(item, QueueCompleteSentinel):

self.\_event\_queue.task\_done()

# Check for errors, in case the queue was completed due to an exception

self.\_check\_errors()

break

yield item

self.\_event\_queue.task\_done()

if self.\_trace:

self.\_trace.finish(reset\_current=True)

self.\_cleanup\_tasks()

if self.\_stored\_exception:

raise self.\_stored\_exception

def \_check\_errors(self):

if self.current\_turn > self.max\_turns:

self.\_stored\_exception = MaxTurnsExceeded(f"Max turns ({self.max\_turns}) exceeded")

# Fetch all the completed guardrail results from the queue and raise if needed

while not self.\_input\_guardrail\_queue.empty():

guardrail\_result = self.\_input\_guardrail\_queue.get\_nowait()

if guardrail\_result.output.tripwire\_triggered:

self.\_stored\_exception = InputGuardrailTripwireTriggered(guardrail\_result)

# Check the tasks for any exceptions

if self.\_run\_impl\_task and self.\_run\_impl\_task.done():

exc = self.\_run\_impl\_task.exception()

if exc and isinstance(exc, Exception):

self.\_stored\_exception = exc

if self.\_input\_guardrails\_task and self.\_input\_guardrails\_task.done():

exc = self.\_input\_guardrails\_task.exception()

if exc and isinstance(exc, Exception):

self.\_stored\_exception = exc

if self.\_output\_guardrails\_task and self.\_output\_guardrails\_task.done():

exc = self.\_output\_guardrails\_task.exception()

if exc and isinstance(exc, Exception):

self.\_stored\_exception = exc

def \_cleanup\_tasks(self):

if self.\_run\_impl\_task and not self.\_run\_impl\_task.done():

self.\_run\_impl\_task.cancel()

if self.\_input\_guardrails\_task and not self.\_input\_guardrails\_task.done():

self.\_input\_guardrails\_task.cancel()

if self.\_output\_guardrails\_task and not self.\_output\_guardrails\_task.done():

self.\_output\_guardrails\_task.cancel()

def \_\_str\_\_(self) -> str:

return pretty\_print\_run\_result\_streaming(self)

input instance-attribute

input: str | list[TResponseInputItem]

The original input items i.e. the items before run() was called. This may be a mutated version of the input, if there are handoff input filters that mutate the input.

new\_items instance-attribute

new\_items: list[RunItem]

The new items generated during the agent run. These include things like new messages, tool calls and their outputs, etc.

raw\_responses instance-attribute

raw\_responses: list[ModelResponse]

The raw LLM responses generated by the model during the agent run.

input\_guardrail\_results instance-attribute

input\_guardrail\_results: list[InputGuardrailResult]

Guardrail results for the input messages.

output\_guardrail\_results instance-attribute

output\_guardrail\_results: list[OutputGuardrailResult]

Guardrail results for the final output of the agent.

current\_agent instance-attribute

current\_agent: Agent[Any]

The current agent that is running.

current\_turn instance-attribute

current\_turn: int

The current turn number.

max\_turns instance-attribute

max\_turns: int

The maximum number of turns the agent can run for.

final\_output instance-attribute

final\_output: Any

The final output of the agent. This is None until the agent has finished running.

is\_complete class-attribute instance-attribute

is\_complete: bool = False

Whether the agent has finished running.

last\_agent property

last\_agent: Agent[Any]

The last agent that was run. Updates as the agent run progresses, so the true last agent is only available after the agent run is complete.

final\_output\_as

final\_output\_as(

cls: type[T], raise\_if\_incorrect\_type: bool = False

) -> T

A convenience method to cast the final output to a specific type. By default, the cast is only for the typechecker. If you set raise\_if\_incorrect\_type to True, we'll raise a TypeError if the final output is not of the given type.

Parameters:

Name Type Description Default

cls type[T] The type to cast the final output to. required

raise\_if\_incorrect\_type bool If True, we'll raise a TypeError if the final output is not of the given type. False

Returns:

Type Description

T The final output casted to the given type.

Source code in src/agents/result.py

def final\_output\_as(self, cls: type[T], raise\_if\_incorrect\_type: bool = False) -> T:

"""A convenience method to cast the final output to a specific type. By default, the cast

is only for the typechecker. If you set `raise\_if\_incorrect\_type` to True, we'll raise a

TypeError if the final output is not of the given type.

Args:

cls: The type to cast the final output to.

raise\_if\_incorrect\_type: If True, we'll raise a TypeError if the final output is not of

the given type.

Returns:

The final output casted to the given type.

"""

if raise\_if\_incorrect\_type and not isinstance(self.final\_output, cls):

raise TypeError(f"Final output is not of type {cls.\_\_name\_\_}")

return cast(T, self.final\_output)

to\_input\_list

to\_input\_list() -> list[TResponseInputItem]

Creates a new input list, merging the original input with all the new items generated.

Source code in src/agents/result.py

def to\_input\_list(self) -> list[TResponseInputItem]:

"""Creates a new input list, merging the original input with all the new items generated."""

original\_items: list[TResponseInputItem] = ItemHelpers.input\_to\_new\_input\_list(self.input)

new\_items = [item.to\_input\_item() for item in self.new\_items]

return original\_items + new\_items

stream\_events async

stream\_events() -> AsyncIterator[StreamEvent]

Stream deltas for new items as they are generated. We're using the types from the OpenAI Responses API, so these are semantic events: each event has a type field that describes the type of the event, along with the data for that event.

This will raise: - A MaxTurnsExceeded exception if the agent exceeds the max\_turns limit. - A GuardrailTripwireTriggered exception if a guardrail is tripped.

Source code in src/agents/result.py

async def stream\_events(self) -> AsyncIterator[StreamEvent]:

"""Stream deltas for new items as they are generated. We're using the types from the

OpenAI Responses API, so these are semantic events: each event has a `type` field that

describes the type of the event, along with the data for that event.

This will raise:

- A MaxTurnsExceeded exception if the agent exceeds the max\_turns limit.

- A GuardrailTripwireTriggered exception if a guardrail is tripped.

"""

while True:

self.\_check\_errors()

if self.\_stored\_exception:

logger.debug("Breaking due to stored exception")

self.is\_complete = True

break

if self.is\_complete and self.\_event\_queue.empty():

break

try:

item = await self.\_event\_queue.get()

except asyncio.CancelledError:

break

if isinstance(item, QueueCompleteSentinel):

self.\_event\_queue.task\_done()

# Check for errors, in case the queue was completed due to an exception

self.\_check\_errors()

break

yield item

self.\_event\_queue.task\_done()

if self.\_trace:

self.\_trace.finish(reset\_current=True)

self.\_cleanup\_tasks()

if self.\_stored\_exception:

raise self.\_stored\_exception

Streaming events

StreamEvent module-attribute

StreamEvent: TypeAlias = Union[

RawResponsesStreamEvent,

RunItemStreamEvent,

AgentUpdatedStreamEvent,

]

A streaming event from an agent.

RawResponsesStreamEvent dataclass

Streaming event from the LLM. These are 'raw' events, i.e. they are directly passed through from the LLM.

Source code in src/agents/stream\_events.py

@dataclass

class RawResponsesStreamEvent:

"""Streaming event from the LLM. These are 'raw' events, i.e. they are directly passed through

from the LLM.

"""

data: TResponseStreamEvent

"""The raw responses streaming event from the LLM."""

type: Literal["raw\_response\_event"] = "raw\_response\_event"

"""The type of the event."""

data instance-attribute

data: TResponseStreamEvent

The raw responses streaming event from the LLM.

type class-attribute instance-attribute

type: Literal['raw\_response\_event'] = 'raw\_response\_event'

The type of the event.

RunItemStreamEvent dataclass

Streaming events that wrap a RunItem. As the agent processes the LLM response, it will generate these events for new messages, tool calls, tool outputs, handoffs, etc.

Source code in src/agents/stream\_events.py

@dataclass

class RunItemStreamEvent:

"""Streaming events that wrap a `RunItem`. As the agent processes the LLM response, it will

generate these events for new messages, tool calls, tool outputs, handoffs, etc.

"""

name: Literal[

"message\_output\_created",

"handoff\_requested",

"handoff\_occured",

"tool\_called",

"tool\_output",

"reasoning\_item\_created",

]

"""The name of the event."""

item: RunItem

"""The item that was created."""

type: Literal["run\_item\_stream\_event"] = "run\_item\_stream\_event"

name instance-attribute

name: Literal[

"message\_output\_created",

"handoff\_requested",

"handoff\_occured",

"tool\_called",

"tool\_output",

"reasoning\_item\_created",

]

The name of the event.

item instance-attribute

item: RunItem

The item that was created.

AgentUpdatedStreamEvent dataclass

Event that notifies that there is a new agent running.

Source code in src/agents/stream\_events.py

@dataclass

class AgentUpdatedStreamEvent:

"""Event that notifies that there is a new agent running."""

new\_agent: Agent[Any]

"""The new agent."""

type: Literal["agent\_updated\_stream\_event"] = "agent\_updated\_stream\_event"

new\_agent instance-attribute

new\_agent: Agent[Any]

The new agent.

Handoffs

HandoffInputFilter module-attribute

HandoffInputFilter: TypeAlias = Callable[

[HandoffInputData], HandoffInputData

]

A function that filters the input data passed to the next agent.

HandoffInputData dataclass

Source code in src/agents/handoffs.py

@dataclass(frozen=True)

class HandoffInputData:

input\_history: str | tuple[TResponseInputItem, ...]

"""

The input history before `Runner.run()` was called.

"""

pre\_handoff\_items: tuple[RunItem, ...]

"""

The items generated before the agent turn where the handoff was invoked.

"""

new\_items: tuple[RunItem, ...]

"""

The new items generated during the current agent turn, including the item that triggered the

handoff and the tool output message representing the response from the handoff output.

"""

input\_history instance-attribute

input\_history: str | tuple[TResponseInputItem, ...]

The input history before Runner.run() was called.

pre\_handoff\_items instance-attribute

pre\_handoff\_items: tuple[RunItem, ...]

The items generated before the agent turn where the handoff was invoked.

new\_items instance-attribute

new\_items: tuple[RunItem, ...]

The new items generated during the current agent turn, including the item that triggered the handoff and the tool output message representing the response from the handoff output.

Handoff dataclass

Bases: Generic[TContext]

A handoff is when an agent delegates a task to another agent. For example, in a customer support scenario you might have a "triage agent" that determines which agent should handle the user's request, and sub-agents that specialize in different areas like billing, account management, etc.

Source code in src/agents/handoffs.py

@dataclass

class Handoff(Generic[TContext]):

"""A handoff is when an agent delegates a task to another agent.

For example, in a customer support scenario you might have a "triage agent" that determines

which agent should handle the user's request, and sub-agents that specialize in different

areas like billing, account management, etc.

"""

tool\_name: str

"""The name of the tool that represents the handoff."""

tool\_description: str

"""The description of the tool that represents the handoff."""

input\_json\_schema: dict[str, Any]

"""The JSON schema for the handoff input. Can be empty if the handoff does not take an input.

"""

on\_invoke\_handoff: Callable[[RunContextWrapper[Any], str], Awaitable[Agent[TContext]]]

"""The function that invokes the handoff. The parameters passed are:

1. The handoff run context

2. The arguments from the LLM, as a JSON string. Empty string if input\_json\_schema is empty.

Must return an agent.

"""

agent\_name: str

"""The name of the agent that is being handed off to."""

input\_filter: HandoffInputFilter | None = None

"""A function that filters the inputs that are passed to the next agent. By default, the new

agent sees the entire conversation history. In some cases, you may want to filter inputs e.g.

to remove older inputs, or remove tools from existing inputs.

The function will receive the entire conversation history so far, including the input item

that triggered the handoff and a tool call output item representing the handoff tool's output.

You are free to modify the input history or new items as you see fit. The next agent that

runs will receive `handoff\_input\_data.all\_items`.

IMPORTANT: in streaming mode, we will not stream anything as a result of this function. The

items generated before will already have been streamed.

"""

strict\_json\_schema: bool = True

"""Whether the input JSON schema is in strict mode. We \*\*strongly\*\* recommend setting this to

True, as it increases the likelihood of correct JSON input.

"""

def get\_transfer\_message(self, agent: Agent[Any]) -> str:

base = f"{{'assistant': '{agent.name}'}}"

return base

@classmethod

def default\_tool\_name(cls, agent: Agent[Any]) -> str:

return \_transforms.transform\_string\_function\_style(f"transfer\_to\_{agent.name}")

@classmethod

def default\_tool\_description(cls, agent: Agent[Any]) -> str:

return (

f"Handoff to the {agent.name} agent to handle the request. "

f"{agent.handoff\_description or ''}"

)

tool\_name instance-attribute

tool\_name: str

The name of the tool that represents the handoff.

tool\_description instance-attribute

tool\_description: str

The description of the tool that represents the handoff.

input\_json\_schema instance-attribute

input\_json\_schema: dict[str, Any]

The JSON schema for the handoff input. Can be empty if the handoff does not take an input.

on\_invoke\_handoff instance-attribute

on\_invoke\_handoff: Callable[

[RunContextWrapper[Any], str],

Awaitable[Agent[TContext]],

]

The function that invokes the handoff. The parameters passed are: 1. The handoff run context 2. The arguments from the LLM, as a JSON string. Empty string if input\_json\_schema is empty.

Must return an agent.

agent\_name instance-attribute

agent\_name: str

The name of the agent that is being handed off to.

input\_filter class-attribute instance-attribute

input\_filter: HandoffInputFilter | None = None

A function that filters the inputs that are passed to the next agent. By default, the new agent sees the entire conversation history. In some cases, you may want to filter inputs e.g. to remove older inputs, or remove tools from existing inputs.

The function will receive the entire conversation history so far, including the input item that triggered the handoff and a tool call output item representing the handoff tool's output.

You are free to modify the input history or new items as you see fit. The next agent that runs will receive handoff\_input\_data.all\_items.

IMPORTANT: in streaming mode, we will not stream anything as a result of this function. The items generated before will already have been streamed.

strict\_json\_schema class-attribute instance-attribute

strict\_json\_schema: bool = True

Whether the input JSON schema is in strict mode. We strongly recommend setting this to True, as it increases the likelihood of correct JSON input.

handoff

handoff(

agent: Agent[TContext],

\*,

tool\_name\_override: str | None = None,

tool\_description\_override: str | None = None,

input\_filter: Callable[

[HandoffInputData], HandoffInputData

]

| None = None,

) -> Handoff[TContext]

handoff(

agent: Agent[TContext],

\*,

on\_handoff: OnHandoffWithInput[THandoffInput],

input\_type: type[THandoffInput],

tool\_description\_override: str | None = None,

tool\_name\_override: str | None = None,

input\_filter: Callable[

[HandoffInputData], HandoffInputData

]

| None = None,

) -> Handoff[TContext]

handoff(

agent: Agent[TContext],

\*,

on\_handoff: OnHandoffWithoutInput,

tool\_description\_override: str | None = None,

tool\_name\_override: str | None = None,

input\_filter: Callable[

[HandoffInputData], HandoffInputData

]

| None = None,

) -> Handoff[TContext]

handoff(

agent: Agent[TContext],

tool\_name\_override: str | None = None,

tool\_description\_override: str | None = None,

on\_handoff: OnHandoffWithInput[THandoffInput]

| OnHandoffWithoutInput

| None = None,

input\_type: type[THandoffInput] | None = None,

input\_filter: Callable[

[HandoffInputData], HandoffInputData

]

| None = None,

) -> Handoff[TContext]

Create a handoff from an agent.

Parameters:

Name Type Description Default

agent Agent[TContext] The agent to handoff to, or a function that returns an agent. required

tool\_name\_override str | None Optional override for the name of the tool that represents the handoff. None

tool\_description\_override str | None Optional override for the description of the tool that represents the handoff. None

on\_handoff OnHandoffWithInput[THandoffInput] | OnHandoffWithoutInput | None A function that runs when the handoff is invoked. None

input\_type type[THandoffInput] | None the type of the input to the handoff. If provided, the input will be validated against this type. Only relevant if you pass a function that takes an input. None

input\_filter Callable[[HandoffInputData], HandoffInputData] | None a function that filters the inputs that are passed to the next agent. None

Source code in src/agents/handoffs.py

def handoff(

agent: Agent[TContext],

tool\_name\_override: str | None = None,

tool\_description\_override: str | None = None,

on\_handoff: OnHandoffWithInput[THandoffInput] | OnHandoffWithoutInput | None = None,

input\_type: type[THandoffInput] | None = None,

input\_filter: Callable[[HandoffInputData], HandoffInputData] | None = None,

) -> Handoff[TContext]:

"""Create a handoff from an agent.

Args:

agent: The agent to handoff to, or a function that returns an agent.

tool\_name\_override: Optional override for the name of the tool that represents the handoff.

tool\_description\_override: Optional override for the description of the tool that

represents the handoff.

on\_handoff: A function that runs when the handoff is invoked.

input\_type: the type of the input to the handoff. If provided, the input will be validated

against this type. Only relevant if you pass a function that takes an input.

input\_filter: a function that filters the inputs that are passed to the next agent.

"""

assert (on\_handoff and input\_type) or not (on\_handoff and input\_type), (

"You must provide either both on\_input and input\_type, or neither"

)

type\_adapter: TypeAdapter[Any] | None

if input\_type is not None:

assert callable(on\_handoff), "on\_handoff must be callable"

sig = inspect.signature(on\_handoff)

if len(sig.parameters) != 2:

raise UserError("on\_handoff must take two arguments: context and input")

type\_adapter = TypeAdapter(input\_type)

input\_json\_schema = type\_adapter.json\_schema()

else:

type\_adapter = None

input\_json\_schema = {}

if on\_handoff is not None:

sig = inspect.signature(on\_handoff)

if len(sig.parameters) != 1:

raise UserError("on\_handoff must take one argument: context")

async def \_invoke\_handoff(

ctx: RunContextWrapper[Any], input\_json: str | None = None

) -> Agent[Any]:

if input\_type is not None and type\_adapter is not None:

if input\_json is None:

\_error\_tracing.attach\_error\_to\_current\_span(

SpanError(

message="Handoff function expected non-null input, but got None",

data={"details": "input\_json is None"},

)

)

raise ModelBehaviorError("Handoff function expected non-null input, but got None")

validated\_input = \_json.validate\_json(

json\_str=input\_json,

type\_adapter=type\_adapter,

partial=False,

)

input\_func = cast(OnHandoffWithInput[THandoffInput], on\_handoff)

if inspect.iscoroutinefunction(input\_func):

await input\_func(ctx, validated\_input)

else:

input\_func(ctx, validated\_input)

elif on\_handoff is not None:

no\_input\_func = cast(OnHandoffWithoutInput, on\_handoff)

if inspect.iscoroutinefunction(no\_input\_func):

await no\_input\_func(ctx)

else:

no\_input\_func(ctx)

return agent

tool\_name = tool\_name\_override or Handoff.default\_tool\_name(agent)

tool\_description = tool\_description\_override or Handoff.default\_tool\_description(agent)

# Always ensure the input JSON schema is in strict mode

# If there is a need, we can make this configurable in the future

input\_json\_schema = ensure\_strict\_json\_schema(input\_json\_schema)

return Handoff(

tool\_name=tool\_name,

tool\_description=tool\_description,

input\_json\_schema=input\_json\_schema,

on\_invoke\_handoff=\_invoke\_handoff,

input\_filter=input\_filter,

agent\_name=agent.name,

)

Lifecycle

RunHooks

Bases: Generic[TContext]

A class that receives callbacks on various lifecycle events in an agent run. Subclass and override the methods you need.

on\_agent\_start async

on\_agent\_start(

context: RunContextWrapper[TContext],

agent: Agent[TContext],

) -> None

Called before the agent is invoked. Called each time the current agent changes.

on\_agent\_end async

on\_agent\_end(

context: RunContextWrapper[TContext],

agent: Agent[TContext],

output: Any,

) -> None

Called when the agent produces a final output.

on\_handoff async

on\_handoff(

context: RunContextWrapper[TContext],

from\_agent: Agent[TContext],

to\_agent: Agent[TContext],

) -> None

Called when a handoff occurs.

on\_tool\_start async

on\_tool\_start(

context: RunContextWrapper[TContext],

agent: Agent[TContext],

tool: Tool,

) -> None

Called before a tool is invoked.

on\_tool\_end async

on\_tool\_end(

context: RunContextWrapper[TContext],

agent: Agent[TContext],

tool: Tool,

result: str,

) -> None

Called after a tool is invoked.

AgentHooks

Bases: Generic[TContext]

A class that receives callbacks on various lifecycle events for a specific agent. You can set this on agent.hooks to receive events for that specific agent.

Subclass and override the methods you need.

on\_start async

on\_start(

context: RunContextWrapper[TContext],

agent: Agent[TContext],

) -> None

Called before the agent is invoked. Called each time the running agent is changed to this agent.

on\_end async

on\_end(

context: RunContextWrapper[TContext],

agent: Agent[TContext],

output: Any,

) -> None

Called when the agent produces a final output.

on\_handoff async

on\_handoff(

context: RunContextWrapper[TContext],

agent: Agent[TContext],

source: Agent[TContext],

) -> None

Called when the agent is being handed off to. The source is the agent that is handing off to this agent.

on\_tool\_start async

on\_tool\_start(

context: RunContextWrapper[TContext],

agent: Agent[TContext],

tool: Tool,

) -> None

Called before a tool is invoked.

on\_tool\_end async

on\_tool\_end(

context: RunContextWrapper[TContext],

agent: Agent[TContext],

tool: Tool,

result: str,

) -> None

Called after a tool is invoked.

Items

TResponse module-attribute

TResponse = Response

A type alias for the Response type from the OpenAI SDK.

TResponseInputItem module-attribute

TResponseInputItem = ResponseInputItemParam

A type alias for the ResponseInputItemParam type from the OpenAI SDK.

TResponseOutputItem module-attribute

TResponseOutputItem = ResponseOutputItem

A type alias for the ResponseOutputItem type from the OpenAI SDK.

TResponseStreamEvent module-attribute

TResponseStreamEvent = ResponseStreamEvent

A type alias for the ResponseStreamEvent type from the OpenAI SDK.

ToolCallItemTypes module-attribute

ToolCallItemTypes: TypeAlias = Union[

ResponseFunctionToolCall,

ResponseComputerToolCall,

ResponseFileSearchToolCall,

ResponseFunctionWebSearch,

]

A type that represents a tool call item.

RunItem module-attribute

RunItem: TypeAlias = Union[

MessageOutputItem,

HandoffCallItem,

HandoffOutputItem,

ToolCallItem,

ToolCallOutputItem,

ReasoningItem,

]

An item generated by an agent.

RunItemBase dataclass

Bases: Generic[T], ABC

Source code in src/agents/items.py

agent instance-attribute

agent: Agent[Any]

The agent whose run caused this item to be generated.

raw\_item instance-attribute

raw\_item: T

The raw Responses item from the run. This will always be a either an output item (i.e. openai.types.responses.ResponseOutputItem or an input item (i.e. openai.types.responses.ResponseInputItemParam).

to\_input\_item

to\_input\_item() -> TResponseInputItem

Converts this item into an input item suitable for passing to the model.

Source code in src/agents/items.py

def to\_input\_item(self) -> TResponseInputItem:

"""Converts this item into an input item suitable for passing to the model."""

if isinstance(self.raw\_item, dict):

# We know that input items are dicts, so we can ignore the type error

return self.raw\_item # type: ignore

elif isinstance(self.raw\_item, BaseModel):

# All output items are Pydantic models that can be converted to input items.

return self.raw\_item.model\_dump(exclude\_unset=True) # type: ignore

else:

raise AgentsException(f"Unexpected raw item type: {type(self.raw\_item)}")

MessageOutputItem dataclass

Bases: RunItemBase[ResponseOutputMessage]

Represents a message from the LLM.

Source code in src/agents/items.py

@dataclass

class MessageOutputItem(RunItemBase[ResponseOutputMessage]):

"""Represents a message from the LLM."""

raw\_item: ResponseOutputMessage

"""The raw response output message."""

type: Literal["message\_output\_item"] = "message\_output\_item"

agent instance-attribute

agent: Agent[Any]

The agent whose run caused this item to be generated.

raw\_item instance-attribute

raw\_item: ResponseOutputMessage

The raw response output message.

to\_input\_item

to\_input\_item() -> TResponseInputItem

Converts this item into an input item suitable for passing to the model.

Source code in src/agents/items.py

def to\_input\_item(self) -> TResponseInputItem:

"""Converts this item into an input item suitable for passing to the model."""

if isinstance(self.raw\_item, dict):

# We know that input items are dicts, so we can ignore the type error

return self.raw\_item # type: ignore

elif isinstance(self.raw\_item, BaseModel):

# All output items are Pydantic models that can be converted to input items.

return self.raw\_item.model\_dump(exclude\_unset=True) # type: ignore

else:

raise AgentsException(f"Unexpected raw item type: {type(self.raw\_item)}")

HandoffCallItem dataclass

Bases: RunItemBase[ResponseFunctionToolCall]

Represents a tool call for a handoff from one agent to another.

Source code in src/agents/items.py

@dataclass

class HandoffCallItem(RunItemBase[ResponseFunctionToolCall]):

"""Represents a tool call for a handoff from one agent to another."""

raw\_item: ResponseFunctionToolCall

"""The raw response function tool call that represents the handoff."""

type: Literal["handoff\_call\_item"] = "handoff\_call\_item"

agent instance-attribute

agent: Agent[Any]

The agent whose run caused this item to be generated.

raw\_item instance-attribute

raw\_item: ResponseFunctionToolCall

The raw response function tool call that represents the handoff.

to\_input\_item

to\_input\_item() -> TResponseInputItem

Converts this item into an input item suitable for passing to the model.

Source code in src/agents/items.py

def to\_input\_item(self) -> TResponseInputItem:

"""Converts this item into an input item suitable for passing to the model."""

if isinstance(self.raw\_item, dict):

# We know that input items are dicts, so we can ignore the type error

return self.raw\_item # type: ignore

elif isinstance(self.raw\_item, BaseModel):

# All output items are Pydantic models that can be converted to input items.

return self.raw\_item.model\_dump(exclude\_unset=True) # type: ignore

else:

raise AgentsException(f"Unexpected raw item type: {type(self.raw\_item)}")

HandoffOutputItem dataclass

Bases: RunItemBase[TResponseInputItem]

Represents the output of a handoff.

Source code in src/agents/items.py

@dataclass

class HandoffOutputItem(RunItemBase[TResponseInputItem]):

"""Represents the output of a handoff."""

raw\_item: TResponseInputItem

"""The raw input item that represents the handoff taking place."""

source\_agent: Agent[Any]

"""The agent that made the handoff."""

target\_agent: Agent[Any]

"""The agent that is being handed off to."""

type: Literal["handoff\_output\_item"] = "handoff\_output\_item"

agent instance-attribute

agent: Agent[Any]

The agent whose run caused this item to be generated.

raw\_item instance-attribute

raw\_item: TResponseInputItem

The raw input item that represents the handoff taking place.

source\_agent instance-attribute

source\_agent: Agent[Any]

The agent that made the handoff.

target\_agent instance-attribute

target\_agent: Agent[Any]

The agent that is being handed off to.

to\_input\_item

to\_input\_item() -> TResponseInputItem

Converts this item into an input item suitable for passing to the model.

Source code in src/agents/items.py

def to\_input\_item(self) -> TResponseInputItem:

"""Converts this item into an input item suitable for passing to the model."""

if isinstance(self.raw\_item, dict):

# We know that input items are dicts, so we can ignore the type error

return self.raw\_item # type: ignore

elif isinstance(self.raw\_item, BaseModel):

# All output items are Pydantic models that can be converted to input items.

return self.raw\_item.model\_dump(exclude\_unset=True) # type: ignore

else:

raise AgentsException(f"Unexpected raw item type: {type(self.raw\_item)}")

ToolCallItem dataclass

Bases: RunItemBase[ToolCallItemTypes]

Represents a tool call e.g. a function call or computer action call.

Source code in src/agents/items.py

@dataclass

class ToolCallItem(RunItemBase[ToolCallItemTypes]):

"""Represents a tool call e.g. a function call or computer action call."""

raw\_item: ToolCallItemTypes

"""The raw tool call item."""

type: Literal["tool\_call\_item"] = "tool\_call\_item"

agent instance-attribute

agent: Agent[Any]

The agent whose run caused this item to be generated.

raw\_item instance-attribute

raw\_item: ToolCallItemTypes

The raw tool call item.

to\_input\_item

to\_input\_item() -> TResponseInputItem

Converts this item into an input item suitable for passing to the model.

Source code in src/agents/items.py

def to\_input\_item(self) -> TResponseInputItem:

"""Converts this item into an input item suitable for passing to the model."""

if isinstance(self.raw\_item, dict):

# We know that input items are dicts, so we can ignore the type error

return self.raw\_item # type: ignore

elif isinstance(self.raw\_item, BaseModel):

# All output items are Pydantic models that can be converted to input items.

return self.raw\_item.model\_dump(exclude\_unset=True) # type: ignore

else:

raise AgentsException(f"Unexpected raw item type: {type(self.raw\_item)}")

ToolCallOutputItem dataclass

Bases: RunItemBase[Union[FunctionCallOutput, ComputerCallOutput]]

Represents the output of a tool call.

Source code in src/agents/items.py

@dataclass

class ToolCallOutputItem(RunItemBase[Union[FunctionCallOutput, ComputerCallOutput]]):

"""Represents the output of a tool call."""

raw\_item: FunctionCallOutput | ComputerCallOutput

"""The raw item from the model."""

output: Any

"""The output of the tool call. This is whatever the tool call returned; the `raw\_item`

contains a string representation of the output.

"""

type: Literal["tool\_call\_output\_item"] = "tool\_call\_output\_item"

agent instance-attribute

agent: Agent[Any]

The agent whose run caused this item to be generated.

raw\_item instance-attribute

raw\_item: FunctionCallOutput | ComputerCallOutput

The raw item from the model.

output instance-attribute

output: Any

The output of the tool call. This is whatever the tool call returned; the raw\_item contains a string representation of the output.

to\_input\_item

to\_input\_item() -> TResponseInputItem

Converts this item into an input item suitable for passing to the model.

Source code in src/agents/items.py

def to\_input\_item(self) -> TResponseInputItem:

"""Converts this item into an input item suitable for passing to the model."""

if isinstance(self.raw\_item, dict):

# We know that input items are dicts, so we can ignore the type error

return self.raw\_item # type: ignore

elif isinstance(self.raw\_item, BaseModel):

# All output items are Pydantic models that can be converted to input items.

return self.raw\_item.model\_dump(exclude\_unset=True) # type: ignore

else:

raise AgentsException(f"Unexpected raw item type: {type(self.raw\_item)}")

ReasoningItem dataclass

Bases: RunItemBase[ResponseReasoningItem]

Represents a reasoning item.

Source code in src/agents/items.py

@dataclass

class ReasoningItem(RunItemBase[ResponseReasoningItem]):

"""Represents a reasoning item."""

raw\_item: ResponseReasoningItem

"""The raw reasoning item."""

type: Literal["reasoning\_item"] = "reasoning\_item"

agent instance-attribute

agent: Agent[Any]

The agent whose run caused this item to be generated.

raw\_item instance-attribute

raw\_item: ResponseReasoningItem

The raw reasoning item.

to\_input\_item

to\_input\_item() -> TResponseInputItem

Converts this item into an input item suitable for passing to the model.

Source code in src/agents/items.py

def to\_input\_item(self) -> TResponseInputItem:

"""Converts this item into an input item suitable for passing to the model."""

if isinstance(self.raw\_item, dict):

# We know that input items are dicts, so we can ignore the type error

return self.raw\_item # type: ignore

elif isinstance(self.raw\_item, BaseModel):

# All output items are Pydantic models that can be converted to input items.

return self.raw\_item.model\_dump(exclude\_unset=True) # type: ignore

else:

raise AgentsException(f"Unexpected raw item type: {type(self.raw\_item)}")

ModelResponse dataclass

Source code in src/agents/items.py

@dataclass

class ModelResponse:

output: list[TResponseOutputItem]

"""A list of outputs (messages, tool calls, etc) generated by the model"""

usage: Usage

"""The usage information for the response."""

referenceable\_id: str | None

"""An ID for the response which can be used to refer to the response in subsequent calls to the

model. Not supported by all model providers.

"""

def to\_input\_items(self) -> list[TResponseInputItem]:

"""Convert the output into a list of input items suitable for passing to the model."""

# We happen to know that the shape of the Pydantic output items are the same as the

# equivalent TypedDict input items, so we can just convert each one.

# This is also tested via unit tests.

return [it.model\_dump(exclude\_unset=True) for it in self.output] # type: ignore

output instance-attribute

output: list[TResponseOutputItem]

A list of outputs (messages, tool calls, etc) generated by the model

usage instance-attribute

usage: Usage

The usage information for the response.

referenceable\_id instance-attribute

referenceable\_id: str | None

An ID for the response which can be used to refer to the response in subsequent calls to the model. Not supported by all model providers.

to\_input\_items

to\_input\_items() -> list[TResponseInputItem]

Convert the output into a list of input items suitable for passing to the model.

Source code in src/agents/items.py

def to\_input\_items(self) -> list[TResponseInputItem]:

"""Convert the output into a list of input items suitable for passing to the model."""

# We happen to know that the shape of the Pydantic output items are the same as the

# equivalent TypedDict input items, so we can just convert each one.

# This is also tested via unit tests.

return [it.model\_dump(exclude\_unset=True) for it in self.output] # type: ignore

ItemHelpers

Source code in src/agents/items.py

class ItemHelpers:

@classmethod

def extract\_last\_content(cls, message: TResponseOutputItem) -> str:

"""Extracts the last text content or refusal from a message."""

if not isinstance(message, ResponseOutputMessage):

return ""

last\_content = message.content[-1]

if isinstance(last\_content, ResponseOutputText):

return last\_content.text

elif isinstance(last\_content, ResponseOutputRefusal):

return last\_content.refusal

else:

raise ModelBehaviorError(f"Unexpected content type: {type(last\_content)}")

@classmethod

def extract\_last\_text(cls, message: TResponseOutputItem) -> str | None:

"""Extracts the last text content from a message, if any. Ignores refusals."""

if isinstance(message, ResponseOutputMessage):

last\_content = message.content[-1]

if isinstance(last\_content, ResponseOutputText):

return last\_content.text

return None

@classmethod

def input\_to\_new\_input\_list(

cls, input: str | list[TResponseInputItem]

) -> list[TResponseInputItem]:

"""Converts a string or list of input items into a list of input items."""

if isinstance(input, str):

return [

{

"content": input,

"role": "user",

}

]

return copy.deepcopy(input)

@classmethod

def text\_message\_outputs(cls, items: list[RunItem]) -> str:

"""Concatenates all the text content from a list of message output items."""

text = ""

for item in items:

if isinstance(item, MessageOutputItem):

text += cls.text\_message\_output(item)

return text

@classmethod

def text\_message\_output(cls, message: MessageOutputItem) -> str:

"""Extracts all the text content from a single message output item."""

text = ""

for item in message.raw\_item.content:

if isinstance(item, ResponseOutputText):

text += item.text

return text

@classmethod

def tool\_call\_output\_item(

cls, tool\_call: ResponseFunctionToolCall, output: str

) -> FunctionCallOutput:

"""Creates a tool call output item from a tool call and its output."""

return {

"call\_id": tool\_call.call\_id,

"output": output,

"type": "function\_call\_output",

}

extract\_last\_content classmethod

extract\_last\_content(message: TResponseOutputItem) -> str

Extracts the last text content or refusal from a message.

Source code in src/agents/items.py

@classmethod

def extract\_last\_content(cls, message: TResponseOutputItem) -> str:

"""Extracts the last text content or refusal from a message."""

if not isinstance(message, ResponseOutputMessage):

return ""

last\_content = message.content[-1]

if isinstance(last\_content, ResponseOutputText):

return last\_content.text

elif isinstance(last\_content, ResponseOutputRefusal):

return last\_content.refusal

else:

raise ModelBehaviorError(f"Unexpected content type: {type(last\_content)}")

extract\_last\_text classmethod

extract\_last\_text(

message: TResponseOutputItem,

) -> str | None

Extracts the last text content from a message, if any. Ignores refusals.

Source code in src/agents/items.py

@classmethod

def extract\_last\_text(cls, message: TResponseOutputItem) -> str | None:

"""Extracts the last text content from a message, if any. Ignores refusals."""

if isinstance(message, ResponseOutputMessage):

last\_content = message.content[-1]

if isinstance(last\_content, ResponseOutputText):

return last\_content.text

return None

input\_to\_new\_input\_list classmethod

input\_to\_new\_input\_list(

input: str | list[TResponseInputItem],

) -> list[TResponseInputItem]

Converts a string or list of input items into a list of input items.

Source code in src/agents/items.py

@classmethod

def input\_to\_new\_input\_list(

cls, input: str | list[TResponseInputItem]

) -> list[TResponseInputItem]:

"""Converts a string or list of input items into a list of input items."""

if isinstance(input, str):

return [

{

"content": input,

"role": "user",

}

]

return copy.deepcopy(input)

text\_message\_outputs classmethod

text\_message\_outputs(items: list[RunItem]) -> str

Concatenates all the text content from a list of message output items.

Source code in src/agents/items.py

@classmethod

def text\_message\_outputs(cls, items: list[RunItem]) -> str:

"""Concatenates all the text content from a list of message output items."""

text = ""

for item in items:

if isinstance(item, MessageOutputItem):

text += cls.text\_message\_output(item)

return text

text\_message\_output classmethod

text\_message\_output(message: MessageOutputItem) -> str

Extracts all the text content from a single message output item.

Source code in src/agents/items.py

@classmethod

def text\_message\_output(cls, message: MessageOutputItem) -> str:

"""Extracts all the text content from a single message output item."""

text = ""

for item in message.raw\_item.content:

if isinstance(item, ResponseOutputText):

text += item.text

return text

tool\_call\_output\_item classmethod

tool\_call\_output\_item(

tool\_call: ResponseFunctionToolCall, output: str

) -> FunctionCallOutput

Creates a tool call output item from a tool call and its output.

Source code in src/agents/items.py

@classmethod

def tool\_call\_output\_item(

cls, tool\_call: ResponseFunctionToolCall, output: str

) -> FunctionCallOutput:

"""Creates a tool call output item from a tool call and its output."""

return {

"call\_id": tool\_call.call\_id,

"output": output,

"type": "function\_call\_output",

}

Run context

RunContextWrapper dataclass

Bases: Generic[TContext]

This wraps the context object that you passed to Runner.run(). It also contains information about the usage of the agent run so far.

NOTE: Contexts are not passed to the LLM. They're a way to pass dependencies and data to code you implement, like tool functions, callbacks, hooks, etc.

Source code in src/agents/run\_context.py

@dataclass

class RunContextWrapper(Generic[TContext]):

"""This wraps the context object that you passed to `Runner.run()`. It also contains

information about the usage of the agent run so far.

NOTE: Contexts are not passed to the LLM. They're a way to pass dependencies and data to code

you implement, like tool functions, callbacks, hooks, etc.

"""

context: TContext

"""The context object (or None), passed by you to `Runner.run()`"""

usage: Usage = field(default\_factory=Usage)

"""The usage of the agent run so far. For streamed responses, the usage will be stale until the

last chunk of the stream is processed.

"""

context instance-attribute

context: TContext

The context object (or None), passed by you to Runner.run()

usage class-attribute instance-attribute

usage: Usage = field(default\_factory=Usage)

The usage of the agent run so far. For streamed responses, the usage will be stale until the last chunk of the stream is processed.

Usage

Usage dataclass

Source code in src/agents/usage.py

@dataclass

class Usage:

requests: int = 0

"""Total requests made to the LLM API."""

input\_tokens: int = 0

"""Total input tokens sent, across all requests."""

output\_tokens: int = 0

"""Total output tokens received, across all requests."""

total\_tokens: int = 0

"""Total tokens sent and received, across all requests."""

def add(self, other: "Usage") -> None:

self.requests += other.requests if other.requests else 0

self.input\_tokens += other.input\_tokens if other.input\_tokens else 0

self.output\_tokens += other.output\_tokens if other.output\_tokens else 0

self.total\_tokens += other.total\_tokens if other.total\_tokens else 0

requests class-attribute instance-attribute

requests: int = 0

Total requests made to the LLM API.

input\_tokens class-attribute instance-attribute

input\_tokens: int = 0

Total input tokens sent, across all requests.

output\_tokens class-attribute instance-attribute

output\_tokens: int = 0

Total output tokens received, across all requests.

total\_tokens class-attribute instance-attribute

total\_tokens: int = 0

Total tokens sent and received, across all requests.

Exceptions

AgentsException

Bases: Exception

Base class for all exceptions in the Agents SDK.

Source code in src/agents/exceptions.py

class AgentsException(Exception):

"""Base class for all exceptions in the Agents SDK."""

MaxTurnsExceeded

Bases: AgentsException

Exception raised when the maximum number of turns is exceeded.

Source code in src/agents/exceptions.py

class MaxTurnsExceeded(AgentsException):

"""Exception raised when the maximum number of turns is exceeded."""

message: str

def \_\_init\_\_(self, message: str):

self.message = message

ModelBehaviorError

Bases: AgentsException

Exception raised when the model does something unexpected, e.g. calling a tool that doesn't exist, or providing malformed JSON.

Source code in src/agents/exceptions.py

class ModelBehaviorError(AgentsException):

"""Exception raised when the model does something unexpected, e.g. calling a tool that doesn't

exist, or providing malformed JSON.

"""

message: str

def \_\_init\_\_(self, message: str):

self.message = message

UserError

Bases: AgentsException

Exception raised when the user makes an error using the SDK.

Source code in src/agents/exceptions.py

class UserError(AgentsException):

"""Exception raised when the user makes an error using the SDK."""

message: str

def \_\_init\_\_(self, message: str):

self.message = message

InputGuardrailTripwireTriggered

Bases: AgentsException

Exception raised when a guardrail tripwire is triggered.

Source code in src/agents/exceptions.py

class InputGuardrailTripwireTriggered(AgentsException):

"""Exception raised when a guardrail tripwire is triggered."""

guardrail\_result: "InputGuardrailResult"

"""The result data of the guardrail that was triggered."""

def \_\_init\_\_(self, guardrail\_result: "InputGuardrailResult"):

self.guardrail\_result = guardrail\_result

super().\_\_init\_\_(

f"Guardrail {guardrail\_result.guardrail.\_\_class\_\_.\_\_name\_\_} triggered tripwire"

)

guardrail\_result instance-attribute

guardrail\_result: InputGuardrailResult = guardrail\_result

The result data of the guardrail that was triggered.

OutputGuardrailTripwireTriggered

Bases: AgentsException

Exception raised when a guardrail tripwire is triggered.

Source code in src/agents/exceptions.py

class OutputGuardrailTripwireTriggered(AgentsException):

"""Exception raised when a guardrail tripwire is triggered."""

guardrail\_result: "OutputGuardrailResult"

"""The result data of the guardrail that was triggered."""

def \_\_init\_\_(self, guardrail\_result: "OutputGuardrailResult"):

self.guardrail\_result = guardrail\_result

super().\_\_init\_\_(

f"Guardrail {guardrail\_result.guardrail.\_\_class\_\_.\_\_name\_\_} triggered tripwire"

)

guardrail\_result instance-attribute

guardrail\_result: OutputGuardrailResult = guardrail\_result

Guardrails

GuardrailFunctionOutput dataclass

The output of a guardrail function.

Source code in src/agents/guardrail.py

@dataclass

class GuardrailFunctionOutput:

"""The output of a guardrail function."""

output\_info: Any

"""

Optional information about the guardrail's output. For example, the guardrail could include

information about the checks it performed and granular results.

"""

tripwire\_triggered: bool

"""

Whether the tripwire was triggered. If triggered, the agent's execution will be halted.

"""

output\_info instance-attribute

output\_info: Any

Optional information about the guardrail's output. For example, the guardrail could include information about the checks it performed and granular results.

tripwire\_triggered instance-attribute

tripwire\_triggered: bool

Whether the tripwire was triggered. If triggered, the agent's execution will be halted.

InputGuardrailResult dataclass

The result of a guardrail run.

Source code in src/agents/guardrail.py

@dataclass

class InputGuardrailResult:

"""The result of a guardrail run."""

guardrail: InputGuardrail[Any]

"""

The guardrail that was run.

"""

output: GuardrailFunctionOutput

"""The output of the guardrail function."""

guardrail instance-attribute

guardrail: InputGuardrail[Any]

The guardrail that was run.

output instance-attribute

output: GuardrailFunctionOutput

The output of the guardrail function.

OutputGuardrailResult dataclass

The result of a guardrail run.

Source code in src/agents/guardrail.py

@dataclass

class OutputGuardrailResult:

"""The result of a guardrail run."""

guardrail: OutputGuardrail[Any]

"""

The guardrail that was run.

"""

agent\_output: Any

"""

The output of the agent that was checked by the guardrail.

"""

agent: Agent[Any]

"""

The agent that was checked by the guardrail.

"""

output: GuardrailFunctionOutput

"""The output of the guardrail function."""

guardrail instance-attribute

guardrail: OutputGuardrail[Any]

The guardrail that was run.

agent\_output instance-attribute

agent\_output: Any

The output of the agent that was checked by the guardrail.

agent instance-attribute

agent: Agent[Any]

The agent that was checked by the guardrail.

output instance-attribute

output: GuardrailFunctionOutput

The output of the guardrail function.

InputGuardrail dataclass

Bases: Generic[TContext]

Input guardrails are checks that run in parallel to the agent's execution. They can be used to do things like: - Check if input messages are off-topic - Take over control of the agent's execution if an unexpected input is detected

You can use the @input\_guardrail() decorator to turn a function into an InputGuardrail, or create an InputGuardrail manually.

Guardrails return a GuardrailResult. If result.tripwire\_triggered is True, the agent execution will immediately stop and a InputGuardrailTripwireTriggered exception will be raised

Source code in src/agents/guardrail.py

@dataclass

class InputGuardrail(Generic[TContext]):

"""Input guardrails are checks that run in parallel to the agent's execution.

They can be used to do things like:

- Check if input messages are off-topic

- Take over control of the agent's execution if an unexpected input is detected

You can use the `@input\_guardrail()` decorator to turn a function into an `InputGuardrail`, or

create an `InputGuardrail` manually.

Guardrails return a `GuardrailResult`. If `result.tripwire\_triggered` is `True`, the agent

execution will immediately stop and a `InputGuardrailTripwireTriggered` exception will be raised

"""

guardrail\_function: Callable[

[RunContextWrapper[TContext], Agent[Any], str | list[TResponseInputItem]],

MaybeAwaitable[GuardrailFunctionOutput],

]

"""A function that receives the agent input and the context, and returns a

`GuardrailResult`. The result marks whether the tripwire was triggered, and can optionally

include information about the guardrail's output.

"""

name: str | None = None

"""The name of the guardrail, used for tracing. If not provided, we'll use the guardrail

function's name.

"""

def get\_name(self) -> str:

if self.name:

return self.name

return self.guardrail\_function.\_\_name\_\_

async def run(

self,

agent: Agent[Any],

input: str | list[TResponseInputItem],

context: RunContextWrapper[TContext],

) -> InputGuardrailResult:

if not callable(self.guardrail\_function):

raise UserError(f"Guardrail function must be callable, got {self.guardrail\_function}")

output = self.guardrail\_function(context, agent, input)

if inspect.isawaitable(output):

return InputGuardrailResult(

guardrail=self,

output=await output,

)

return InputGuardrailResult(

guardrail=self,

output=output,

)

guardrail\_function instance-attribute

guardrail\_function: Callable[

[

RunContextWrapper[TContext],

Agent[Any],

str | list[TResponseInputItem],

],

MaybeAwaitable[GuardrailFunctionOutput],

]

A function that receives the agent input and the context, and returns a GuardrailResult. The result marks whether the tripwire was triggered, and can optionally include information about the guardrail's output.

name class-attribute instance-attribute

name: str | None = None

The name of the guardrail, used for tracing. If not provided, we'll use the guardrail function's name.

OutputGuardrail dataclass

Bases: Generic[TContext]

Output guardrails are checks that run on the final output of an agent. They can be used to do check if the output passes certain validation criteria

You can use the @output\_guardrail() decorator to turn a function into an OutputGuardrail, or create an OutputGuardrail manually.

Guardrails return a GuardrailResult. If result.tripwire\_triggered is True, a OutputGuardrailTripwireTriggered exception will be raised.

Source code in src/agents/guardrail.py

@dataclass

class OutputGuardrail(Generic[TContext]):

"""Output guardrails are checks that run on the final output of an agent.

They can be used to do check if the output passes certain validation criteria

You can use the `@output\_guardrail()` decorator to turn a function into an `OutputGuardrail`,

or create an `OutputGuardrail` manually.

Guardrails return a `GuardrailResult`. If `result.tripwire\_triggered` is `True`, a

`OutputGuardrailTripwireTriggered` exception will be raised.

"""

guardrail\_function: Callable[

[RunContextWrapper[TContext], Agent[Any], Any],

MaybeAwaitable[GuardrailFunctionOutput],

]

"""A function that receives the final agent, its output, and the context, and returns a

`GuardrailResult`. The result marks whether the tripwire was triggered, and can optionally

include information about the guardrail's output.

"""

name: str | None = None

"""The name of the guardrail, used for tracing. If not provided, we'll use the guardrail

function's name.

"""

def get\_name(self) -> str:

if self.name:

return self.name

return self.guardrail\_function.\_\_name\_\_

async def run(

self, context: RunContextWrapper[TContext], agent: Agent[Any], agent\_output: Any

) -> OutputGuardrailResult:

if not callable(self.guardrail\_function):

raise UserError(f"Guardrail function must be callable, got {self.guardrail\_function}")

output = self.guardrail\_function(context, agent, agent\_output)

if inspect.isawaitable(output):

return OutputGuardrailResult(

guardrail=self,

agent=agent,

agent\_output=agent\_output,

output=await output,

)

return OutputGuardrailResult(

guardrail=self,

agent=agent,

agent\_output=agent\_output,

output=output,

)

guardrail\_function instance-attribute

guardrail\_function: Callable[

[RunContextWrapper[TContext], Agent[Any], Any],

MaybeAwaitable[GuardrailFunctionOutput],

]

A function that receives the final agent, its output, and the context, and returns a GuardrailResult. The result marks whether the tripwire was triggered, and can optionally include information about the guardrail's output.

name class-attribute instance-attribute

name: str | None = None

The name of the guardrail, used for tracing. If not provided, we'll use the guardrail function's name.

input\_guardrail

input\_guardrail(

func: \_InputGuardrailFuncSync[TContext\_co],

) -> InputGuardrail[TContext\_co]

input\_guardrail(

func: \_InputGuardrailFuncAsync[TContext\_co],

) -> InputGuardrail[TContext\_co]

input\_guardrail(

\*, name: str | None = None

) -> Callable[

[

\_InputGuardrailFuncSync[TContext\_co]

| \_InputGuardrailFuncAsync[TContext\_co]

],

InputGuardrail[TContext\_co],

]

input\_guardrail(

func: \_InputGuardrailFuncSync[TContext\_co]

| \_InputGuardrailFuncAsync[TContext\_co]

| None = None,

\*,

name: str | None = None,

) -> (

InputGuardrail[TContext\_co]

| Callable[

[

\_InputGuardrailFuncSync[TContext\_co]

| \_InputGuardrailFuncAsync[TContext\_co]

],

InputGuardrail[TContext\_co],

]

)

Decorator that transforms a sync or async function into an InputGuardrail. It can be used directly (no parentheses) or with keyword args, e.g.:

@input\_guardrail

def my\_sync\_guardrail(...): ...

@input\_guardrail(name="guardrail\_name")

async def my\_async\_guardrail(...): ...

Source code in src/agents/guardrail.py

def input\_guardrail(

func: \_InputGuardrailFuncSync[TContext\_co]

| \_InputGuardrailFuncAsync[TContext\_co]

| None = None,

\*,

name: str | None = None,

) -> (

InputGuardrail[TContext\_co]

| Callable[

[\_InputGuardrailFuncSync[TContext\_co] | \_InputGuardrailFuncAsync[TContext\_co]],

InputGuardrail[TContext\_co],

]

):

"""

Decorator that transforms a sync or async function into an `InputGuardrail`.

It can be used directly (no parentheses) or with keyword args, e.g.:

@input\_guardrail

def my\_sync\_guardrail(...): ...

@input\_guardrail(name="guardrail\_name")

async def my\_async\_guardrail(...): ...

"""

def decorator(

f: \_InputGuardrailFuncSync[TContext\_co] | \_InputGuardrailFuncAsync[TContext\_co],

) -> InputGuardrail[TContext\_co]:

return InputGuardrail(guardrail\_function=f, name=name)

if func is not None:

# Decorator was used without parentheses

return decorator(func)

# Decorator used with keyword arguments

return decorator

output\_guardrail

output\_guardrail(

func: \_OutputGuardrailFuncSync[TContext\_co],

) -> OutputGuardrail[TContext\_co]

output\_guardrail(

func: \_OutputGuardrailFuncAsync[TContext\_co],

) -> OutputGuardrail[TContext\_co]

output\_guardrail(

\*, name: str | None = None

) -> Callable[

[

\_OutputGuardrailFuncSync[TContext\_co]

| \_OutputGuardrailFuncAsync[TContext\_co]

],

OutputGuardrail[TContext\_co],

]

output\_guardrail(

func: \_OutputGuardrailFuncSync[TContext\_co]

| \_OutputGuardrailFuncAsync[TContext\_co]

| None = None,

\*,

name: str | None = None,

) -> (

OutputGuardrail[TContext\_co]

| Callable[

[

\_OutputGuardrailFuncSync[TContext\_co]

| \_OutputGuardrailFuncAsync[TContext\_co]

],

OutputGuardrail[TContext\_co],

]

)

Decorator that transforms a sync or async function into an OutputGuardrail. It can be used directly (no parentheses) or with keyword args, e.g.:

@output\_guardrail

def my\_sync\_guardrail(...): ...

@output\_guardrail(name="guardrail\_name")

async def my\_async\_guardrail(...): ...

Source code in src/agents/guardrail.py

def output\_guardrail(

func: \_OutputGuardrailFuncSync[TContext\_co]

| \_OutputGuardrailFuncAsync[TContext\_co]

| None = None,

\*,

name: str | None = None,

) -> (

OutputGuardrail[TContext\_co]

| Callable[

[\_OutputGuardrailFuncSync[TContext\_co] | \_OutputGuardrailFuncAsync[TContext\_co]],

OutputGuardrail[TContext\_co],

]

):

"""

Decorator that transforms a sync or async function into an `OutputGuardrail`.

It can be used directly (no parentheses) or with keyword args, e.g.:

@output\_guardrail

def my\_sync\_guardrail(...): ...

@output\_guardrail(name="guardrail\_name")

async def my\_async\_guardrail(...): ...

"""

def decorator(

f: \_OutputGuardrailFuncSync[TContext\_co] | \_OutputGuardrailFuncAsync[TContext\_co],

) -> OutputGuardrail[TContext\_co]:

return OutputGuardrail(guardrail\_function=f, name=name)

if func is not None:

# Decorator was used without parentheses

return decorator(func)

# Decorator used with keyword arguments

return decorator

Model settings

ModelSettings dataclass

Settings to use when calling an LLM.

This class holds optional model configuration parameters (e.g. temperature, top\_p, penalties, truncation, etc.).

Not all models/providers support all of these parameters, so please check the API documentation for the specific model and provider you are using.

Source code in src/agents/model\_settings.py

@dataclass

class ModelSettings:

"""Settings to use when calling an LLM.

This class holds optional model configuration parameters (e.g. temperature,

top\_p, penalties, truncation, etc.).

Not all models/providers support all of these parameters, so please check the API documentation

for the specific model and provider you are using.

"""

temperature: float | None = None

"""The temperature to use when calling the model."""

top\_p: float | None = None

"""The top\_p to use when calling the model."""

frequency\_penalty: float | None = None

"""The frequency penalty to use when calling the model."""

presence\_penalty: float | None = None

"""The presence penalty to use when calling the model."""

tool\_choice: Literal["auto", "required", "none"] | str | None = None

"""The tool choice to use when calling the model."""

parallel\_tool\_calls: bool | None = None

"""Whether to use parallel tool calls when calling the model.

Defaults to False if not provided."""

truncation: Literal["auto", "disabled"] | None = None

"""The truncation strategy to use when calling the model."""

max\_tokens: int | None = None

"""The maximum number of output tokens to generate."""

store: bool | None = None

"""Whether to store the generated model response for later retrieval.

Defaults to True if not provided."""

def resolve(self, override: ModelSettings | None) -> ModelSettings:

"""Produce a new ModelSettings by overlaying any non-None values from the

override on top of this instance."""

if override is None:

return self

changes = {

field.name: getattr(override, field.name)

for field in fields(self)

if getattr(override, field.name) is not None

}

return replace(self, \*\*changes)

temperature class-attribute instance-attribute

temperature: float | None = None

The temperature to use when calling the model.

top\_p class-attribute instance-attribute

top\_p: float | None = None

The top\_p to use when calling the model.

frequency\_penalty class-attribute instance-attribute

frequency\_penalty: float | None = None

The frequency penalty to use when calling the model.

presence\_penalty class-attribute instance-attribute

presence\_penalty: float | None = None

The presence penalty to use when calling the model.

tool\_choice class-attribute instance-attribute

tool\_choice: (

Literal["auto", "required", "none"] | str | None

) = None

The tool choice to use when calling the model.

parallel\_tool\_calls class-attribute instance-attribute

parallel\_tool\_calls: bool | None = None

Whether to use parallel tool calls when calling the model. Defaults to False if not provided.

truncation class-attribute instance-attribute

truncation: Literal['auto', 'disabled'] | None = None

The truncation strategy to use when calling the model.

max\_tokens class-attribute instance-attribute

max\_tokens: int | None = None

The maximum number of output tokens to generate.

store class-attribute instance-attribute

store: bool | None = None

Whether to store the generated model response for later retrieval. Defaults to True if not provided.

resolve

resolve(override: ModelSettings | None) -> ModelSettings

Produce a new ModelSettings by overlaying any non-None values from the override on top of this instance.

Source code in src/agents/model\_settings.py

def resolve(self, override: ModelSettings | None) -> ModelSettings:

"""Produce a new ModelSettings by overlaying any non-None values from the

override on top of this instance."""

if override is None:

return self

changes = {

field.name: getattr(override, field.name)

for field in fields(self)

if getattr(override, field.name) is not None

}

return replace(self, \*\*changes)

Agent output

AgentOutputSchema dataclass

An object that captures the JSON schema of the output, as well as validating/parsing JSON produced by the LLM into the output type.

Source code in src/agents/agent\_output.py

@dataclass(init=False)

class AgentOutputSchema:

"""An object that captures the JSON schema of the output, as well as validating/parsing JSON

produced by the LLM into the output type.

"""

output\_type: type[Any]

"""The type of the output."""

\_type\_adapter: TypeAdapter[Any]

"""A type adapter that wraps the output type, so that we can validate JSON."""

\_is\_wrapped: bool

"""Whether the output type is wrapped in a dictionary. This is generally done if the base

output type cannot be represented as a JSON Schema object.

"""

\_output\_schema: dict[str, Any]

"""The JSON schema of the output."""

strict\_json\_schema: bool

"""Whether the JSON schema is in strict mode. We \*\*strongly\*\* recommend setting this to True,

as it increases the likelihood of correct JSON input.

"""

def \_\_init\_\_(self, output\_type: type[Any], strict\_json\_schema: bool = True):

"""

Args:

output\_type: The type of the output.

strict\_json\_schema: Whether the JSON schema is in strict mode. We \*\*strongly\*\* recommend

setting this to True, as it increases the likelihood of correct JSON input.

"""

self.output\_type = output\_type

self.strict\_json\_schema = strict\_json\_schema

if output\_type is None or output\_type is str:

self.\_is\_wrapped = False

self.\_type\_adapter = TypeAdapter(output\_type)

self.\_output\_schema = self.\_type\_adapter.json\_schema()

return

# We should wrap for things that are not plain text, and for things that would definitely

# not be a JSON Schema object.

self.\_is\_wrapped = not \_is\_subclass\_of\_base\_model\_or\_dict(output\_type)

if self.\_is\_wrapped:

OutputType = TypedDict(

"OutputType",

{

\_WRAPPER\_DICT\_KEY: output\_type, # type: ignore

},

)

self.\_type\_adapter = TypeAdapter(OutputType)

self.\_output\_schema = self.\_type\_adapter.json\_schema()

else:

self.\_type\_adapter = TypeAdapter(output\_type)

self.\_output\_schema = self.\_type\_adapter.json\_schema()

if self.strict\_json\_schema:

self.\_output\_schema = ensure\_strict\_json\_schema(self.\_output\_schema)

def is\_plain\_text(self) -> bool:

"""Whether the output type is plain text (versus a JSON object)."""

return self.output\_type is None or self.output\_type is str

def json\_schema(self) -> dict[str, Any]:

"""The JSON schema of the output type."""

if self.is\_plain\_text():

raise UserError("Output type is plain text, so no JSON schema is available")

return self.\_output\_schema

def validate\_json(self, json\_str: str, partial: bool = False) -> Any:

"""Validate a JSON string against the output type. Returns the validated object, or raises

a `ModelBehaviorError` if the JSON is invalid.

"""

validated = \_json.validate\_json(json\_str, self.\_type\_adapter, partial)

if self.\_is\_wrapped:

if not isinstance(validated, dict):

\_error\_tracing.attach\_error\_to\_current\_span(

SpanError(

message="Invalid JSON",

data={"details": f"Expected a dict, got {type(validated)}"},

)

)

raise ModelBehaviorError(

f"Expected a dict, got {type(validated)} for JSON: {json\_str}"

)

if \_WRAPPER\_DICT\_KEY not in validated:

\_error\_tracing.attach\_error\_to\_current\_span(

SpanError(

message="Invalid JSON",

data={"details": f"Could not find key {\_WRAPPER\_DICT\_KEY} in JSON"},

)

)

raise ModelBehaviorError(

f"Could not find key {\_WRAPPER\_DICT\_KEY} in JSON: {json\_str}"

)

return validated[\_WRAPPER\_DICT\_KEY]

return validated

def output\_type\_name(self) -> str:

"""The name of the output type."""

return \_type\_to\_str(self.output\_type)

output\_type instance-attribute

output\_type: type[Any] = output\_type

The type of the output.

strict\_json\_schema instance-attribute

strict\_json\_schema: bool = strict\_json\_schema

Whether the JSON schema is in strict mode. We strongly recommend setting this to True, as it increases the likelihood of correct JSON input.

\_\_init\_\_

\_\_init\_\_(

output\_type: type[Any], strict\_json\_schema: bool = True

)

Parameters:

Name Type Description Default

output\_type type[Any] The type of the output. required

strict\_json\_schema bool Whether the JSON schema is in strict mode. We strongly recommend setting this to True, as it increases the likelihood of correct JSON input. True

Source code in src/agents/agent\_output.py

def \_\_init\_\_(self, output\_type: type[Any], strict\_json\_schema: bool = True):

"""

Args:

output\_type: The type of the output.

strict\_json\_schema: Whether the JSON schema is in strict mode. We \*\*strongly\*\* recommend

setting this to True, as it increases the likelihood of correct JSON input.

"""

self.output\_type = output\_type

self.strict\_json\_schema = strict\_json\_schema

if output\_type is None or output\_type is str:

self.\_is\_wrapped = False

self.\_type\_adapter = TypeAdapter(output\_type)

self.\_output\_schema = self.\_type\_adapter.json\_schema()

return

# We should wrap for things that are not plain text, and for things that would definitely

# not be a JSON Schema object.

self.\_is\_wrapped = not \_is\_subclass\_of\_base\_model\_or\_dict(output\_type)

if self.\_is\_wrapped:

OutputType = TypedDict(

"OutputType",

{

\_WRAPPER\_DICT\_KEY: output\_type, # type: ignore

},

)

self.\_type\_adapter = TypeAdapter(OutputType)

self.\_output\_schema = self.\_type\_adapter.json\_schema()

else:

self.\_type\_adapter = TypeAdapter(output\_type)

self.\_output\_schema = self.\_type\_adapter.json\_schema()

if self.strict\_json\_schema:

self.\_output\_schema = ensure\_strict\_json\_schema(self.\_output\_schema)

is\_plain\_text

is\_plain\_text() -> bool

Whether the output type is plain text (versus a JSON object).

Source code in src/agents/agent\_output.py

def is\_plain\_text(self) -> bool:

"""Whether the output type is plain text (versus a JSON object)."""

return self.output\_type is None or self.output\_type is str

json\_schema

json\_schema() -> dict[str, Any]

The JSON schema of the output type.

Source code in src/agents/agent\_output.py

def json\_schema(self) -> dict[str, Any]:

"""The JSON schema of the output type."""

if self.is\_plain\_text():

raise UserError("Output type is plain text, so no JSON schema is available")

return self.\_output\_schema

validate\_json

validate\_json(json\_str: str, partial: bool = False) -> Any

Validate a JSON string against the output type. Returns the validated object, or raises a ModelBehaviorError if the JSON is invalid.

Source code in src/agents/agent\_output.py

def validate\_json(self, json\_str: str, partial: bool = False) -> Any:

"""Validate a JSON string against the output type. Returns the validated object, or raises

a `ModelBehaviorError` if the JSON is invalid.

"""

validated = \_json.validate\_json(json\_str, self.\_type\_adapter, partial)

if self.\_is\_wrapped:

if not isinstance(validated, dict):

\_error\_tracing.attach\_error\_to\_current\_span(

SpanError(

message="Invalid JSON",

data={"details": f"Expected a dict, got {type(validated)}"},

)

)

raise ModelBehaviorError(

f"Expected a dict, got {type(validated)} for JSON: {json\_str}"

)

if \_WRAPPER\_DICT\_KEY not in validated:

\_error\_tracing.attach\_error\_to\_current\_span(

SpanError(

message="Invalid JSON",

data={"details": f"Could not find key {\_WRAPPER\_DICT\_KEY} in JSON"},

)

)

raise ModelBehaviorError(

f"Could not find key {\_WRAPPER\_DICT\_KEY} in JSON: {json\_str}"

)

return validated[\_WRAPPER\_DICT\_KEY]

return validated

output\_type\_name

output\_type\_name() -> str

The name of the output type.

Source code in src/agents/agent\_output.py

def output\_type\_name(self) -> str:

"""The name of the output type."""

return \_type\_to\_str(self.output\_type)

Function schema

FuncSchema dataclass

Captures the schema for a python function, in preparation for sending it to an LLM as a tool.

Source code in src/agents/function\_schema.py

@dataclass

class FuncSchema:

"""

Captures the schema for a python function, in preparation for sending it to an LLM as a tool.

"""

name: str

"""The name of the function."""

description: str | None

"""The description of the function."""

params\_pydantic\_model: type[BaseModel]

"""A Pydantic model that represents the function's parameters."""

params\_json\_schema: dict[str, Any]

"""The JSON schema for the function's parameters, derived from the Pydantic model."""

signature: inspect.Signature

"""The signature of the function."""

takes\_context: bool = False

"""Whether the function takes a RunContextWrapper argument (must be the first argument)."""

strict\_json\_schema: bool = True

"""Whether the JSON schema is in strict mode. We \*\*strongly\*\* recommend setting this to True,

as it increases the likelihood of correct JSON input."""

def to\_call\_args(self, data: BaseModel) -> tuple[list[Any], dict[str, Any]]:

"""

Converts validated data from the Pydantic model into (args, kwargs), suitable for calling

the original function.

"""

positional\_args: list[Any] = []

keyword\_args: dict[str, Any] = {}

seen\_var\_positional = False

# Use enumerate() so we can skip the first parameter if it's context.

for idx, (name, param) in enumerate(self.signature.parameters.items()):

# If the function takes a RunContextWrapper and this is the first parameter, skip it.

if self.takes\_context and idx == 0:

continue

value = getattr(data, name, None)

if param.kind == param.VAR\_POSITIONAL:

# e.g. \*args: extend positional args and mark that \*args is now seen

positional\_args.extend(value or [])

seen\_var\_positional = True

elif param.kind == param.VAR\_KEYWORD:

# e.g. \*\*kwargs handling

keyword\_args.update(value or {})

elif param.kind in (param.POSITIONAL\_ONLY, param.POSITIONAL\_OR\_KEYWORD):

# Before \*args, add to positional args. After \*args, add to keyword args.

if not seen\_var\_positional:

positional\_args.append(value)

else:

keyword\_args[name] = value

else:

# For KEYWORD\_ONLY parameters, always use keyword args.

keyword\_args[name] = value

return positional\_args, keyword\_args

name instance-attribute

name: str

The name of the function.

description instance-attribute

description: str | None

The description of the function.

params\_pydantic\_model instance-attribute

params\_pydantic\_model: type[BaseModel]

A Pydantic model that represents the function's parameters.

params\_json\_schema instance-attribute

params\_json\_schema: dict[str, Any]

The JSON schema for the function's parameters, derived from the Pydantic model.

signature instance-attribute

signature: Signature

The signature of the function.

takes\_context class-attribute instance-attribute

takes\_context: bool = False

Whether the function takes a RunContextWrapper argument (must be the first argument).

strict\_json\_schema class-attribute instance-attribute

strict\_json\_schema: bool = True

Whether the JSON schema is in strict mode. We strongly recommend setting this to True, as it increases the likelihood of correct JSON input.

to\_call\_args

to\_call\_args(

data: BaseModel,

) -> tuple[list[Any], dict[str, Any]]

Converts validated data from the Pydantic model into (args, kwargs), suitable for calling the original function.

Source code in src/agents/function\_schema.py

def to\_call\_args(self, data: BaseModel) -> tuple[list[Any], dict[str, Any]]:

"""

Converts validated data from the Pydantic model into (args, kwargs), suitable for calling

the original function.

"""

positional\_args: list[Any] = []

keyword\_args: dict[str, Any] = {}

seen\_var\_positional = False

# Use enumerate() so we can skip the first parameter if it's context.

for idx, (name, param) in enumerate(self.signature.parameters.items()):

# If the function takes a RunContextWrapper and this is the first parameter, skip it.

if self.takes\_context and idx == 0:

continue

value = getattr(data, name, None)

if param.kind == param.VAR\_POSITIONAL:

# e.g. \*args: extend positional args and mark that \*args is now seen

positional\_args.extend(value or [])

seen\_var\_positional = True

elif param.kind == param.VAR\_KEYWORD:

# e.g. \*\*kwargs handling

keyword\_args.update(value or {})

elif param.kind in (param.POSITIONAL\_ONLY, param.POSITIONAL\_OR\_KEYWORD):

# Before \*args, add to positional args. After \*args, add to keyword args.

if not seen\_var\_positional:

positional\_args.append(value)

else:

keyword\_args[name] = value

else:

# For KEYWORD\_ONLY parameters, always use keyword args.

keyword\_args[name] = value

return positional\_args, keyword\_args

FuncDocumentation dataclass

Contains metadata about a python function, extracted from its docstring.

Source code in src/agents/function\_schema.py

@dataclass

class FuncDocumentation:

"""Contains metadata about a python function, extracted from its docstring."""

name: str

"""The name of the function, via `\_\_name\_\_`."""

description: str | None

"""The description of the function, derived from the docstring."""

param\_descriptions: dict[str, str] | None

"""The parameter descriptions of the function, derived from the docstring."""

name instance-attribute

name: str

The name of the function, via \_\_name\_\_.

description instance-attribute

description: str | None

The description of the function, derived from the docstring.

param\_descriptions instance-attribute

param\_descriptions: dict[str, str] | None

The parameter descriptions of the function, derived from the docstring.

generate\_func\_documentation

generate\_func\_documentation(

func: Callable[..., Any],

style: DocstringStyle | None = None,

) -> FuncDocumentation

Extracts metadata from a function docstring, in preparation for sending it to an LLM as a tool.

Parameters:

Name Type Description Default

func Callable[..., Any] The function to extract documentation from. required

style DocstringStyle | None The style of the docstring to use for parsing. If not provided, we will attempt to auto-detect the style. None

Returns:

Type Description

FuncDocumentation A FuncDocumentation object containing the function's name, description, and parameter

FuncDocumentation descriptions.

Source code in src/agents/function\_schema.py

def generate\_func\_documentation(

func: Callable[..., Any], style: DocstringStyle | None = None

) -> FuncDocumentation:

"""

Extracts metadata from a function docstring, in preparation for sending it to an LLM as a tool.

Args:

func: The function to extract documentation from.

style: The style of the docstring to use for parsing. If not provided, we will attempt to

auto-detect the style.

Returns:

A FuncDocumentation object containing the function's name, description, and parameter

descriptions.

"""

name = func.\_\_name\_\_

doc = inspect.getdoc(func)

if not doc:

return FuncDocumentation(name=name, description=None, param\_descriptions=None)

with \_suppress\_griffe\_logging():

docstring = Docstring(doc, lineno=1, parser=style or \_detect\_docstring\_style(doc))

parsed = docstring.parse()

description: str | None = next(

(section.value for section in parsed if section.kind == DocstringSectionKind.text), None

)

param\_descriptions: dict[str, str] = {

param.name: param.description

for section in parsed

if section.kind == DocstringSectionKind.parameters

for param in section.value

}

return FuncDocumentation(

name=func.\_\_name\_\_,

description=description,

param\_descriptions=param\_descriptions or None,

)

function\_schema

function\_schema(

func: Callable[..., Any],

docstring\_style: DocstringStyle | None = None,

name\_override: str | None = None,

description\_override: str | None = None,

use\_docstring\_info: bool = True,

strict\_json\_schema: bool = True,

) -> FuncSchema

Given a python function, extracts a FuncSchema from it, capturing the name, description, parameter descriptions, and other metadata.

Parameters:

Name Type Description Default

func Callable[..., Any] The function to extract the schema from. required

docstring\_style DocstringStyle | None The style of the docstring to use for parsing. If not provided, we will attempt to auto-detect the style. None

name\_override str | None If provided, use this name instead of the function's \_\_name\_\_. None

description\_override str | None If provided, use this description instead of the one derived from the docstring. None

use\_docstring\_info bool If True, uses the docstring to generate the description and parameter descriptions. True

strict\_json\_schema bool Whether the JSON schema is in strict mode. If True, we'll ensure that the schema adheres to the "strict" standard the OpenAI API expects. We strongly recommend setting this to True, as it increases the likelihood of the LLM providing correct JSON input. True

Returns:

Type Description

FuncSchema A FuncSchema object containing the function's name, description, parameter descriptions,

FuncSchema and other metadata.

Source code in src/agents/function\_schema.py

def function\_schema(

func: Callable[..., Any],

docstring\_style: DocstringStyle | None = None,

name\_override: str | None = None,

description\_override: str | None = None,

use\_docstring\_info: bool = True,

strict\_json\_schema: bool = True,

) -> FuncSchema:

"""

Given a python function, extracts a `FuncSchema` from it, capturing the name, description,

parameter descriptions, and other metadata.

Args:

func: The function to extract the schema from.

docstring\_style: The style of the docstring to use for parsing. If not provided, we will

attempt to auto-detect the style.

name\_override: If provided, use this name instead of the function's `\_\_name\_\_`.

description\_override: If provided, use this description instead of the one derived from the

docstring.

use\_docstring\_info: If True, uses the docstring to generate the description and parameter

descriptions.

strict\_json\_schema: Whether the JSON schema is in strict mode. If True, we'll ensure that

the schema adheres to the "strict" standard the OpenAI API expects. We \*\*strongly\*\*

recommend setting this to True, as it increases the likelihood of the LLM providing

correct JSON input.

Returns:

A `FuncSchema` object containing the function's name, description, parameter descriptions,

and other metadata.

"""

# 1. Grab docstring info

if use\_docstring\_info:

doc\_info = generate\_func\_documentation(func, docstring\_style)

param\_descs = doc\_info.param\_descriptions or {}

else:

doc\_info = None

param\_descs = {}

func\_name = name\_override or doc\_info.name if doc\_info else func.\_\_name\_\_

# 2. Inspect function signature and get type hints

sig = inspect.signature(func)

type\_hints = get\_type\_hints(func)

params = list(sig.parameters.items())

takes\_context = False

filtered\_params = []

if params:

first\_name, first\_param = params[0]

# Prefer the evaluated type hint if available

ann = type\_hints.get(first\_name, first\_param.annotation)

if ann != inspect.\_empty:

origin = get\_origin(ann) or ann

if origin is RunContextWrapper:

takes\_context = True # Mark that the function takes context

else:

filtered\_params.append((first\_name, first\_param))

else:

filtered\_params.append((first\_name, first\_param))

# For parameters other than the first, raise error if any use RunContextWrapper.

for name, param in params[1:]:

ann = type\_hints.get(name, param.annotation)

if ann != inspect.\_empty:

origin = get\_origin(ann) or ann

if origin is RunContextWrapper:

raise UserError(

f"RunContextWrapper param found at non-first position in function"

f" {func.\_\_name\_\_}"

)

filtered\_params.append((name, param))

# We will collect field definitions for create\_model as a dict:

# field\_name -> (type\_annotation, default\_value\_or\_Field(...))

fields: dict[str, Any] = {}

for name, param in filtered\_params:

ann = type\_hints.get(name, param.annotation)

default = param.default

# If there's no type hint, assume `Any`

if ann == inspect.\_empty:

ann = Any

# If a docstring param description exists, use it

field\_description = param\_descs.get(name, None)

# Handle different parameter kinds

if param.kind == param.VAR\_POSITIONAL:

# e.g. \*args: extend positional args

if get\_origin(ann) is tuple:

# e.g. def foo(\*args: tuple[int, ...]) -> treat as List[int]

args\_of\_tuple = get\_args(ann)

if len(args\_of\_tuple) == 2 and args\_of\_tuple[1] is Ellipsis:

ann = list[args\_of\_tuple[0]] # type: ignore

else:

ann = list[Any]

else:

# If user wrote \*args: int, treat as List[int]

ann = list[ann] # type: ignore

# Default factory to empty list

fields[name] = (

ann,

Field(default\_factory=list, description=field\_description), # type: ignore

)

elif param.kind == param.VAR\_KEYWORD:

# \*\*kwargs handling

if get\_origin(ann) is dict:

# e.g. def foo(\*\*kwargs: dict[str, int])

dict\_args = get\_args(ann)

if len(dict\_args) == 2:

ann = dict[dict\_args[0], dict\_args[1]] # type: ignore

else:

ann = dict[str, Any]

else:

# e.g. def foo(\*\*kwargs: int) -> Dict[str, int]

ann = dict[str, ann] # type: ignore

fields[name] = (

ann,

Field(default\_factory=dict, description=field\_description), # type: ignore

)

else:

# Normal parameter

if default == inspect.\_empty:

# Required field

fields[name] = (

ann,

Field(..., description=field\_description),

)

else:

# Parameter with a default value

fields[name] = (

ann,

Field(default=default, description=field\_description),

)

# 3. Dynamically build a Pydantic model

dynamic\_model = create\_model(f"{func\_name}\_args", \_\_base\_\_=BaseModel, \*\*fields)

# 4. Build JSON schema from that model

json\_schema = dynamic\_model.model\_json\_schema()

if strict\_json\_schema:

json\_schema = ensure\_strict\_json\_schema(json\_schema)

# 5. Return as a FuncSchema dataclass

return FuncSchema(

name=func\_name,

description=description\_override or doc\_info.description if doc\_info else None,

params\_pydantic\_model=dynamic\_model,

params\_json\_schema=json\_schema,

signature=sig,

takes\_context=takes\_context,

strict\_json\_schema=strict\_json\_schema,

)

Model interface

ModelTracing

Bases: Enum

Source code in src/agents/models/interface.py

class ModelTracing(enum.Enum):

DISABLED = 0

"""Tracing is disabled entirely."""

ENABLED = 1

"""Tracing is enabled, and all data is included."""

ENABLED\_WITHOUT\_DATA = 2

"""Tracing is enabled, but inputs/outputs are not included."""

def is\_disabled(self) -> bool:

return self == ModelTracing.DISABLED

def include\_data(self) -> bool:

return self == ModelTracing.ENABLED

DISABLED class-attribute instance-attribute

DISABLED = 0

Tracing is disabled entirely.

ENABLED class-attribute instance-attribute

ENABLED = 1

Tracing is enabled, and all data is included.

ENABLED\_WITHOUT\_DATA class-attribute instance-attribute

ENABLED\_WITHOUT\_DATA = 2

Tracing is enabled, but inputs/outputs are not included.

Model

Bases: ABC

The base interface for calling an LLM.

Source code in src/agents/models/interface.py

class Model(abc.ABC):

"""The base interface for calling an LLM."""

@abc.abstractmethod

async def get\_response(

self,

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

tracing: ModelTracing,

) -> ModelResponse:

"""Get a response from the model.

Args:

system\_instructions: The system instructions to use.

input: The input items to the model, in OpenAI Responses format.

model\_settings: The model settings to use.

tools: The tools available to the model.

output\_schema: The output schema to use.

handoffs: The handoffs available to the model.

tracing: Tracing configuration.

Returns:

The full model response.

"""

pass

@abc.abstractmethod

def stream\_response(

self,

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

tracing: ModelTracing,

) -> AsyncIterator[TResponseStreamEvent]:

"""Stream a response from the model.

Args:

system\_instructions: The system instructions to use.

input: The input items to the model, in OpenAI Responses format.

model\_settings: The model settings to use.

tools: The tools available to the model.

output\_schema: The output schema to use.

handoffs: The handoffs available to the model.

tracing: Tracing configuration.

Returns:

An iterator of response stream events, in OpenAI Responses format.

"""

pass

get\_response abstractmethod async

get\_response(

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

tracing: ModelTracing,

) -> ModelResponse

Get a response from the model.

Parameters:

Name Type Description Default

system\_instructions str | None The system instructions to use. required

input str | list[TResponseInputItem] The input items to the model, in OpenAI Responses format. required

model\_settings ModelSettings The model settings to use. required

tools list[Tool] The tools available to the model. required

output\_schema AgentOutputSchema | None The output schema to use. required

handoffs list[Handoff] The handoffs available to the model. required

tracing ModelTracing Tracing configuration. required

Returns:

Type Description

ModelResponse The full model response.

Source code in src/agents/models/interface.py

@abc.abstractmethod

async def get\_response(

self,

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

tracing: ModelTracing,

) -> ModelResponse:

"""Get a response from the model.

Args:

system\_instructions: The system instructions to use.

input: The input items to the model, in OpenAI Responses format.

model\_settings: The model settings to use.

tools: The tools available to the model.

output\_schema: The output schema to use.

handoffs: The handoffs available to the model.

tracing: Tracing configuration.

Returns:

The full model response.

"""

pass

stream\_response abstractmethod

stream\_response(

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

tracing: ModelTracing,

) -> AsyncIterator[TResponseStreamEvent]

Stream a response from the model.

Parameters:

Name Type Description Default

system\_instructions str | None The system instructions to use. required

input str | list[TResponseInputItem] The input items to the model, in OpenAI Responses format. required

model\_settings ModelSettings The model settings to use. required

tools list[Tool] The tools available to the model. required

output\_schema AgentOutputSchema | None The output schema to use. required

handoffs list[Handoff] The handoffs available to the model. required

tracing ModelTracing Tracing configuration. required

Returns:

Type Description

AsyncIterator[TResponseStreamEvent] An iterator of response stream events, in OpenAI Responses format.

Source code in src/agents/models/interface.py

@abc.abstractmethod

def stream\_response(

self,

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

tracing: ModelTracing,

) -> AsyncIterator[TResponseStreamEvent]:

"""Stream a response from the model.

Args:

system\_instructions: The system instructions to use.

input: The input items to the model, in OpenAI Responses format.

model\_settings: The model settings to use.

tools: The tools available to the model.

output\_schema: The output schema to use.

handoffs: The handoffs available to the model.

tracing: Tracing configuration.

Returns:

An iterator of response stream events, in OpenAI Responses format.

"""

pass

ModelProvider

Bases: ABC

The base interface for a model provider.

Model provider is responsible for looking up Models by name.

Source code in src/agents/models/interface.py

class ModelProvider(abc.ABC):

"""The base interface for a model provider.

Model provider is responsible for looking up Models by name.

"""

@abc.abstractmethod

def get\_model(self, model\_name: str | None) -> Model:

"""Get a model by name.

Args:

model\_name: The name of the model to get.

Returns:

The model.

"""

get\_model abstractmethod

get\_model(model\_name: str | None) -> Model

Get a model by name.

Parameters:

Name Type Description Default

model\_name str | None The name of the model to get. required

Returns:

Type Description

Model The model.

Source code in src/agents/models/interface.py

@abc.abstractmethod

def get\_model(self, model\_name: str | None) -> Model:

"""Get a model by name.

Args:

model\_name: The name of the model to get.

Returns:

The model.

"""

OpenAI Chat Completions model

OpenAIChatCompletionsModel

Bases: Model

Source code in src/agents/models/openai\_chatcompletions.py

class OpenAIChatCompletionsModel(Model):

def \_\_init\_\_(

self,

model: str | ChatModel,

openai\_client: AsyncOpenAI,

) -> None:

self.model = model

self.\_client = openai\_client

def \_non\_null\_or\_not\_given(self, value: Any) -> Any:

return value if value is not None else NOT\_GIVEN

async def get\_response(

self,

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

tracing: ModelTracing,

) -> ModelResponse:

with generation\_span(

model=str(self.model),

model\_config=dataclasses.asdict(model\_settings)

| {"base\_url": str(self.\_client.base\_url)},

disabled=tracing.is\_disabled(),

) as span\_generation:

response = await self.\_fetch\_response(

system\_instructions,

input,

model\_settings,

tools,

output\_schema,

handoffs,

span\_generation,

tracing,

stream=False,

)

if \_debug.DONT\_LOG\_MODEL\_DATA:

logger.debug("Received model response")

else:

logger.debug(

f"LLM resp:\n{json.dumps(response.choices[0].message.model\_dump(), indent=2)}\n"

)

usage = (

Usage(

requests=1,

input\_tokens=response.usage.prompt\_tokens,

output\_tokens=response.usage.completion\_tokens,

total\_tokens=response.usage.total\_tokens,

)

if response.usage

else Usage()

)

if tracing.include\_data():

span\_generation.span\_data.output = [response.choices[0].message.model\_dump()]

span\_generation.span\_data.usage = {

"input\_tokens": usage.input\_tokens,

"output\_tokens": usage.output\_tokens,

}

items = \_Converter.message\_to\_output\_items(response.choices[0].message)

return ModelResponse(

output=items,

usage=usage,

referenceable\_id=None,

)

async def stream\_response(

self,

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

tracing: ModelTracing,

) -> AsyncIterator[TResponseStreamEvent]:

"""

Yields a partial message as it is generated, as well as the usage information.

"""

with generation\_span(

model=str(self.model),

model\_config=dataclasses.asdict(model\_settings)

| {"base\_url": str(self.\_client.base\_url)},

disabled=tracing.is\_disabled(),

) as span\_generation:

response, stream = await self.\_fetch\_response(

system\_instructions,

input,

model\_settings,

tools,

output\_schema,

handoffs,

span\_generation,

tracing,

stream=True,

)

usage: CompletionUsage | None = None

state = \_StreamingState()

async for chunk in stream:

if not state.started:

state.started = True

yield ResponseCreatedEvent(

response=response,

type="response.created",

)

# The usage is only available in the last chunk

usage = chunk.usage

if not chunk.choices or not chunk.choices[0].delta:

continue

delta = chunk.choices[0].delta

# Handle text

if delta.content:

if not state.text\_content\_index\_and\_output:

# Initialize a content tracker for streaming text

state.text\_content\_index\_and\_output = (

0 if not state.refusal\_content\_index\_and\_output else 1,

ResponseOutputText(

text="",

type="output\_text",

annotations=[],

),

)

# Start a new assistant message stream

assistant\_item = ResponseOutputMessage(

id=FAKE\_RESPONSES\_ID,

content=[],

role="assistant",

type="message",

status="in\_progress",

)

# Notify consumers of the start of a new output message + first content part

yield ResponseOutputItemAddedEvent(

item=assistant\_item,

output\_index=0,

type="response.output\_item.added",

)

yield ResponseContentPartAddedEvent(

content\_index=state.text\_content\_index\_and\_output[0],

item\_id=FAKE\_RESPONSES\_ID,

output\_index=0,

part=ResponseOutputText(

text="",

type="output\_text",

annotations=[],

),

type="response.content\_part.added",

)

# Emit the delta for this segment of content

yield ResponseTextDeltaEvent(

content\_index=state.text\_content\_index\_and\_output[0],

delta=delta.content,

item\_id=FAKE\_RESPONSES\_ID,

output\_index=0,

type="response.output\_text.delta",

)

# Accumulate the text into the response part

state.text\_content\_index\_and\_output[1].text += delta.content

# Handle refusals (model declines to answer)

if delta.refusal:

if not state.refusal\_content\_index\_and\_output:

# Initialize a content tracker for streaming refusal text

state.refusal\_content\_index\_and\_output = (

0 if not state.text\_content\_index\_and\_output else 1,

ResponseOutputRefusal(refusal="", type="refusal"),

)

# Start a new assistant message if one doesn't exist yet (in-progress)

assistant\_item = ResponseOutputMessage(

id=FAKE\_RESPONSES\_ID,

content=[],

role="assistant",

type="message",

status="in\_progress",

)

# Notify downstream that assistant message + first content part are starting

yield ResponseOutputItemAddedEvent(

item=assistant\_item,

output\_index=0,

type="response.output\_item.added",

)

yield ResponseContentPartAddedEvent(

content\_index=state.refusal\_content\_index\_and\_output[0],

item\_id=FAKE\_RESPONSES\_ID,

output\_index=0,

part=ResponseOutputText(

text="",

type="output\_text",

annotations=[],

),

type="response.content\_part.added",

)

# Emit the delta for this segment of refusal

yield ResponseRefusalDeltaEvent(

content\_index=state.refusal\_content\_index\_and\_output[0],

delta=delta.refusal,

item\_id=FAKE\_RESPONSES\_ID,

output\_index=0,

type="response.refusal.delta",

)

# Accumulate the refusal string in the output part

state.refusal\_content\_index\_and\_output[1].refusal += delta.refusal

# Handle tool calls

# Because we don't know the name of the function until the end of the stream, we'll

# save everything and yield events at the end

if delta.tool\_calls:

for tc\_delta in delta.tool\_calls:

if tc\_delta.index not in state.function\_calls:

state.function\_calls[tc\_delta.index] = ResponseFunctionToolCall(

id=FAKE\_RESPONSES\_ID,

arguments="",

name="",

type="function\_call",

call\_id="",

)

tc\_function = tc\_delta.function

state.function\_calls[tc\_delta.index].arguments += (

tc\_function.arguments if tc\_function else ""

) or ""

state.function\_calls[tc\_delta.index].name += (

tc\_function.name if tc\_function else ""

) or ""

state.function\_calls[tc\_delta.index].call\_id += tc\_delta.id or ""

function\_call\_starting\_index = 0

if state.text\_content\_index\_and\_output:

function\_call\_starting\_index += 1

# Send end event for this content part

yield ResponseContentPartDoneEvent(

content\_index=state.text\_content\_index\_and\_output[0],

item\_id=FAKE\_RESPONSES\_ID,

output\_index=0,

part=state.text\_content\_index\_and\_output[1],

type="response.content\_part.done",

)

if state.refusal\_content\_index\_and\_output:

function\_call\_starting\_index += 1

# Send end event for this content part

yield ResponseContentPartDoneEvent(

content\_index=state.refusal\_content\_index\_and\_output[0],

item\_id=FAKE\_RESPONSES\_ID,

output\_index=0,

part=state.refusal\_content\_index\_and\_output[1],

type="response.content\_part.done",

)

# Actually send events for the function calls

for function\_call in state.function\_calls.values():

# First, a ResponseOutputItemAdded for the function call

yield ResponseOutputItemAddedEvent(

item=ResponseFunctionToolCall(

id=FAKE\_RESPONSES\_ID,

call\_id=function\_call.call\_id,

arguments=function\_call.arguments,

name=function\_call.name,

type="function\_call",

),

output\_index=function\_call\_starting\_index,

type="response.output\_item.added",

)

# Then, yield the args

yield ResponseFunctionCallArgumentsDeltaEvent(

delta=function\_call.arguments,

item\_id=FAKE\_RESPONSES\_ID,

output\_index=function\_call\_starting\_index,

type="response.function\_call\_arguments.delta",

)

# Finally, the ResponseOutputItemDone

yield ResponseOutputItemDoneEvent(

item=ResponseFunctionToolCall(

id=FAKE\_RESPONSES\_ID,

call\_id=function\_call.call\_id,

arguments=function\_call.arguments,

name=function\_call.name,

type="function\_call",

),

output\_index=function\_call\_starting\_index,

type="response.output\_item.done",

)

# Finally, send the Response completed event

outputs: list[ResponseOutputItem] = []

if state.text\_content\_index\_and\_output or state.refusal\_content\_index\_and\_output:

assistant\_msg = ResponseOutputMessage(

id=FAKE\_RESPONSES\_ID,

content=[],

role="assistant",

type="message",

status="completed",

)

if state.text\_content\_index\_and\_output:

assistant\_msg.content.append(state.text\_content\_index\_and\_output[1])

if state.refusal\_content\_index\_and\_output:

assistant\_msg.content.append(state.refusal\_content\_index\_and\_output[1])

outputs.append(assistant\_msg)

# send a ResponseOutputItemDone for the assistant message

yield ResponseOutputItemDoneEvent(

item=assistant\_msg,

output\_index=0,

type="response.output\_item.done",

)

for function\_call in state.function\_calls.values():

outputs.append(function\_call)

final\_response = response.model\_copy()

final\_response.output = outputs

final\_response.usage = (

ResponseUsage(

input\_tokens=usage.prompt\_tokens,

output\_tokens=usage.completion\_tokens,

total\_tokens=usage.total\_tokens,

output\_tokens\_details=OutputTokensDetails(

reasoning\_tokens=usage.completion\_tokens\_details.reasoning\_tokens

if usage.completion\_tokens\_details

and usage.completion\_tokens\_details.reasoning\_tokens

else 0

),

input\_tokens\_details=InputTokensDetails(

cached\_tokens=usage.prompt\_tokens\_details.cached\_tokens

if usage.prompt\_tokens\_details and usage.prompt\_tokens\_details.cached\_tokens

else 0

),

)

if usage

else None

)

yield ResponseCompletedEvent(

response=final\_response,

type="response.completed",

)

if tracing.include\_data():

span\_generation.span\_data.output = [final\_response.model\_dump()]

if usage:

span\_generation.span\_data.usage = {

"input\_tokens": usage.prompt\_tokens,

"output\_tokens": usage.completion\_tokens,

}

@overload

async def \_fetch\_response(

self,

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

span: Span[GenerationSpanData],

tracing: ModelTracing,

stream: Literal[True],

) -> tuple[Response, AsyncStream[ChatCompletionChunk]]: ...

@overload

async def \_fetch\_response(

self,

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

span: Span[GenerationSpanData],

tracing: ModelTracing,

stream: Literal[False],

) -> ChatCompletion: ...

async def \_fetch\_response(

self,

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

span: Span[GenerationSpanData],

tracing: ModelTracing,

stream: bool = False,

) -> ChatCompletion | tuple[Response, AsyncStream[ChatCompletionChunk]]:

converted\_messages = \_Converter.items\_to\_messages(input)

if system\_instructions:

converted\_messages.insert(

0,

{

"content": system\_instructions,

"role": "system",

},

)

if tracing.include\_data():

span.span\_data.input = converted\_messages

parallel\_tool\_calls = (

True if model\_settings.parallel\_tool\_calls and tools and len(tools) > 0 else NOT\_GIVEN

)

tool\_choice = \_Converter.convert\_tool\_choice(model\_settings.tool\_choice)

response\_format = \_Converter.convert\_response\_format(output\_schema)

converted\_tools = [ToolConverter.to\_openai(tool) for tool in tools] if tools else []

for handoff in handoffs:

converted\_tools.append(ToolConverter.convert\_handoff\_tool(handoff))

if \_debug.DONT\_LOG\_MODEL\_DATA:

logger.debug("Calling LLM")

else:

logger.debug(

f"{json.dumps(converted\_messages, indent=2)}\n"

f"Tools:\n{json.dumps(converted\_tools, indent=2)}\n"

f"Stream: {stream}\n"

f"Tool choice: {tool\_choice}\n"

f"Response format: {response\_format}\n"

)

# Match the behavior of Responses where store is True when not given

store = model\_settings.store if model\_settings.store is not None else True

ret = await self.\_get\_client().chat.completions.create(

model=self.model,

messages=converted\_messages,

tools=converted\_tools or NOT\_GIVEN,

temperature=self.\_non\_null\_or\_not\_given(model\_settings.temperature),

top\_p=self.\_non\_null\_or\_not\_given(model\_settings.top\_p),

frequency\_penalty=self.\_non\_null\_or\_not\_given(model\_settings.frequency\_penalty),

presence\_penalty=self.\_non\_null\_or\_not\_given(model\_settings.presence\_penalty),

max\_tokens=self.\_non\_null\_or\_not\_given(model\_settings.max\_tokens),

tool\_choice=tool\_choice,

response\_format=response\_format,

parallel\_tool\_calls=parallel\_tool\_calls,

stream=stream,

stream\_options={"include\_usage": True} if stream else NOT\_GIVEN,

store=store,

extra\_headers=\_HEADERS,

)

if isinstance(ret, ChatCompletion):

return ret

response = Response(

id=FAKE\_RESPONSES\_ID,

created\_at=time.time(),

model=self.model,

object="response",

output=[],

tool\_choice=cast(Literal["auto", "required", "none"], tool\_choice)

if tool\_choice != NOT\_GIVEN

else "auto",

top\_p=model\_settings.top\_p,

temperature=model\_settings.temperature,

tools=[],

parallel\_tool\_calls=parallel\_tool\_calls or False,

)

return response, ret

def \_get\_client(self) -> AsyncOpenAI:

if self.\_client is None:

self.\_client = AsyncOpenAI()

return self.\_client

stream\_response async

stream\_response(

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

tracing: ModelTracing,

) -> AsyncIterator[TResponseStreamEvent]

Yields a partial message as it is generated, as well as the usage information.

Source code in src/agents/models/openai\_chatcompletions.py

async def stream\_response(

self,

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

tracing: ModelTracing,

) -> AsyncIterator[TResponseStreamEvent]:

"""

Yields a partial message as it is generated, as well as the usage information.

"""

with generation\_span(

model=str(self.model),

model\_config=dataclasses.asdict(model\_settings)

| {"base\_url": str(self.\_client.base\_url)},

disabled=tracing.is\_disabled(),

) as span\_generation:

response, stream = await self.\_fetch\_response(

system\_instructions,

input,

model\_settings,

tools,

output\_schema,

handoffs,

span\_generation,

tracing,

stream=True,

)

usage: CompletionUsage | None = None

state = \_StreamingState()

async for chunk in stream:

if not state.started:

state.started = True

yield ResponseCreatedEvent(

response=response,

type="response.created",

)

# The usage is only available in the last chunk

usage = chunk.usage

if not chunk.choices or not chunk.choices[0].delta:

continue

delta = chunk.choices[0].delta

# Handle text

if delta.content:

if not state.text\_content\_index\_and\_output:

# Initialize a content tracker for streaming text

state.text\_content\_index\_and\_output = (

0 if not state.refusal\_content\_index\_and\_output else 1,

ResponseOutputText(

text="",

type="output\_text",

annotations=[],

),

)

# Start a new assistant message stream

assistant\_item = ResponseOutputMessage(

id=FAKE\_RESPONSES\_ID,

content=[],

role="assistant",

type="message",

status="in\_progress",

)

# Notify consumers of the start of a new output message + first content part

yield ResponseOutputItemAddedEvent(

item=assistant\_item,

output\_index=0,

type="response.output\_item.added",

)

yield ResponseContentPartAddedEvent(

content\_index=state.text\_content\_index\_and\_output[0],

item\_id=FAKE\_RESPONSES\_ID,

output\_index=0,

part=ResponseOutputText(

text="",

type="output\_text",

annotations=[],

),

type="response.content\_part.added",

)

# Emit the delta for this segment of content

yield ResponseTextDeltaEvent(

content\_index=state.text\_content\_index\_and\_output[0],

delta=delta.content,

item\_id=FAKE\_RESPONSES\_ID,

output\_index=0,

type="response.output\_text.delta",

)

# Accumulate the text into the response part

state.text\_content\_index\_and\_output[1].text += delta.content

# Handle refusals (model declines to answer)

if delta.refusal:

if not state.refusal\_content\_index\_and\_output:

# Initialize a content tracker for streaming refusal text

state.refusal\_content\_index\_and\_output = (

0 if not state.text\_content\_index\_and\_output else 1,

ResponseOutputRefusal(refusal="", type="refusal"),

)

# Start a new assistant message if one doesn't exist yet (in-progress)

assistant\_item = ResponseOutputMessage(

id=FAKE\_RESPONSES\_ID,

content=[],

role="assistant",

type="message",

status="in\_progress",

)

# Notify downstream that assistant message + first content part are starting

yield ResponseOutputItemAddedEvent(

item=assistant\_item,

output\_index=0,

type="response.output\_item.added",

)

yield ResponseContentPartAddedEvent(

content\_index=state.refusal\_content\_index\_and\_output[0],

item\_id=FAKE\_RESPONSES\_ID,

output\_index=0,

part=ResponseOutputText(

text="",

type="output\_text",

annotations=[],

),

type="response.content\_part.added",

)

# Emit the delta for this segment of refusal

yield ResponseRefusalDeltaEvent(

content\_index=state.refusal\_content\_index\_and\_output[0],

delta=delta.refusal,

item\_id=FAKE\_RESPONSES\_ID,

output\_index=0,

type="response.refusal.delta",

)

# Accumulate the refusal string in the output part

state.refusal\_content\_index\_and\_output[1].refusal += delta.refusal

# Handle tool calls

# Because we don't know the name of the function until the end of the stream, we'll

# save everything and yield events at the end

if delta.tool\_calls:

for tc\_delta in delta.tool\_calls:

if tc\_delta.index not in state.function\_calls:

state.function\_calls[tc\_delta.index] = ResponseFunctionToolCall(

id=FAKE\_RESPONSES\_ID,

arguments="",

name="",

type="function\_call",

call\_id="",

)

tc\_function = tc\_delta.function

state.function\_calls[tc\_delta.index].arguments += (

tc\_function.arguments if tc\_function else ""

) or ""

state.function\_calls[tc\_delta.index].name += (

tc\_function.name if tc\_function else ""

) or ""

state.function\_calls[tc\_delta.index].call\_id += tc\_delta.id or ""

function\_call\_starting\_index = 0

if state.text\_content\_index\_and\_output:

function\_call\_starting\_index += 1

# Send end event for this content part

yield ResponseContentPartDoneEvent(

content\_index=state.text\_content\_index\_and\_output[0],

item\_id=FAKE\_RESPONSES\_ID,

output\_index=0,

part=state.text\_content\_index\_and\_output[1],

type="response.content\_part.done",

)

if state.refusal\_content\_index\_and\_output:

function\_call\_starting\_index += 1

# Send end event for this content part

yield ResponseContentPartDoneEvent(

content\_index=state.refusal\_content\_index\_and\_output[0],

item\_id=FAKE\_RESPONSES\_ID,

output\_index=0,

part=state.refusal\_content\_index\_and\_output[1],

type="response.content\_part.done",

)

# Actually send events for the function calls

for function\_call in state.function\_calls.values():

# First, a ResponseOutputItemAdded for the function call

yield ResponseOutputItemAddedEvent(

item=ResponseFunctionToolCall(

id=FAKE\_RESPONSES\_ID,

call\_id=function\_call.call\_id,

arguments=function\_call.arguments,

name=function\_call.name,

type="function\_call",

),

output\_index=function\_call\_starting\_index,

type="response.output\_item.added",

)

# Then, yield the args

yield ResponseFunctionCallArgumentsDeltaEvent(

delta=function\_call.arguments,

item\_id=FAKE\_RESPONSES\_ID,

output\_index=function\_call\_starting\_index,

type="response.function\_call\_arguments.delta",

)

# Finally, the ResponseOutputItemDone

yield ResponseOutputItemDoneEvent(

item=ResponseFunctionToolCall(

id=FAKE\_RESPONSES\_ID,

call\_id=function\_call.call\_id,

arguments=function\_call.arguments,

name=function\_call.name,

type="function\_call",

),

output\_index=function\_call\_starting\_index,

type="response.output\_item.done",

)

# Finally, send the Response completed event

outputs: list[ResponseOutputItem] = []

if state.text\_content\_index\_and\_output or state.refusal\_content\_index\_and\_output:

assistant\_msg = ResponseOutputMessage(

id=FAKE\_RESPONSES\_ID,

content=[],

role="assistant",

type="message",

status="completed",

)

if state.text\_content\_index\_and\_output:

assistant\_msg.content.append(state.text\_content\_index\_and\_output[1])

if state.refusal\_content\_index\_and\_output:

assistant\_msg.content.append(state.refusal\_content\_index\_and\_output[1])

outputs.append(assistant\_msg)

# send a ResponseOutputItemDone for the assistant message

yield ResponseOutputItemDoneEvent(

item=assistant\_msg,

output\_index=0,

type="response.output\_item.done",

)

for function\_call in state.function\_calls.values():

outputs.append(function\_call)

final\_response = response.model\_copy()

final\_response.output = outputs

final\_response.usage = (

ResponseUsage(

input\_tokens=usage.prompt\_tokens,

output\_tokens=usage.completion\_tokens,

total\_tokens=usage.total\_tokens,

output\_tokens\_details=OutputTokensDetails(

reasoning\_tokens=usage.completion\_tokens\_details.reasoning\_tokens

if usage.completion\_tokens\_details

and usage.completion\_tokens\_details.reasoning\_tokens

else 0

),

input\_tokens\_details=InputTokensDetails(

cached\_tokens=usage.prompt\_tokens\_details.cached\_tokens

if usage.prompt\_tokens\_details and usage.prompt\_tokens\_details.cached\_tokens

else 0

),

)

if usage

else None

)

yield ResponseCompletedEvent(

response=final\_response,

type="response.completed",

)

if tracing.include\_data():

span\_generation.span\_data.output = [final\_response.model\_dump()]

if usage:

span\_generation.span\_data.usage = {

"input\_tokens": usage.prompt\_tokens,

"output\_tokens": usage.completion\_tokens,

}

OpenAI Responses model

OpenAIResponsesModel

Bases: Model

Implementation of Model that uses the OpenAI Responses API.

Source code in src/agents/models/openai\_responses.py

class OpenAIResponsesModel(Model):

"""

Implementation of `Model` that uses the OpenAI Responses API.

"""

def \_\_init\_\_(

self,

model: str | ChatModel,

openai\_client: AsyncOpenAI,

) -> None:

self.model = model

self.\_client = openai\_client

def \_non\_null\_or\_not\_given(self, value: Any) -> Any:

return value if value is not None else NOT\_GIVEN

async def get\_response(

self,

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

tracing: ModelTracing,

) -> ModelResponse:

with response\_span(disabled=tracing.is\_disabled()) as span\_response:

try:

response = await self.\_fetch\_response(

system\_instructions,

input,

model\_settings,

tools,

output\_schema,

handoffs,

stream=False,

)

if \_debug.DONT\_LOG\_MODEL\_DATA:

logger.debug("LLM responded")

else:

logger.debug(

"LLM resp:\n"

f"{json.dumps([x.model\_dump() for x in response.output], indent=2)}\n"

)

usage = (

Usage(

requests=1,

input\_tokens=response.usage.input\_tokens,

output\_tokens=response.usage.output\_tokens,

total\_tokens=response.usage.total\_tokens,

)

if response.usage

else Usage()

)

if tracing.include\_data():

span\_response.span\_data.response = response

span\_response.span\_data.input = input

except Exception as e:

span\_response.set\_error(

SpanError(

message="Error getting response",

data={

"error": str(e) if tracing.include\_data() else e.\_\_class\_\_.\_\_name\_\_,

},

)

)

request\_id = e.request\_id if isinstance(e, APIStatusError) else None

logger.error(f"Error getting response: {e}. (request\_id: {request\_id})")

raise

return ModelResponse(

output=response.output,

usage=usage,

referenceable\_id=response.id,

)

async def stream\_response(

self,

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

tracing: ModelTracing,

) -> AsyncIterator[ResponseStreamEvent]:

"""

Yields a partial message as it is generated, as well as the usage information.

"""

with response\_span(disabled=tracing.is\_disabled()) as span\_response:

try:

stream = await self.\_fetch\_response(

system\_instructions,

input,

model\_settings,

tools,

output\_schema,

handoffs,

stream=True,

)

final\_response: Response | None = None

async for chunk in stream:

if isinstance(chunk, ResponseCompletedEvent):

final\_response = chunk.response

yield chunk

if final\_response and tracing.include\_data():

span\_response.span\_data.response = final\_response

span\_response.span\_data.input = input

except Exception as e:

span\_response.set\_error(

SpanError(

message="Error streaming response",

data={

"error": str(e) if tracing.include\_data() else e.\_\_class\_\_.\_\_name\_\_,

},

)

)

logger.error(f"Error streaming response: {e}")

raise

@overload

async def \_fetch\_response(

self,

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

stream: Literal[True],

) -> AsyncStream[ResponseStreamEvent]: ...

@overload

async def \_fetch\_response(

self,

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

stream: Literal[False],

) -> Response: ...

async def \_fetch\_response(

self,

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

stream: Literal[True] | Literal[False] = False,

) -> Response | AsyncStream[ResponseStreamEvent]:

list\_input = ItemHelpers.input\_to\_new\_input\_list(input)

parallel\_tool\_calls = (

True

if model\_settings.parallel\_tool\_calls and tools and len(tools) > 0

else False

if model\_settings.parallel\_tool\_calls is False

else NOT\_GIVEN

)

tool\_choice = Converter.convert\_tool\_choice(model\_settings.tool\_choice)

converted\_tools = Converter.convert\_tools(tools, handoffs)

response\_format = Converter.get\_response\_format(output\_schema)

if \_debug.DONT\_LOG\_MODEL\_DATA:

logger.debug("Calling LLM")

else:

logger.debug(

f"Calling LLM {self.model} with input:\n"

f"{json.dumps(list\_input, indent=2)}\n"

f"Tools:\n{json.dumps(converted\_tools.tools, indent=2)}\n"

f"Stream: {stream}\n"

f"Tool choice: {tool\_choice}\n"

f"Response format: {response\_format}\n"

)

return await self.\_client.responses.create(

instructions=self.\_non\_null\_or\_not\_given(system\_instructions),

model=self.model,

input=list\_input,

include=converted\_tools.includes,

tools=converted\_tools.tools,

temperature=self.\_non\_null\_or\_not\_given(model\_settings.temperature),

top\_p=self.\_non\_null\_or\_not\_given(model\_settings.top\_p),

truncation=self.\_non\_null\_or\_not\_given(model\_settings.truncation),

max\_output\_tokens=self.\_non\_null\_or\_not\_given(model\_settings.max\_tokens),

tool\_choice=tool\_choice,

parallel\_tool\_calls=parallel\_tool\_calls,

stream=stream,

extra\_headers=\_HEADERS,

text=response\_format,

store=self.\_non\_null\_or\_not\_given(model\_settings.store),

)

def \_get\_client(self) -> AsyncOpenAI:

if self.\_client is None:

self.\_client = AsyncOpenAI()

return self.\_client

stream\_response async

stream\_response(

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

tracing: ModelTracing,

) -> AsyncIterator[ResponseStreamEvent]

Yields a partial message as it is generated, as well as the usage information.

Source code in src/agents/models/openai\_responses.py

async def stream\_response(

self,

system\_instructions: str | None,

input: str | list[TResponseInputItem],

model\_settings: ModelSettings,

tools: list[Tool],

output\_schema: AgentOutputSchema | None,

handoffs: list[Handoff],

tracing: ModelTracing,

) -> AsyncIterator[ResponseStreamEvent]:

"""

Yields a partial message as it is generated, as well as the usage information.

"""

with response\_span(disabled=tracing.is\_disabled()) as span\_response:

try:

stream = await self.\_fetch\_response(

system\_instructions,

input,

model\_settings,

tools,

output\_schema,

handoffs,

stream=True,

)

final\_response: Response | None = None

async for chunk in stream:

if isinstance(chunk, ResponseCompletedEvent):

final\_response = chunk.response

yield chunk

if final\_response and tracing.include\_data():

span\_response.span\_data.response = final\_response

span\_response.span\_data.input = input

except Exception as e:

span\_response.set\_error(

SpanError(

message="Error streaming response",

data={

"error": str(e) if tracing.include\_data() else e.\_\_class\_\_.\_\_name\_\_,

},

)

)

logger.error(f"Error streaming response: {e}")

raise

Converter

Source code in src/agents/models/openai\_responses.py

class Converter:

@classmethod

def convert\_tool\_choice(

cls, tool\_choice: Literal["auto", "required", "none"] | str | None

) -> response\_create\_params.ToolChoice | NotGiven:

if tool\_choice is None:

return NOT\_GIVEN

elif tool\_choice == "required":

return "required"

elif tool\_choice == "auto":

return "auto"

elif tool\_choice == "none":

return "none"

elif tool\_choice == "file\_search":

return {

"type": "file\_search",

}

elif tool\_choice == "web\_search\_preview":

return {

"type": "web\_search\_preview",

}

elif tool\_choice == "computer\_use\_preview":

return {

"type": "computer\_use\_preview",

}

else:

return {

"type": "function",

"name": tool\_choice,

}

@classmethod

def get\_response\_format(

cls, output\_schema: AgentOutputSchema | None

) -> ResponseTextConfigParam | NotGiven:

if output\_schema is None or output\_schema.is\_plain\_text():

return NOT\_GIVEN

else:

return {

"format": {

"type": "json\_schema",

"name": "final\_output",

"schema": output\_schema.json\_schema(),

"strict": output\_schema.strict\_json\_schema,

}

}

@classmethod

def convert\_tools(

cls,

tools: list[Tool],

handoffs: list[Handoff[Any]],

) -> ConvertedTools:

converted\_tools: list[ToolParam] = []

includes: list[IncludeLiteral] = []

computer\_tools = [tool for tool in tools if isinstance(tool, ComputerTool)]

if len(computer\_tools) > 1:

raise UserError(f"You can only provide one computer tool. Got {len(computer\_tools)}")

for tool in tools:

converted\_tool, include = cls.\_convert\_tool(tool)

converted\_tools.append(converted\_tool)

if include:

includes.append(include)

for handoff in handoffs:

converted\_tools.append(cls.\_convert\_handoff\_tool(handoff))

return ConvertedTools(tools=converted\_tools, includes=includes)

@classmethod

def \_convert\_tool(cls, tool: Tool) -> tuple[ToolParam, IncludeLiteral | None]:

"""Returns converted tool and includes"""

if isinstance(tool, FunctionTool):

converted\_tool: ToolParam = {

"name": tool.name,

"parameters": tool.params\_json\_schema,

"strict": tool.strict\_json\_schema,

"type": "function",

"description": tool.description,

}

includes: IncludeLiteral | None = None

elif isinstance(tool, WebSearchTool):

ws: WebSearchToolParam = {

"type": "web\_search\_preview",

"user\_location": tool.user\_location,

"search\_context\_size": tool.search\_context\_size,

}

converted\_tool = ws

includes = None

elif isinstance(tool, FileSearchTool):

converted\_tool = {

"type": "file\_search",

"vector\_store\_ids": tool.vector\_store\_ids,

}

if tool.max\_num\_results:

converted\_tool["max\_num\_results"] = tool.max\_num\_results

if tool.ranking\_options:

converted\_tool["ranking\_options"] = tool.ranking\_options

if tool.filters:

converted\_tool["filters"] = tool.filters

includes = "file\_search\_call.results" if tool.include\_search\_results else None

elif isinstance(tool, ComputerTool):

converted\_tool = {

"type": "computer\_use\_preview",

"environment": tool.computer.environment,

"display\_width": tool.computer.dimensions[0],

"display\_height": tool.computer.dimensions[1],

}

includes = None

else:

raise UserError(f"Unknown tool type: {type(tool)}, tool")

return converted\_tool, includes

@classmethod

def \_convert\_handoff\_tool(cls, handoff: Handoff) -> ToolParam:

return {

"name": handoff.tool\_name,

"parameters": handoff.input\_json\_schema,

"strict": handoff.strict\_json\_schema,

"type": "function",

"description": handoff.tool\_description,

}

MCP Servers

MCPServer

Bases: ABC

Base class for Model Context Protocol servers.

Source code in src/agents/mcp/server.py

class MCPServer(abc.ABC):

"""Base class for Model Context Protocol servers."""

@abc.abstractmethod

async def connect(self):

"""Connect to the server. For example, this might mean spawning a subprocess or

opening a network connection. The server is expected to remain connected until

`cleanup()` is called.

"""

pass

@property

@abc.abstractmethod

def name(self) -> str:

"""A readable name for the server."""

pass

@abc.abstractmethod

async def cleanup(self):

"""Cleanup the server. For example, this might mean closing a subprocess or

closing a network connection.

"""

pass

@abc.abstractmethod

async def list\_tools(self) -> list[MCPTool]:

"""List the tools available on the server."""

pass

@abc.abstractmethod

async def call\_tool(self, tool\_name: str, arguments: dict[str, Any] | None) -> CallToolResult:

"""Invoke a tool on the server."""

pass

name abstractmethod property

name: str

A readable name for the server.

connect abstractmethod async

connect()

Connect to the server. For example, this might mean spawning a subprocess or opening a network connection. The server is expected to remain connected until cleanup() is called.

Source code in src/agents/mcp/server.py

@abc.abstractmethod

async def connect(self):

"""Connect to the server. For example, this might mean spawning a subprocess or

opening a network connection. The server is expected to remain connected until

`cleanup()` is called.

"""

pass

cleanup abstractmethod async

cleanup()

Cleanup the server. For example, this might mean closing a subprocess or closing a network connection.

Source code in src/agents/mcp/server.py

@abc.abstractmethod

async def cleanup(self):

"""Cleanup the server. For example, this might mean closing a subprocess or

closing a network connection.

"""

pass

list\_tools abstractmethod async

list\_tools() -> list[Tool]

List the tools available on the server.

Source code in src/agents/mcp/server.py

@abc.abstractmethod

async def list\_tools(self) -> list[MCPTool]:

"""List the tools available on the server."""

pass

call\_tool abstractmethod async

call\_tool(

tool\_name: str, arguments: dict[str, Any] | None

) -> CallToolResult

Invoke a tool on the server.

Source code in src/agents/mcp/server.py

@abc.abstractmethod

async def call\_tool(self, tool\_name: str, arguments: dict[str, Any] | None) -> CallToolResult:

"""Invoke a tool on the server."""

pass

MCPServerStdioParams

Bases: TypedDict

Mirrors mcp.client.stdio.StdioServerParameters, but lets you pass params without another import.

Source code in src/agents/mcp/server.py

class MCPServerStdioParams(TypedDict):

"""Mirrors `mcp.client.stdio.StdioServerParameters`, but lets you pass params without another

import.

"""

command: str

"""The executable to run to start the server. For example, `python` or `node`."""

args: NotRequired[list[str]]

"""Command line args to pass to the `command` executable. For example, `['foo.py']` or

`['server.js', '--port', '8080']`."""

env: NotRequired[dict[str, str]]

"""The environment variables to set for the server. ."""

cwd: NotRequired[str | Path]

"""The working directory to use when spawning the process."""

encoding: NotRequired[str]

"""The text encoding used when sending/receiving messages to the server. Defaults to `utf-8`."""

encoding\_error\_handler: NotRequired[Literal["strict", "ignore", "replace"]]

"""The text encoding error handler. Defaults to `strict`.

See https://docs.python.org/3/library/codecs.html#codec-base-classes for

explanations of possible values.

"""

command instance-attribute

command: str

The executable to run to start the server. For example, python or node.

args instance-attribute

args: NotRequired[list[str]]

Command line args to pass to the command executable. For example, ['foo.py'] or ['server.js', '--port', '8080'].

env instance-attribute

env: NotRequired[dict[str, str]]

The environment variables to set for the server. .

cwd instance-attribute

cwd: NotRequired[str | Path]

The working directory to use when spawning the process.

encoding instance-attribute

encoding: NotRequired[str]

The text encoding used when sending/receiving messages to the server. Defaults to utf-8.

encoding\_error\_handler instance-attribute

encoding\_error\_handler: NotRequired[

Literal["strict", "ignore", "replace"]

]

The text encoding error handler. Defaults to strict.

See https://docs.python.org/3/library/codecs.html#codec-base-classes for explanations of possible values.

MCPServerStdio

Bases: \_MCPServerWithClientSession

MCP server implementation that uses the stdio transport. See the [spec] (https://spec.modelcontextprotocol.io/specification/2024-11-05/basic/transports/#stdio) for details.

Source code in src/agents/mcp/server.py

class MCPServerStdio(\_MCPServerWithClientSession):

"""MCP server implementation that uses the stdio transport. See the [spec]

(https://spec.modelcontextprotocol.io/specification/2024-11-05/basic/transports/#stdio) for

details.

"""

def \_\_init\_\_(

self,

params: MCPServerStdioParams,

cache\_tools\_list: bool = False,

name: str | None = None,

):

"""Create a new MCP server based on the stdio transport.

Args:

params: The params that configure the server. This includes the command to run to

start the server, the args to pass to the command, the environment variables to

set for the server, the working directory to use when spawning the process, and

the text encoding used when sending/receiving messages to the server.

cache\_tools\_list: Whether to cache the tools list. If `True`, the tools list will be

cached and only fetched from the server once. If `False`, the tools list will be

fetched from the server on each call to `list\_tools()`. The cache can be

invalidated by calling `invalidate\_tools\_cache()`. You should set this to `True`

if you know the server will not change its tools list, because it can drastically

improve latency (by avoiding a round-trip to the server every time).

name: A readable name for the server. If not provided, we'll create one from the

command.

"""

super().\_\_init\_\_(cache\_tools\_list)

self.params = StdioServerParameters(

command=params["command"],

args=params.get("args", []),

env=params.get("env"),

cwd=params.get("cwd"),

encoding=params.get("encoding", "utf-8"),

encoding\_error\_handler=params.get("encoding\_error\_handler", "strict"),

)

self.\_name = name or f"stdio: {self.params.command}"

def create\_streams(

self,

) -> AbstractAsyncContextManager[

tuple[

MemoryObjectReceiveStream[JSONRPCMessage | Exception],

MemoryObjectSendStream[JSONRPCMessage],

]

]:

"""Create the streams for the server."""

return stdio\_client(self.params)

@property

def name(self) -> str:

"""A readable name for the server."""

return self.\_name

name property

name: str

A readable name for the server.

connect async

connect()

Connect to the server.

Source code in src/agents/mcp/server.py

async def connect(self):

"""Connect to the server."""

try:

transport = await self.exit\_stack.enter\_async\_context(self.create\_streams())

read, write = transport

session = await self.exit\_stack.enter\_async\_context(ClientSession(read, write))

await session.initialize()

self.session = session

except Exception as e:

logger.error(f"Error initializing MCP server: {e}")

await self.cleanup()

raise

cleanup async

cleanup()

Cleanup the server.

Source code in src/agents/mcp/server.py

async def cleanup(self):

"""Cleanup the server."""

async with self.\_cleanup\_lock:

try:

await self.exit\_stack.aclose()

self.session = None

except Exception as e:

logger.error(f"Error cleaning up server: {e}")

list\_tools async

list\_tools() -> list[Tool]

List the tools available on the server.

Source code in src/agents/mcp/server.py

async def list\_tools(self) -> list[MCPTool]:

"""List the tools available on the server."""

if not self.session:

raise UserError("Server not initialized. Make sure you call `connect()` first.")

# Return from cache if caching is enabled, we have tools, and the cache is not dirty

if self.cache\_tools\_list and not self.\_cache\_dirty and self.\_tools\_list:

return self.\_tools\_list

# Reset the cache dirty to False

self.\_cache\_dirty = False

# Fetch the tools from the server

self.\_tools\_list = (await self.session.list\_tools()).tools

return self.\_tools\_list

call\_tool async

call\_tool(

tool\_name: str, arguments: dict[str, Any] | None

) -> CallToolResult

Invoke a tool on the server.

Source code in src/agents/mcp/server.py

async def call\_tool(self, tool\_name: str, arguments: dict[str, Any] | None) -> CallToolResult:

"""Invoke a tool on the server."""

if not self.session:

raise UserError("Server not initialized. Make sure you call `connect()` first.")

return await self.session.call\_tool(tool\_name, arguments)

invalidate\_tools\_cache

invalidate\_tools\_cache()

Invalidate the tools cache.

Source code in src/agents/mcp/server.py

def invalidate\_tools\_cache(self):

"""Invalidate the tools cache."""

self.\_cache\_dirty = True

\_\_init\_\_

\_\_init\_\_(

params: MCPServerStdioParams,

cache\_tools\_list: bool = False,

name: str | None = None,

)

Create a new MCP server based on the stdio transport.

Parameters:

Name Type Description Default

params MCPServerStdioParams The params that configure the server. This includes the command to run to start the server, the args to pass to the command, the environment variables to set for the server, the working directory to use when spawning the process, and the text encoding used when sending/receiving messages to the server. required

cache\_tools\_list bool Whether to cache the tools list. If True, the tools list will be cached and only fetched from the server once. If False, the tools list will be fetched from the server on each call to list\_tools(). The cache can be invalidated by calling invalidate\_tools\_cache(). You should set this to True if you know the server will not change its tools list, because it can drastically improve latency (by avoiding a round-trip to the server every time). False

name str | None A readable name for the server. If not provided, we'll create one from the command. None

Source code in src/agents/mcp/server.py

def \_\_init\_\_(

self,

params: MCPServerStdioParams,

cache\_tools\_list: bool = False,

name: str | None = None,

):

"""Create a new MCP server based on the stdio transport.

Args:

params: The params that configure the server. This includes the command to run to

start the server, the args to pass to the command, the environment variables to

set for the server, the working directory to use when spawning the process, and

the text encoding used when sending/receiving messages to the server.

cache\_tools\_list: Whether to cache the tools list. If `True`, the tools list will be

cached and only fetched from the server once. If `False`, the tools list will be

fetched from the server on each call to `list\_tools()`. The cache can be

invalidated by calling `invalidate\_tools\_cache()`. You should set this to `True`

if you know the server will not change its tools list, because it can drastically

improve latency (by avoiding a round-trip to the server every time).

name: A readable name for the server. If not provided, we'll create one from the

command.

"""

super().\_\_init\_\_(cache\_tools\_list)

self.params = StdioServerParameters(

command=params["command"],

args=params.get("args", []),

env=params.get("env"),

cwd=params.get("cwd"),

encoding=params.get("encoding", "utf-8"),

encoding\_error\_handler=params.get("encoding\_error\_handler", "strict"),

)

self.\_name = name or f"stdio: {self.params.command}"

create\_streams

create\_streams() -> AbstractAsyncContextManager[

tuple[

MemoryObjectReceiveStream[

JSONRPCMessage | Exception

],

MemoryObjectSendStream[JSONRPCMessage],

]

]

Create the streams for the server.

Source code in src/agents/mcp/server.py

def create\_streams(

self,

) -> AbstractAsyncContextManager[

tuple[

MemoryObjectReceiveStream[JSONRPCMessage | Exception],

MemoryObjectSendStream[JSONRPCMessage],

]

]:

"""Create the streams for the server."""

return stdio\_client(self.params)

MCPServerSseParams

Bases: TypedDict

Mirrors the params inmcp.client.sse.sse\_client.

Source code in src/agents/mcp/server.py

class MCPServerSseParams(TypedDict):

"""Mirrors the params in`mcp.client.sse.sse\_client`."""

url: str

"""The URL of the server."""

headers: NotRequired[dict[str, str]]

"""The headers to send to the server."""

timeout: NotRequired[float]

"""The timeout for the HTTP request. Defaults to 5 seconds."""

sse\_read\_timeout: NotRequired[float]

"""The timeout for the SSE connection, in seconds. Defaults to 5 minutes."""

url instance-attribute

url: str

The URL of the server.

headers instance-attribute

headers: NotRequired[dict[str, str]]

The headers to send to the server.

timeout instance-attribute

timeout: NotRequired[float]

The timeout for the HTTP request. Defaults to 5 seconds.

sse\_read\_timeout instance-attribute

sse\_read\_timeout: NotRequired[float]

The timeout for the SSE connection, in seconds. Defaults to 5 minutes.

MCPServerSse

Bases: \_MCPServerWithClientSession

MCP server implementation that uses the HTTP with SSE transport. See the [spec] (https://spec.modelcontextprotocol.io/specification/2024-11-05/basic/transports/#http-with-sse) for details.

Source code in src/agents/mcp/server.py

class MCPServerSse(\_MCPServerWithClientSession):

"""MCP server implementation that uses the HTTP with SSE transport. See the [spec]

(https://spec.modelcontextprotocol.io/specification/2024-11-05/basic/transports/#http-with-sse)

for details.

"""

def \_\_init\_\_(

self,

params: MCPServerSseParams,

cache\_tools\_list: bool = False,

name: str | None = None,

):

"""Create a new MCP server based on the HTTP with SSE transport.

Args:

params: The params that configure the server. This includes the URL of the server,

the headers to send to the server, the timeout for the HTTP request, and the

timeout for the SSE connection.

cache\_tools\_list: Whether to cache the tools list. If `True`, the tools list will be

cached and only fetched from the server once. If `False`, the tools list will be

fetched from the server on each call to `list\_tools()`. The cache can be

invalidated by calling `invalidate\_tools\_cache()`. You should set this to `True`

if you know the server will not change its tools list, because it can drastically

improve latency (by avoiding a round-trip to the server every time).

name: A readable name for the server. If not provided, we'll create one from the

URL.

"""

super().\_\_init\_\_(cache\_tools\_list)

self.params = params

self.\_name = name or f"sse: {self.params['url']}"

def create\_streams(

self,

) -> AbstractAsyncContextManager[

tuple[

MemoryObjectReceiveStream[JSONRPCMessage | Exception],

MemoryObjectSendStream[JSONRPCMessage],

]

]:

"""Create the streams for the server."""

return sse\_client(

url=self.params["url"],

headers=self.params.get("headers", None),

timeout=self.params.get("timeout", 5),

sse\_read\_timeout=self.params.get("sse\_read\_timeout", 60 \* 5),

)

@property

def name(self) -> str:

"""A readable name for the server."""

return self.\_name

name property

name: str

A readable name for the server.

connect async

connect()

Connect to the server.

Source code in src/agents/mcp/server.py

async def connect(self):

"""Connect to the server."""

try:

transport = await self.exit\_stack.enter\_async\_context(self.create\_streams())

read, write = transport

session = await self.exit\_stack.enter\_async\_context(ClientSession(read, write))

await session.initialize()

self.session = session

except Exception as e:

logger.error(f"Error initializing MCP server: {e}")

await self.cleanup()

raise

cleanup async

cleanup()

Cleanup the server.

Source code in src/agents/mcp/server.py

async def cleanup(self):

"""Cleanup the server."""

async with self.\_cleanup\_lock:

try:

await self.exit\_stack.aclose()

self.session = None

except Exception as e:

logger.error(f"Error cleaning up server: {e}")

list\_tools async

list\_tools() -> list[Tool]

List the tools available on the server.

Source code in src/agents/mcp/server.py

async def list\_tools(self) -> list[MCPTool]:

"""List the tools available on the server."""

if not self.session:

raise UserError("Server not initialized. Make sure you call `connect()` first.")

# Return from cache if caching is enabled, we have tools, and the cache is not dirty

if self.cache\_tools\_list and not self.\_cache\_dirty and self.\_tools\_list:

return self.\_tools\_list

# Reset the cache dirty to False

self.\_cache\_dirty = False

# Fetch the tools from the server

self.\_tools\_list = (await self.session.list\_tools()).tools

return self.\_tools\_list

call\_tool async

call\_tool(

tool\_name: str, arguments: dict[str, Any] | None

) -> CallToolResult

Invoke a tool on the server.

Source code in src/agents/mcp/server.py

async def call\_tool(self, tool\_name: str, arguments: dict[str, Any] | None) -> CallToolResult:

"""Invoke a tool on the server."""

if not self.session:

raise UserError("Server not initialized. Make sure you call `connect()` first.")

return await self.session.call\_tool(tool\_name, arguments)

invalidate\_tools\_cache

invalidate\_tools\_cache()

Invalidate the tools cache.

Source code in src/agents/mcp/server.py

def invalidate\_tools\_cache(self):

"""Invalidate the tools cache."""

self.\_cache\_dirty = True

\_\_init\_\_

\_\_init\_\_(

params: MCPServerSseParams,

cache\_tools\_list: bool = False,

name: str | None = None,

)

Create a new MCP server based on the HTTP with SSE transport.

Parameters:

Name Type Description Default

params MCPServerSseParams The params that configure the server. This includes the URL of the server, the headers to send to the server, the timeout for the HTTP request, and the timeout for the SSE connection. required

cache\_tools\_list bool Whether to cache the tools list. If True, the tools list will be cached and only fetched from the server once. If False, the tools list will be fetched from the server on each call to list\_tools(). The cache can be invalidated by calling invalidate\_tools\_cache(). You should set this to True if you know the server will not change its tools list, because it can drastically improve latency (by avoiding a round-trip to the server every time). False

name str | None A readable name for the server. If not provided, we'll create one from the URL. None

Source code in src/agents/mcp/server.py

create\_streams

create\_streams() -> AbstractAsyncContextManager[

tuple[

MemoryObjectReceiveStream[

JSONRPCMessage | Exception

],

MemoryObjectSendStream[JSONRPCMessage],

]

]

Create the streams for the server.

Source code in src/agents/mcp/server.py

def create\_streams(

self,

) -> AbstractAsyncContextManager[

tuple[

MemoryObjectReceiveStream[JSONRPCMessage | Exception],

MemoryObjectSendStream[JSONRPCMessage],

]

]:

"""Create the streams for the server."""

return sse\_client(

url=self.params["url"],

headers=self.params.get("headers", None),

timeout=self.params.get("timeout", 5),

sse\_read\_timeout=self.params.get("sse\_read\_timeout", 60 \* 5),

)

MCP Util

MCPUtil

Set of utilities for interop between MCP and Agents SDK tools.

Source code in src/agents/mcp/util.py

class MCPUtil:

"""Set of utilities for interop between MCP and Agents SDK tools."""

@classmethod

async def get\_all\_function\_tools(cls, servers: list["MCPServer"]) -> list[Tool]:

"""Get all function tools from a list of MCP servers."""

tools = []

tool\_names: set[str] = set()

for server in servers:

server\_tools = await cls.get\_function\_tools(server)

server\_tool\_names = {tool.name for tool in server\_tools}

if len(server\_tool\_names & tool\_names) > 0:

raise UserError(

f"Duplicate tool names found across MCP servers: "

f"{server\_tool\_names & tool\_names}"

)

tool\_names.update(server\_tool\_names)

tools.extend(server\_tools)

return tools

@classmethod

async def get\_function\_tools(cls, server: "MCPServer") -> list[Tool]:

"""Get all function tools from a single MCP server."""

with mcp\_tools\_span(server=server.name) as span:

tools = await server.list\_tools()

span.span\_data.result = [tool.name for tool in tools]

return [cls.to\_function\_tool(tool, server) for tool in tools]

@classmethod

def to\_function\_tool(cls, tool: "MCPTool", server: "MCPServer") -> FunctionTool:

"""Convert an MCP tool to an Agents SDK function tool."""

invoke\_func = functools.partial(cls.invoke\_mcp\_tool, server, tool)

return FunctionTool(

name=tool.name,

description=tool.description or "",

params\_json\_schema=tool.inputSchema,

on\_invoke\_tool=invoke\_func,

strict\_json\_schema=False,

)

@classmethod

async def invoke\_mcp\_tool(

cls, server: "MCPServer", tool: "MCPTool", context: RunContextWrapper[Any], input\_json: str

) -> str:

"""Invoke an MCP tool and return the result as a string."""

try:

json\_data: dict[str, Any] = json.loads(input\_json) if input\_json else {}

except Exception as e:

if \_debug.DONT\_LOG\_TOOL\_DATA:

logger.debug(f"Invalid JSON input for tool {tool.name}")

else:

logger.debug(f"Invalid JSON input for tool {tool.name}: {input\_json}")

raise ModelBehaviorError(

f"Invalid JSON input for tool {tool.name}: {input\_json}"

) from e

if \_debug.DONT\_LOG\_TOOL\_DATA:

logger.debug(f"Invoking MCP tool {tool.name}")

else:

logger.debug(f"Invoking MCP tool {tool.name} with input {input\_json}")

try:

result = await server.call\_tool(tool.name, json\_data)

except Exception as e:

logger.error(f"Error invoking MCP tool {tool.name}: {e}")

raise AgentsException(f"Error invoking MCP tool {tool.name}: {e}") from e

if \_debug.DONT\_LOG\_TOOL\_DATA:

logger.debug(f"MCP tool {tool.name} completed.")

else:

logger.debug(f"MCP tool {tool.name} returned {result}")

# The MCP tool result is a list of content items, whereas OpenAI tool outputs are a single

# string. We'll try to convert.

if len(result.content) == 1:

tool\_output = result.content[0].model\_dump\_json()

elif len(result.content) > 1:

tool\_output = json.dumps([item.model\_dump() for item in result.content])

else:

logger.error(f"Errored MCP tool result: {result}")

tool\_output = "Error running tool."

current\_span = get\_current\_span()

if current\_span:

if isinstance(current\_span.span\_data, FunctionSpanData):

current\_span.span\_data.output = tool\_output

current\_span.span\_data.mcp\_data = {

"server": server.name,

}

else:

logger.warning(

f"Current span is not a FunctionSpanData, skipping tool output: {current\_span}"

)

return tool\_output

get\_all\_function\_tools async classmethod

get\_all\_function\_tools(

servers: list[MCPServer],

) -> list[Tool]

Get all function tools from a list of MCP servers.

Source code in src/agents/mcp/util.py

@classmethod

async def get\_all\_function\_tools(cls, servers: list["MCPServer"]) -> list[Tool]:

"""Get all function tools from a list of MCP servers."""

tools = []

tool\_names: set[str] = set()

for server in servers:

server\_tools = await cls.get\_function\_tools(server)

server\_tool\_names = {tool.name for tool in server\_tools}

if len(server\_tool\_names & tool\_names) > 0:

raise UserError(

f"Duplicate tool names found across MCP servers: "

f"{server\_tool\_names & tool\_names}"

)

tool\_names.update(server\_tool\_names)

tools.extend(server\_tools)

return tools

get\_function\_tools async classmethod

get\_function\_tools(server: MCPServer) -> list[Tool]

Get all function tools from a single MCP server.

Source code in src/agents/mcp/util.py

@classmethod

async def get\_function\_tools(cls, server: "MCPServer") -> list[Tool]:

"""Get all function tools from a single MCP server."""

with mcp\_tools\_span(server=server.name) as span:

tools = await server.list\_tools()

span.span\_data.result = [tool.name for tool in tools]

return [cls.to\_function\_tool(tool, server) for tool in tools]

to\_function\_tool classmethod

to\_function\_tool(

tool: Tool, server: MCPServer

) -> FunctionTool

Convert an MCP tool to an Agents SDK function tool.

Source code in src/agents/mcp/util.py

@classmethod

def to\_function\_tool(cls, tool: "MCPTool", server: "MCPServer") -> FunctionTool:

"""Convert an MCP tool to an Agents SDK function tool."""

invoke\_func = functools.partial(cls.invoke\_mcp\_tool, server, tool)

return FunctionTool(

name=tool.name,

description=tool.description or "",

params\_json\_schema=tool.inputSchema,

on\_invoke\_tool=invoke\_func,

strict\_json\_schema=False,

)

invoke\_mcp\_tool async classmethod

invoke\_mcp\_tool(

server: MCPServer,

tool: Tool,

context: RunContextWrapper[Any],

input\_json: str,

) -> str

Invoke an MCP tool and return the result as a string.

Source code in src/agents/mcp/util.py

@classmethod

async def invoke\_mcp\_tool(

cls, server: "MCPServer", tool: "MCPTool", context: RunContextWrapper[Any], input\_json: str

) -> str:

"""Invoke an MCP tool and return the result as a string."""

try:

json\_data: dict[str, Any] = json.loads(input\_json) if input\_json else {}

except Exception as e:

if \_debug.DONT\_LOG\_TOOL\_DATA:

logger.debug(f"Invalid JSON input for tool {tool.name}")

else:

logger.debug(f"Invalid JSON input for tool {tool.name}: {input\_json}")

raise ModelBehaviorError(

f"Invalid JSON input for tool {tool.name}: {input\_json}"

) from e

if \_debug.DONT\_LOG\_TOOL\_DATA:

logger.debug(f"Invoking MCP tool {tool.name}")

else:

logger.debug(f"Invoking MCP tool {tool.name} with input {input\_json}")

try:

result = await server.call\_tool(tool.name, json\_data)

except Exception as e:

logger.error(f"Error invoking MCP tool {tool.name}: {e}")

raise AgentsException(f"Error invoking MCP tool {tool.name}: {e}") from e

if \_debug.DONT\_LOG\_TOOL\_DATA:

logger.debug(f"MCP tool {tool.name} completed.")

else:

logger.debug(f"MCP tool {tool.name} returned {result}")

# The MCP tool result is a list of content items, whereas OpenAI tool outputs are a single

# string. We'll try to convert.

if len(result.content) == 1:

tool\_output = result.content[0].model\_dump\_json()

elif len(result.content) > 1:

tool\_output = json.dumps([item.model\_dump() for item in result.content])

else:

logger.error(f"Errored MCP tool result: {result}")

tool\_output = "Error running tool."

current\_span = get\_current\_span()

if current\_span:

if isinstance(current\_span.span\_data, FunctionSpanData):

current\_span.span\_data.output = tool\_output

current\_span.span\_data.mcp\_data = {

"server": server.name,

}

else:

logger.warning(

f"Current span is not a FunctionSpanData, skipping tool output: {current\_span}"

)

return tool\_output

Tracing:

Tracing module

TracingProcessor

Bases: ABC

Interface for processing spans.

Source code in src/agents/tracing/processor\_interface.py

class TracingProcessor(abc.ABC):

"""Interface for processing spans."""

@abc.abstractmethod

def on\_trace\_start(self, trace: "Trace") -> None:

"""Called when a trace is started.

Args:

trace: The trace that started.

"""

pass

@abc.abstractmethod

def on\_trace\_end(self, trace: "Trace") -> None:

"""Called when a trace is finished.

Args:

trace: The trace that started.

"""

pass

@abc.abstractmethod

def on\_span\_start(self, span: "Span[Any]") -> None:

"""Called when a span is started.

Args:

span: The span that started.

"""

pass

@abc.abstractmethod

def on\_span\_end(self, span: "Span[Any]") -> None:

"""Called when a span is finished. Should not block or raise exceptions.

Args:

span: The span that finished.

"""

pass

@abc.abstractmethod

def shutdown(self) -> None:

"""Called when the application stops."""

pass

@abc.abstractmethod

def force\_flush(self) -> None:

"""Forces an immediate flush of all queued spans/traces."""

pass

on\_trace\_start abstractmethod

on\_trace\_start(trace: Trace) -> None

Called when a trace is started.

Parameters:

Name Type Description Default

trace Trace The trace that started. required

Source code in src/agents/tracing/processor\_interface.py

@abc.abstractmethod

def on\_trace\_start(self, trace: "Trace") -> None:

"""Called when a trace is started.

Args:

trace: The trace that started.

"""

pass

on\_trace\_end abstractmethod

on\_trace\_end(trace: Trace) -> None

Called when a trace is finished.

Parameters:

Name Type Description Default

trace Trace The trace that started. required

Source code in src/agents/tracing/processor\_interface.py

@abc.abstractmethod

def on\_trace\_end(self, trace: "Trace") -> None:

"""Called when a trace is finished.

Args:

trace: The trace that started.

"""

pass

on\_span\_start abstractmethod

on\_span\_start(span: Span[Any]) -> None

Called when a span is started.

Parameters:

Name Type Description Default

span Span[Any] The span that started. required

Source code in src/agents/tracing/processor\_interface.py

@abc.abstractmethod

def on\_span\_start(self, span: "Span[Any]") -> None:

"""Called when a span is started.

Args:

span: The span that started.

"""

pass

on\_span\_end abstractmethod

on\_span\_end(span: Span[Any]) -> None

Called when a span is finished. Should not block or raise exceptions.

Parameters:

Name Type Description Default

span Span[Any] The span that finished. required

Source code in src/agents/tracing/processor\_interface.py

@abc.abstractmethod

def on\_span\_end(self, span: "Span[Any]") -> None:

"""Called when a span is finished. Should not block or raise exceptions.

Args:

span: The span that finished.

"""

pass

shutdown abstractmethod

shutdown() -> None

Called when the application stops.

Source code in src/agents/tracing/processor\_interface.py

@abc.abstractmethod

def shutdown(self) -> None:

"""Called when the application stops."""

pass

force\_flush abstractmethod

force\_flush() -> None

Forces an immediate flush of all queued spans/traces.

Source code in src/agents/tracing/processor\_interface.py

@abc.abstractmethod

def force\_flush(self) -> None:

"""Forces an immediate flush of all queued spans/traces."""

pass

Span

Bases: ABC, Generic[TSpanData]

Source code in src/agents/tracing/spans.py

class Span(abc.ABC, Generic[TSpanData]):

@property

@abc.abstractmethod

def trace\_id(self) -> str:

pass

@property

@abc.abstractmethod

def span\_id(self) -> str:

pass

@property

@abc.abstractmethod

def span\_data(self) -> TSpanData:

pass

@abc.abstractmethod

def start(self, mark\_as\_current: bool = False):

"""

Start the span.

Args:

mark\_as\_current: If true, the span will be marked as the current span.

"""

pass

@abc.abstractmethod

def finish(self, reset\_current: bool = False) -> None:

"""

Finish the span.

Args:

reset\_current: If true, the span will be reset as the current span.

"""

pass

@abc.abstractmethod

def \_\_enter\_\_(self) -> Span[TSpanData]:

pass

@abc.abstractmethod

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

pass

@property

@abc.abstractmethod

def parent\_id(self) -> str | None:

pass

@abc.abstractmethod

def set\_error(self, error: SpanError) -> None:

pass

@property

@abc.abstractmethod

def error(self) -> SpanError | None:

pass

@abc.abstractmethod

def export(self) -> dict[str, Any] | None:

pass

@property

@abc.abstractmethod

def started\_at(self) -> str | None:

pass

@property

@abc.abstractmethod

def ended\_at(self) -> str | None:

pass

start abstractmethod

start(mark\_as\_current: bool = False)

Start the span.

Parameters:

Name Type Description Default

mark\_as\_current bool If true, the span will be marked as the current span. False

Source code in src/agents/tracing/spans.py

@abc.abstractmethod

def start(self, mark\_as\_current: bool = False):

"""

Start the span.

Args:

mark\_as\_current: If true, the span will be marked as the current span.

"""

pass

finish abstractmethod

finish(reset\_current: bool = False) -> None

Finish the span.

Parameters:

Name Type Description Default

reset\_current bool If true, the span will be reset as the current span. False

Source code in src/agents/tracing/spans.py

@abc.abstractmethod

def finish(self, reset\_current: bool = False) -> None:

"""

Finish the span.

Args:

reset\_current: If true, the span will be reset as the current span.

"""

pass

Trace

A trace is the root level object that tracing creates. It represents a logical "workflow".

Source code in src/agents/tracing/traces.py

class Trace:

"""

A trace is the root level object that tracing creates. It represents a logical "workflow".

"""

@abc.abstractmethod

def \_\_enter\_\_(self) -> Trace:

pass

@abc.abstractmethod

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

pass

@abc.abstractmethod

def start(self, mark\_as\_current: bool = False):

"""

Start the trace.

Args:

mark\_as\_current: If true, the trace will be marked as the current trace.

"""

pass

@abc.abstractmethod

def finish(self, reset\_current: bool = False):

"""

Finish the trace.

Args:

reset\_current: If true, the trace will be reset as the current trace.

"""

pass

@property

@abc.abstractmethod

def trace\_id(self) -> str:

"""

The trace ID.

"""

pass

@property

@abc.abstractmethod

def name(self) -> str:

"""

The name of the workflow being traced.

"""

pass

@abc.abstractmethod

def export(self) -> dict[str, Any] | None:

"""

Export the trace as a dictionary.

"""

pass

trace\_id abstractmethod property

trace\_id: str

The trace ID.

name abstractmethod property

name: str

The name of the workflow being traced.

start abstractmethod

start(mark\_as\_current: bool = False)

Start the trace.

Parameters:

Name Type Description Default

mark\_as\_current bool If true, the trace will be marked as the current trace. False

Source code in src/agents/tracing/traces.py

@abc.abstractmethod

def start(self, mark\_as\_current: bool = False):

"""

Start the trace.

Args:

mark\_as\_current: If true, the trace will be marked as the current trace.

"""

pass

finish abstractmethod

finish(reset\_current: bool = False)

Finish the trace.

Parameters:

Name Type Description Default

reset\_current bool If true, the trace will be reset as the current trace. False

Source code in src/agents/tracing/traces.py

@abc.abstractmethod

def finish(self, reset\_current: bool = False):

"""

Finish the trace.

Args:

reset\_current: If true, the trace will be reset as the current trace.

"""

pass

export abstractmethod

export() -> dict[str, Any] | None

Export the trace as a dictionary.

Source code in src/agents/tracing/traces.py

@abc.abstractmethod

def export(self) -> dict[str, Any] | None:

"""

Export the trace as a dictionary.

"""

pass

agent\_span

agent\_span(

name: str,

handoffs: list[str] | None = None,

tools: list[str] | None = None,

output\_type: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[AgentSpanData]

Create a new agent span. The span will not be started automatically, you should either do with agent\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

name str The name of the agent. required

handoffs list[str] | None Optional list of agent names to which this agent could hand off control. None

tools list[str] | None Optional list of tool names available to this agent. None

output\_type str | None Optional name of the output type produced by the agent. None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Returns:

Type Description

Span[AgentSpanData] The newly created agent span.

Source code in src/agents/tracing/create.py

def agent\_span(

name: str,

handoffs: list[str] | None = None,

tools: list[str] | None = None,

output\_type: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[AgentSpanData]:

"""Create a new agent span. The span will not be started automatically, you should either do

`with agent\_span() ...` or call `span.start()` + `span.finish()` manually.

Args:

name: The name of the agent.

handoffs: Optional list of agent names to which this agent could hand off control.

tools: Optional list of tool names available to this agent.

output\_type: Optional name of the output type produced by the agent.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

Returns:

The newly created agent span.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=AgentSpanData(name=name, handoffs=handoffs, tools=tools, output\_type=output\_type),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

custom\_span

custom\_span(

name: str,

data: dict[str, Any] | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[CustomSpanData]

Create a new custom span, to which you can add your own metadata. The span will not be started automatically, you should either do with custom\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

name str The name of the custom span. required

data dict[str, Any] | None Arbitrary structured data to associate with the span. None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Returns:

Type Description

Span[CustomSpanData] The newly created custom span.

Source code in src/agents/tracing/create.py

def custom\_span(

name: str,

data: dict[str, Any] | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[CustomSpanData]:

"""Create a new custom span, to which you can add your own metadata. The span will not be

started automatically, you should either do `with custom\_span() ...` or call

`span.start()` + `span.finish()` manually.

Args:

name: The name of the custom span.

data: Arbitrary structured data to associate with the span.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

Returns:

The newly created custom span.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=CustomSpanData(name=name, data=data or {}),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

function\_span

function\_span(

name: str,

input: str | None = None,

output: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[FunctionSpanData]

Create a new function span. The span will not be started automatically, you should either do with function\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

name str The name of the function. required

input str | None The input to the function. None

output str | None The output of the function. None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Returns:

Type Description

Span[FunctionSpanData] The newly created function span.

Source code in src/agents/tracing/create.py

def function\_span(

name: str,

input: str | None = None,

output: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[FunctionSpanData]:

"""Create a new function span. The span will not be started automatically, you should either do

`with function\_span() ...` or call `span.start()` + `span.finish()` manually.

Args:

name: The name of the function.

input: The input to the function.

output: The output of the function.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

Returns:

The newly created function span.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=FunctionSpanData(name=name, input=input, output=output),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

generation\_span

generation\_span(

input: Sequence[Mapping[str, Any]] | None = None,

output: Sequence[Mapping[str, Any]] | None = None,

model: str | None = None,

model\_config: Mapping[str, Any] | None = None,

usage: dict[str, Any] | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[GenerationSpanData]

Create a new generation span. The span will not be started automatically, you should either do with generation\_span() ... or call span.start() + span.finish() manually.

This span captures the details of a model generation, including the input message sequence, any generated outputs, the model name and configuration, and usage data. If you only need to capture a model response identifier, use response\_span() instead.

Parameters:

Name Type Description Default

input Sequence[Mapping[str, Any]] | None The sequence of input messages sent to the model. None

output Sequence[Mapping[str, Any]] | None The sequence of output messages received from the model. None

model str | None The model identifier used for the generation. None

model\_config Mapping[str, Any] | None The model configuration (hyperparameters) used. None

usage dict[str, Any] | None A dictionary of usage information (input tokens, output tokens, etc.). None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Returns:

Type Description

Span[GenerationSpanData] The newly created generation span.

Source code in src/agents/tracing/create.py

def generation\_span(

input: Sequence[Mapping[str, Any]] | None = None,

output: Sequence[Mapping[str, Any]] | None = None,

model: str | None = None,

model\_config: Mapping[str, Any] | None = None,

usage: dict[str, Any] | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[GenerationSpanData]:

"""Create a new generation span. The span will not be started automatically, you should either

do `with generation\_span() ...` or call `span.start()` + `span.finish()` manually.

This span captures the details of a model generation, including the

input message sequence, any generated outputs, the model name and

configuration, and usage data. If you only need to capture a model

response identifier, use `response\_span()` instead.

Args:

input: The sequence of input messages sent to the model.

output: The sequence of output messages received from the model.

model: The model identifier used for the generation.

model\_config: The model configuration (hyperparameters) used.

usage: A dictionary of usage information (input tokens, output tokens, etc.).

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

Returns:

The newly created generation span.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=GenerationSpanData(

input=input,

output=output,

model=model,

model\_config=model\_config,

usage=usage,

),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

get\_current\_span

get\_current\_span() -> Span[Any] | None

Returns the currently active span, if present.

Source code in src/agents/tracing/create.py

def get\_current\_span() -> Span[Any] | None:

"""Returns the currently active span, if present."""

return GLOBAL\_TRACE\_PROVIDER.get\_current\_span()

get\_current\_trace

get\_current\_trace() -> Trace | None

Returns the currently active trace, if present.

Source code in src/agents/tracing/create.py

def get\_current\_trace() -> Trace | None:

"""Returns the currently active trace, if present."""

return GLOBAL\_TRACE\_PROVIDER.get\_current\_trace()

guardrail\_span

guardrail\_span(

name: str,

triggered: bool = False,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[GuardrailSpanData]

Create a new guardrail span. The span will not be started automatically, you should either do with guardrail\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

name str The name of the guardrail. required

triggered bool Whether the guardrail was triggered. False

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Source code in src/agents/tracing/create.py

def guardrail\_span(

name: str,

triggered: bool = False,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[GuardrailSpanData]:

"""Create a new guardrail span. The span will not be started automatically, you should either

do `with guardrail\_span() ...` or call `span.start()` + `span.finish()` manually.

Args:

name: The name of the guardrail.

triggered: Whether the guardrail was triggered.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=GuardrailSpanData(name=name, triggered=triggered),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

handoff\_span

handoff\_span(

from\_agent: str | None = None,

to\_agent: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[HandoffSpanData]

Create a new handoff span. The span will not be started automatically, you should either do with handoff\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

from\_agent str | None The name of the agent that is handing off. None

to\_agent str | None The name of the agent that is receiving the handoff. None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Returns:

Type Description

Span[HandoffSpanData] The newly created handoff span.

Source code in src/agents/tracing/create.py

def handoff\_span(

from\_agent: str | None = None,

to\_agent: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[HandoffSpanData]:

"""Create a new handoff span. The span will not be started automatically, you should either do

`with handoff\_span() ...` or call `span.start()` + `span.finish()` manually.

Args:

from\_agent: The name of the agent that is handing off.

to\_agent: The name of the agent that is receiving the handoff.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

Returns:

The newly created handoff span.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=HandoffSpanData(from\_agent=from\_agent, to\_agent=to\_agent),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

mcp\_tools\_span

mcp\_tools\_span(

server: str | None = None,

result: list[str] | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[MCPListToolsSpanData]

Create a new MCP list tools span. The span will not be started automatically, you should either do with mcp\_tools\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

server str | None The name of the MCP server. None

result list[str] | None The result of the MCP list tools call. None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Source code in src/agents/tracing/create.py

def mcp\_tools\_span(

server: str | None = None,

result: list[str] | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[MCPListToolsSpanData]:

"""Create a new MCP list tools span. The span will not be started automatically, you should

either do `with mcp\_tools\_span() ...` or call `span.start()` + `span.finish()` manually.

Args:

server: The name of the MCP server.

result: The result of the MCP list tools call.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=MCPListToolsSpanData(server=server, result=result),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

response\_span

response\_span(

response: Response | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[ResponseSpanData]

Create a new response span. The span will not be started automatically, you should either do with response\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

response Response | None The OpenAI Response object. None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Source code in src/agents/tracing/create.py

def response\_span(

response: Response | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[ResponseSpanData]:

"""Create a new response span. The span will not be started automatically, you should either do

`with response\_span() ...` or call `span.start()` + `span.finish()` manually.

Args:

response: The OpenAI Response object.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=ResponseSpanData(response=response),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

speech\_group\_span

speech\_group\_span(

input: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[SpeechGroupSpanData]

Create a new speech group span. The span will not be started automatically, you should either do with speech\_group\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

input str | None The input text used for the speech request. None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Source code in src/agents/tracing/create.py

def speech\_group\_span(

input: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[SpeechGroupSpanData]:

"""Create a new speech group span. The span will not be started automatically, you should

either do `with speech\_group\_span() ...` or call `span.start()` + `span.finish()` manually.

Args:

input: The input text used for the speech request.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=SpeechGroupSpanData(input=input),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

speech\_span

speech\_span(

model: str | None = None,

input: str | None = None,

output: str | None = None,

output\_format: str | None = "pcm",

model\_config: Mapping[str, Any] | None = None,

first\_content\_at: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[SpeechSpanData]

Create a new speech span. The span will not be started automatically, you should either do with speech\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

model str | None The name of the model used for the text-to-speech. None

input str | None The text input of the text-to-speech. None

output str | None The audio output of the text-to-speech as base64 encoded string of PCM audio bytes. None

output\_format str | None The format of the audio output (defaults to "pcm"). 'pcm'

model\_config Mapping[str, Any] | None The model configuration (hyperparameters) used. None

first\_content\_at str | None The time of the first byte of the audio output. None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Source code in src/agents/tracing/create.py

def speech\_span(

model: str | None = None,

input: str | None = None,

output: str | None = None,

output\_format: str | None = "pcm",

model\_config: Mapping[str, Any] | None = None,

first\_content\_at: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[SpeechSpanData]:

"""Create a new speech span. The span will not be started automatically, you should either do

`with speech\_span() ...` or call `span.start()` + `span.finish()` manually.

Args:

model: The name of the model used for the text-to-speech.

input: The text input of the text-to-speech.

output: The audio output of the text-to-speech as base64 encoded string of PCM audio bytes.

output\_format: The format of the audio output (defaults to "pcm").

model\_config: The model configuration (hyperparameters) used.

first\_content\_at: The time of the first byte of the audio output.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=SpeechSpanData(

model=model,

input=input,

output=output,

output\_format=output\_format,

model\_config=model\_config,

first\_content\_at=first\_content\_at,

),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

trace

trace(

workflow\_name: str,

trace\_id: str | None = None,

group\_id: str | None = None,

metadata: dict[str, Any] | None = None,

disabled: bool = False,

) -> Trace

Create a new trace. The trace will not be started automatically; you should either use it as a context manager (with trace(...):) or call trace.start() + trace.finish() manually.

In addition to the workflow name and optional grouping identifier, you can provide an arbitrary metadata dictionary to attach additional user-defined information to the trace.

Parameters:

Name Type Description Default

workflow\_name str The name of the logical app or workflow. For example, you might provide "code\_bot" for a coding agent, or "customer\_support\_agent" for a customer support agent. required

trace\_id str | None The ID of the trace. Optional. If not provided, we will generate an ID. We recommend using util.gen\_trace\_id() to generate a trace ID, to guarantee that IDs are correctly formatted. None

group\_id str | None Optional grouping identifier to link multiple traces from the same conversation or process. For instance, you might use a chat thread ID. None

metadata dict[str, Any] | None Optional dictionary of additional metadata to attach to the trace. None

disabled bool If True, we will return a Trace but the Trace will not be recorded. This will not be checked if there's an existing trace and even\_if\_trace\_running is True. False

Returns:

Type Description

Trace The newly created trace object.

Source code in src/agents/tracing/create.py

def trace(

workflow\_name: str,

trace\_id: str | None = None,

group\_id: str | None = None,

metadata: dict[str, Any] | None = None,

disabled: bool = False,

) -> Trace:

"""

Create a new trace. The trace will not be started automatically; you should either use

it as a context manager (`with trace(...):`) or call `trace.start()` + `trace.finish()`

manually.

In addition to the workflow name and optional grouping identifier, you can provide

an arbitrary metadata dictionary to attach additional user-defined information to

the trace.

Args:

workflow\_name: The name of the logical app or workflow. For example, you might provide

"code\_bot" for a coding agent, or "customer\_support\_agent" for a customer support agent.

trace\_id: The ID of the trace. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_trace\_id()` to generate a trace ID, to guarantee that IDs are

correctly formatted.

group\_id: Optional grouping identifier to link multiple traces from the same conversation

or process. For instance, you might use a chat thread ID.

metadata: Optional dictionary of additional metadata to attach to the trace.

disabled: If True, we will return a Trace but the Trace will not be recorded. This will

not be checked if there's an existing trace and `even\_if\_trace\_running` is True.

Returns:

The newly created trace object.

"""

current\_trace = GLOBAL\_TRACE\_PROVIDER.get\_current\_trace()

if current\_trace:

logger.warning(

"Trace already exists. Creating a new trace, but this is probably a mistake."

)

return GLOBAL\_TRACE\_PROVIDER.create\_trace(

name=workflow\_name,

trace\_id=trace\_id,

group\_id=group\_id,

metadata=metadata,

disabled=disabled,

)

transcription\_span

transcription\_span(

model: str | None = None,

input: str | None = None,

input\_format: str | None = "pcm",

output: str | None = None,

model\_config: Mapping[str, Any] | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[TranscriptionSpanData]

Create a new transcription span. The span will not be started automatically, you should either do with transcription\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

model str | None The name of the model used for the speech-to-text. None

input str | None The audio input of the speech-to-text transcription, as a base64 encoded string of audio bytes. None

input\_format str | None The format of the audio input (defaults to "pcm"). 'pcm'

output str | None The output of the speech-to-text transcription. None

model\_config Mapping[str, Any] | None The model configuration (hyperparameters) used. None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Returns:

Type Description

Span[TranscriptionSpanData] The newly created speech-to-text span.

Source code in src/agents/tracing/create.py

def transcription\_span(

model: str | None = None,

input: str | None = None,

input\_format: str | None = "pcm",

output: str | None = None,

model\_config: Mapping[str, Any] | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[TranscriptionSpanData]:

"""Create a new transcription span. The span will not be started automatically, you should

either do `with transcription\_span() ...` or call `span.start()` + `span.finish()` manually.

Args:

model: The name of the model used for the speech-to-text.

input: The audio input of the speech-to-text transcription, as a base64 encoded string of

audio bytes.

input\_format: The format of the audio input (defaults to "pcm").

output: The output of the speech-to-text transcription.

model\_config: The model configuration (hyperparameters) used.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

Returns:

The newly created speech-to-text span.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=TranscriptionSpanData(

input=input,

input\_format=input\_format,

output=output,

model=model,

model\_config=model\_config,

),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

gen\_span\_id

gen\_span\_id() -> str

Generates a new span ID.

Source code in src/agents/tracing/util.py

def gen\_span\_id() -> str:

"""Generates a new span ID."""

return f"span\_{uuid.uuid4().hex[:24]}"

gen\_trace\_id

gen\_trace\_id() -> str

Generates a new trace ID.

Source code in src/agents/tracing/util.py

def gen\_trace\_id() -> str:

"""Generates a new trace ID."""

return f"trace\_{uuid.uuid4().hex}"

add\_trace\_processor

add\_trace\_processor(

span\_processor: TracingProcessor,

) -> None

Adds a new trace processor. This processor will receive all traces/spans.

Source code in src/agents/tracing/\_\_init\_\_.py

def add\_trace\_processor(span\_processor: TracingProcessor) -> None:

"""

Adds a new trace processor. This processor will receive all traces/spans.

"""

GLOBAL\_TRACE\_PROVIDER.register\_processor(span\_processor)

set\_trace\_processors

set\_trace\_processors(

processors: list[TracingProcessor],

) -> None

Set the list of trace processors. This will replace the current list of processors.

Source code in src/agents/tracing/\_\_init\_\_.py

def set\_trace\_processors(processors: list[TracingProcessor]) -> None:

"""

Set the list of trace processors. This will replace the current list of processors.

"""

GLOBAL\_TRACE\_PROVIDER.set\_processors(processors)

set\_tracing\_disabled

set\_tracing\_disabled(disabled: bool) -> None

Set whether tracing is globally disabled.

Source code in src/agents/tracing/\_\_init\_\_.py

def set\_tracing\_disabled(disabled: bool) -> None:

"""

Set whether tracing is globally disabled.

"""

GLOBAL\_TRACE\_PROVIDER.set\_disabled(disabled)

set\_tracing\_export\_api\_key

set\_tracing\_export\_api\_key(api\_key: str) -> None

Set the OpenAI API key for the backend exporter.

Source code in src/agents/tracing/\_\_init\_\_.py

def set\_tracing\_export\_api\_key(api\_key: str) -> None:

"""

Set the OpenAI API key for the backend exporter.

"""

default\_exporter().set\_api\_key(api\_key)

Creating traces/spans

trace

trace(

workflow\_name: str,

trace\_id: str | None = None,

group\_id: str | None = None,

metadata: dict[str, Any] | None = None,

disabled: bool = False,

) -> Trace

Create a new trace. The trace will not be started automatically; you should either use it as a context manager (with trace(...):) or call trace.start() + trace.finish() manually.

In addition to the workflow name and optional grouping identifier, you can provide an arbitrary metadata dictionary to attach additional user-defined information to the trace.

Parameters:

Name Type Description Default

workflow\_name str The name of the logical app or workflow. For example, you might provide "code\_bot" for a coding agent, or "customer\_support\_agent" for a customer support agent. required

trace\_id str | None The ID of the trace. Optional. If not provided, we will generate an ID. We recommend using util.gen\_trace\_id() to generate a trace ID, to guarantee that IDs are correctly formatted. None

group\_id str | None Optional grouping identifier to link multiple traces from the same conversation or process. For instance, you might use a chat thread ID. None

metadata dict[str, Any] | None Optional dictionary of additional metadata to attach to the trace. None

disabled bool If True, we will return a Trace but the Trace will not be recorded. This will not be checked if there's an existing trace and even\_if\_trace\_running is True. False

Returns:

Type Description

Trace The newly created trace object.

Source code in src/agents/tracing/create.py

def trace(

workflow\_name: str,

trace\_id: str | None = None,

group\_id: str | None = None,

metadata: dict[str, Any] | None = None,

disabled: bool = False,

) -> Trace:

"""

Create a new trace. The trace will not be started automatically; you should either use

it as a context manager (`with trace(...):`) or call `trace.start()` + `trace.finish()`

manually.

In addition to the workflow name and optional grouping identifier, you can provide

an arbitrary metadata dictionary to attach additional user-defined information to

the trace.

Args:

workflow\_name: The name of the logical app or workflow. For example, you might provide

"code\_bot" for a coding agent, or "customer\_support\_agent" for a customer support agent.

trace\_id: The ID of the trace. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_trace\_id()` to generate a trace ID, to guarantee that IDs are

correctly formatted.

group\_id: Optional grouping identifier to link multiple traces from the same conversation

or process. For instance, you might use a chat thread ID.

metadata: Optional dictionary of additional metadata to attach to the trace.

disabled: If True, we will return a Trace but the Trace will not be recorded. This will

not be checked if there's an existing trace and `even\_if\_trace\_running` is True.

Returns:

The newly created trace object.

"""

current\_trace = GLOBAL\_TRACE\_PROVIDER.get\_current\_trace()

if current\_trace:

logger.warning(

"Trace already exists. Creating a new trace, but this is probably a mistake."

)

return GLOBAL\_TRACE\_PROVIDER.create\_trace(

name=workflow\_name,

trace\_id=trace\_id,

group\_id=group\_id,

metadata=metadata,

disabled=disabled,

)

get\_current\_trace

get\_current\_trace() -> Trace | None

Returns the currently active trace, if present.

Source code in src/agents/tracing/create.py

def get\_current\_trace() -> Trace | None:

"""Returns the currently active trace, if present."""

return GLOBAL\_TRACE\_PROVIDER.get\_current\_trace()

get\_current\_span

get\_current\_span() -> Span[Any] | None

Returns the currently active span, if present.

Source code in src/agents/tracing/create.py

def get\_current\_span() -> Span[Any] | None:

"""Returns the currently active span, if present."""

return GLOBAL\_TRACE\_PROVIDER.get\_current\_span()

agent\_span

agent\_span(

name: str,

handoffs: list[str] | None = None,

tools: list[str] | None = None,

output\_type: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[AgentSpanData]

Create a new agent span. The span will not be started automatically, you should either do with agent\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

name str The name of the agent. required

handoffs list[str] | None Optional list of agent names to which this agent could hand off control. None

tools list[str] | None Optional list of tool names available to this agent. None

output\_type str | None Optional name of the output type produced by the agent. None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Returns:

Type Description

Span[AgentSpanData] The newly created agent span.

Source code in src/agents/tracing/create.py

def agent\_span(

name: str,

handoffs: list[str] | None = None,

tools: list[str] | None = None,

output\_type: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[AgentSpanData]:

"""Create a new agent span. The span will not be started automatically, you should either do

`with agent\_span() ...` or call `span.start()` + `span.finish()` manually.

Args:

name: The name of the agent.

handoffs: Optional list of agent names to which this agent could hand off control.

tools: Optional list of tool names available to this agent.

output\_type: Optional name of the output type produced by the agent.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

Returns:

The newly created agent span.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=AgentSpanData(name=name, handoffs=handoffs, tools=tools, output\_type=output\_type),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

function\_span

function\_span(

name: str,

input: str | None = None,

output: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[FunctionSpanData]

Create a new function span. The span will not be started automatically, you should either do with function\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

name str The name of the function. required

input str | None The input to the function. None

output str | None The output of the function. None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Returns:

Type Description

Span[FunctionSpanData] The newly created function span.

Source code in src/agents/tracing/create.py

def function\_span(

name: str,

input: str | None = None,

output: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[FunctionSpanData]:

"""Create a new function span. The span will not be started automatically, you should either do

`with function\_span() ...` or call `span.start()` + `span.finish()` manually.

Args:

name: The name of the function.

input: The input to the function.

output: The output of the function.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

Returns:

The newly created function span.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=FunctionSpanData(name=name, input=input, output=output),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

generation\_span

generation\_span(

input: Sequence[Mapping[str, Any]] | None = None,

output: Sequence[Mapping[str, Any]] | None = None,

model: str | None = None,

model\_config: Mapping[str, Any] | None = None,

usage: dict[str, Any] | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[GenerationSpanData]

Create a new generation span. The span will not be started automatically, you should either do with generation\_span() ... or call span.start() + span.finish() manually.

This span captures the details of a model generation, including the input message sequence, any generated outputs, the model name and configuration, and usage data. If you only need to capture a model response identifier, use response\_span() instead.

Parameters:

Name Type Description Default

input Sequence[Mapping[str, Any]] | None The sequence of input messages sent to the model. None

output Sequence[Mapping[str, Any]] | None The sequence of output messages received from the model. None

model str | None The model identifier used for the generation. None

model\_config Mapping[str, Any] | None The model configuration (hyperparameters) used. None

usage dict[str, Any] | None A dictionary of usage information (input tokens, output tokens, etc.). None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Returns:

Type Description

Span[GenerationSpanData] The newly created generation span.

Source code in src/agents/tracing/create.py

def generation\_span(

input: Sequence[Mapping[str, Any]] | None = None,

output: Sequence[Mapping[str, Any]] | None = None,

model: str | None = None,

model\_config: Mapping[str, Any] | None = None,

usage: dict[str, Any] | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[GenerationSpanData]:

"""Create a new generation span. The span will not be started automatically, you should either

do `with generation\_span() ...` or call `span.start()` + `span.finish()` manually.

This span captures the details of a model generation, including the

input message sequence, any generated outputs, the model name and

configuration, and usage data. If you only need to capture a model

response identifier, use `response\_span()` instead.

Args:

input: The sequence of input messages sent to the model.

output: The sequence of output messages received from the model.

model: The model identifier used for the generation.

model\_config: The model configuration (hyperparameters) used.

usage: A dictionary of usage information (input tokens, output tokens, etc.).

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

Returns:

The newly created generation span.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=GenerationSpanData(

input=input,

output=output,

model=model,

model\_config=model\_config,

usage=usage,

),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

response\_span

response\_span(

response: Response | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[ResponseSpanData]

Create a new response span. The span will not be started automatically, you should either do with response\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

response Response | None The OpenAI Response object. None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Source code in src/agents/tracing/create.py

def response\_span(

response: Response | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[ResponseSpanData]:

"""Create a new response span. The span will not be started automatically, you should either do

`with response\_span() ...` or call `span.start()` + `span.finish()` manually.

Args:

response: The OpenAI Response object.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=ResponseSpanData(response=response),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

handoff\_span

handoff\_span(

from\_agent: str | None = None,

to\_agent: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[HandoffSpanData]

Create a new handoff span. The span will not be started automatically, you should either do with handoff\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

from\_agent str | None The name of the agent that is handing off. None

to\_agent str | None The name of the agent that is receiving the handoff. None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Returns:

Type Description

Span[HandoffSpanData] The newly created handoff span.

Source code in src/agents/tracing/create.py

def handoff\_span(

from\_agent: str | None = None,

to\_agent: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[HandoffSpanData]:

"""Create a new handoff span. The span will not be started automatically, you should either do

`with handoff\_span() ...` or call `span.start()` + `span.finish()` manually.

Args:

from\_agent: The name of the agent that is handing off.

to\_agent: The name of the agent that is receiving the handoff.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

Returns:

The newly created handoff span.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=HandoffSpanData(from\_agent=from\_agent, to\_agent=to\_agent),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

custom\_span

custom\_span(

name: str,

data: dict[str, Any] | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[CustomSpanData]

Create a new custom span, to which you can add your own metadata. The span will not be started automatically, you should either do with custom\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

name str The name of the custom span. required

data dict[str, Any] | None Arbitrary structured data to associate with the span. None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Returns:

Type Description

Span[CustomSpanData] The newly created custom span.

Source code in src/agents/tracing/create.py

def custom\_span(

name: str,

data: dict[str, Any] | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[CustomSpanData]:

"""Create a new custom span, to which you can add your own metadata. The span will not be

started automatically, you should either do `with custom\_span() ...` or call

`span.start()` + `span.finish()` manually.

Args:

name: The name of the custom span.

data: Arbitrary structured data to associate with the span.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

Returns:

The newly created custom span.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=CustomSpanData(name=name, data=data or {}),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

guardrail\_span

guardrail\_span(

name: str,

triggered: bool = False,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[GuardrailSpanData]

Create a new guardrail span. The span will not be started automatically, you should either do with guardrail\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

name str The name of the guardrail. required

triggered bool Whether the guardrail was triggered. False

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Source code in src/agents/tracing/create.py

def guardrail\_span(

name: str,

triggered: bool = False,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[GuardrailSpanData]:

"""Create a new guardrail span. The span will not be started automatically, you should either

do `with guardrail\_span() ...` or call `span.start()` + `span.finish()` manually.

Args:

name: The name of the guardrail.

triggered: Whether the guardrail was triggered.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=GuardrailSpanData(name=name, triggered=triggered),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

transcription\_span

transcription\_span(

model: str | None = None,

input: str | None = None,

input\_format: str | None = "pcm",

output: str | None = None,

model\_config: Mapping[str, Any] | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[TranscriptionSpanData]

Create a new transcription span. The span will not be started automatically, you should either do with transcription\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

model str | None The name of the model used for the speech-to-text. None

input str | None The audio input of the speech-to-text transcription, as a base64 encoded string of audio bytes. None

input\_format str | None The format of the audio input (defaults to "pcm"). 'pcm'

output str | None The output of the speech-to-text transcription. None

model\_config Mapping[str, Any] | None The model configuration (hyperparameters) used. None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Returns:

Type Description

Span[TranscriptionSpanData] The newly created speech-to-text span.

Source code in src/agents/tracing/create.py

def transcription\_span(

model: str | None = None,

input: str | None = None,

input\_format: str | None = "pcm",

output: str | None = None,

model\_config: Mapping[str, Any] | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[TranscriptionSpanData]:

"""Create a new transcription span. The span will not be started automatically, you should

either do `with transcription\_span() ...` or call `span.start()` + `span.finish()` manually.

Args:

model: The name of the model used for the speech-to-text.

input: The audio input of the speech-to-text transcription, as a base64 encoded string of

audio bytes.

input\_format: The format of the audio input (defaults to "pcm").

output: The output of the speech-to-text transcription.

model\_config: The model configuration (hyperparameters) used.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

Returns:

The newly created speech-to-text span.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=TranscriptionSpanData(

input=input,

input\_format=input\_format,

output=output,

model=model,

model\_config=model\_config,

),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

speech\_span

speech\_span(

model: str | None = None,

input: str | None = None,

output: str | None = None,

output\_format: str | None = "pcm",

model\_config: Mapping[str, Any] | None = None,

first\_content\_at: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[SpeechSpanData]

Create a new speech span. The span will not be started automatically, you should either do with speech\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

model str | None The name of the model used for the text-to-speech. None

input str | None The text input of the text-to-speech. None

output str | None The audio output of the text-to-speech as base64 encoded string of PCM audio bytes. None

output\_format str | None The format of the audio output (defaults to "pcm"). 'pcm'

model\_config Mapping[str, Any] | None The model configuration (hyperparameters) used. None

first\_content\_at str | None The time of the first byte of the audio output. None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Source code in src/agents/tracing/create.py

def speech\_span(

model: str | None = None,

input: str | None = None,

output: str | None = None,

output\_format: str | None = "pcm",

model\_config: Mapping[str, Any] | None = None,

first\_content\_at: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[SpeechSpanData]:

"""Create a new speech span. The span will not be started automatically, you should either do

`with speech\_span() ...` or call `span.start()` + `span.finish()` manually.

Args:

model: The name of the model used for the text-to-speech.

input: The text input of the text-to-speech.

output: The audio output of the text-to-speech as base64 encoded string of PCM audio bytes.

output\_format: The format of the audio output (defaults to "pcm").

model\_config: The model configuration (hyperparameters) used.

first\_content\_at: The time of the first byte of the audio output.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=SpeechSpanData(

model=model,

input=input,

output=output,

output\_format=output\_format,

model\_config=model\_config,

first\_content\_at=first\_content\_at,

),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

speech\_group\_span

speech\_group\_span(

input: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[SpeechGroupSpanData]

Create a new speech group span. The span will not be started automatically, you should either do with speech\_group\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

input str | None The input text used for the speech request. None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Source code in src/agents/tracing/create.py

def speech\_group\_span(

input: str | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[SpeechGroupSpanData]:

"""Create a new speech group span. The span will not be started automatically, you should

either do `with speech\_group\_span() ...` or call `span.start()` + `span.finish()` manually.

Args:

input: The input text used for the speech request.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=SpeechGroupSpanData(input=input),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

mcp\_tools\_span

mcp\_tools\_span(

server: str | None = None,

result: list[str] | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[MCPListToolsSpanData]

Create a new MCP list tools span. The span will not be started automatically, you should either do with mcp\_tools\_span() ... or call span.start() + span.finish() manually.

Parameters:

Name Type Description Default

server str | None The name of the MCP server. None

result list[str] | None The result of the MCP list tools call. None

span\_id str | None The ID of the span. Optional. If not provided, we will generate an ID. We recommend using util.gen\_span\_id() to generate a span ID, to guarantee that IDs are correctly formatted. None

parent Trace | Span[Any] | None The parent span or trace. If not provided, we will automatically use the current trace/span as the parent. None

disabled bool If True, we will return a Span but the Span will not be recorded. False

Source code in src/agents/tracing/create.py

def mcp\_tools\_span(

server: str | None = None,

result: list[str] | None = None,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[MCPListToolsSpanData]:

"""Create a new MCP list tools span. The span will not be started automatically, you should

either do `with mcp\_tools\_span() ...` or call `span.start()` + `span.finish()` manually.

Args:

server: The name of the MCP server.

result: The result of the MCP list tools call.

span\_id: The ID of the span. Optional. If not provided, we will generate an ID. We

recommend using `util.gen\_span\_id()` to generate a span ID, to guarantee that IDs are

correctly formatted.

parent: The parent span or trace. If not provided, we will automatically use the current

trace/span as the parent.

disabled: If True, we will return a Span but the Span will not be recorded.

"""

return GLOBAL\_TRACE\_PROVIDER.create\_span(

span\_data=MCPListToolsSpanData(server=server, result=result),

span\_id=span\_id,

parent=parent,

disabled=disabled,

)

Traces

Trace

A trace is the root level object that tracing creates. It represents a logical "workflow".

Source code in src/agents/tracing/traces.py

class Trace:

"""

A trace is the root level object that tracing creates. It represents a logical "workflow".

"""

@abc.abstractmethod

def \_\_enter\_\_(self) -> Trace:

pass

@abc.abstractmethod

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

pass

@abc.abstractmethod

def start(self, mark\_as\_current: bool = False):

"""

Start the trace.

Args:

mark\_as\_current: If true, the trace will be marked as the current trace.

"""

pass

@abc.abstractmethod

def finish(self, reset\_current: bool = False):

"""

Finish the trace.

Args:

reset\_current: If true, the trace will be reset as the current trace.

"""

pass

@property

@abc.abstractmethod

def trace\_id(self) -> str:

"""

The trace ID.

"""

pass

@property

@abc.abstractmethod

def name(self) -> str:

"""

The name of the workflow being traced.

"""

pass

@abc.abstractmethod

def export(self) -> dict[str, Any] | None:

"""

Export the trace as a dictionary.

"""

pass

trace\_id abstractmethod property

trace\_id: str

The trace ID.

name abstractmethod property

name: str

The name of the workflow being traced.

start abstractmethod

start(mark\_as\_current: bool = False)

Start the trace.

Parameters:

Name Type Description Default

mark\_as\_current bool If true, the trace will be marked as the current trace. False

Source code in src/agents/tracing/traces.py

@abc.abstractmethod

def start(self, mark\_as\_current: bool = False):

"""

Start the trace.

Args:

mark\_as\_current: If true, the trace will be marked as the current trace.

"""

pass

finish abstractmethod

finish(reset\_current: bool = False)

Finish the trace.

Parameters:

Name Type Description Default

reset\_current bool If true, the trace will be reset as the current trace. False

Source code in src/agents/tracing/traces.py

@abc.abstractmethod

def finish(self, reset\_current: bool = False):

"""

Finish the trace.

Args:

reset\_current: If true, the trace will be reset as the current trace.

"""

pass

export abstractmethod

export() -> dict[str, Any] | None

Export the trace as a dictionary.

Source code in src/agents/tracing/traces.py

@abc.abstractmethod

def export(self) -> dict[str, Any] | None:

"""

Export the trace as a dictionary.

"""

pass

NoOpTrace

Bases: Trace

A no-op trace that will not be recorded.

Source code in src/agents/tracing/traces.py

class NoOpTrace(Trace):

"""

A no-op trace that will not be recorded.

"""

def \_\_init\_\_(self):

self.\_started = False

self.\_prev\_context\_token: contextvars.Token[Trace | None] | None = None

def \_\_enter\_\_(self) -> Trace:

if self.\_started:

if not self.\_prev\_context\_token:

logger.error("Trace already started but no context token set")

return self

self.\_started = True

self.start(mark\_as\_current=True)

return self

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

self.finish(reset\_current=True)

def start(self, mark\_as\_current: bool = False):

if mark\_as\_current:

self.\_prev\_context\_token = Scope.set\_current\_trace(self)

def finish(self, reset\_current: bool = False):

if reset\_current and self.\_prev\_context\_token is not None:

Scope.reset\_current\_trace(self.\_prev\_context\_token)

self.\_prev\_context\_token = None

@property

def trace\_id(self) -> str:

return "no-op"

@property

def name(self) -> str:

return "no-op"

def export(self) -> dict[str, Any] | None:

return None

TraceImpl

Bases: Trace

A trace that will be recorded by the tracing library.

Source code in src/agents/tracing/traces.py

class TraceImpl(Trace):

"""

A trace that will be recorded by the tracing library.

"""

\_\_slots\_\_ = (

"\_name",

"\_trace\_id",

"group\_id",

"metadata",

"\_prev\_context\_token",

"\_processor",

"\_started",

)

def \_\_init\_\_(

self,

name: str,

trace\_id: str | None,

group\_id: str | None,

metadata: dict[str, Any] | None,

processor: TracingProcessor,

):

self.\_name = name

self.\_trace\_id = trace\_id or util.gen\_trace\_id()

self.group\_id = group\_id

self.metadata = metadata

self.\_prev\_context\_token: contextvars.Token[Trace | None] | None = None

self.\_processor = processor

self.\_started = False

@property

def trace\_id(self) -> str:

return self.\_trace\_id

@property

def name(self) -> str:

return self.\_name

def start(self, mark\_as\_current: bool = False):

if self.\_started:

return

self.\_started = True

self.\_processor.on\_trace\_start(self)

if mark\_as\_current:

self.\_prev\_context\_token = Scope.set\_current\_trace(self)

def finish(self, reset\_current: bool = False):

if not self.\_started:

return

self.\_processor.on\_trace\_end(self)

if reset\_current and self.\_prev\_context\_token is not None:

Scope.reset\_current\_trace(self.\_prev\_context\_token)

self.\_prev\_context\_token = None

def \_\_enter\_\_(self) -> Trace:

if self.\_started:

if not self.\_prev\_context\_token:

logger.error("Trace already started but no context token set")

return self

self.start(mark\_as\_current=True)

return self

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

self.finish(reset\_current=exc\_type is not GeneratorExit)

def export(self) -> dict[str, Any] | None:

return {

"object": "trace",

"id": self.trace\_id,

"workflow\_name": self.name,

"group\_id": self.group\_id,

"metadata": self.metadata,

}

Spans

Span

Bases: ABC, Generic[TSpanData]

Source code in src/agents/tracing/spans.py

class Span(abc.ABC, Generic[TSpanData]):

@property

@abc.abstractmethod

def trace\_id(self) -> str:

pass

@property

@abc.abstractmethod

def span\_id(self) -> str:

pass

@property

@abc.abstractmethod

def span\_data(self) -> TSpanData:

pass

@abc.abstractmethod

def start(self, mark\_as\_current: bool = False):

"""

Start the span.

Args:

mark\_as\_current: If true, the span will be marked as the current span.

"""

pass

@abc.abstractmethod

def finish(self, reset\_current: bool = False) -> None:

"""

Finish the span.

Args:

reset\_current: If true, the span will be reset as the current span.

"""

pass

@abc.abstractmethod

def \_\_enter\_\_(self) -> Span[TSpanData]:

pass

@abc.abstractmethod

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

pass

@property

@abc.abstractmethod

def parent\_id(self) -> str | None:

pass

@abc.abstractmethod

def set\_error(self, error: SpanError) -> None:

pass

@property

@abc.abstractmethod

def error(self) -> SpanError | None:

pass

@abc.abstractmethod

def export(self) -> dict[str, Any] | None:

pass

@property

@abc.abstractmethod

def started\_at(self) -> str | None:

pass

@property

@abc.abstractmethod

def ended\_at(self) -> str | None:

pass

start abstractmethod

start(mark\_as\_current: bool = False)

Start the span.

Parameters:

Name Type Description Default

mark\_as\_current bool If true, the span will be marked as the current span. False

Source code in src/agents/tracing/spans.py

@abc.abstractmethod

def start(self, mark\_as\_current: bool = False):

"""

Start the span.

Args:

mark\_as\_current: If true, the span will be marked as the current span.

"""

pass

finish abstractmethod

finish(reset\_current: bool = False) -> None

Finish the span.

Parameters:

Name Type Description Default

reset\_current bool If true, the span will be reset as the current span. False

Source code in src/agents/tracing/spans.py

@abc.abstractmethod

def finish(self, reset\_current: bool = False) -> None:

"""

Finish the span.

Args:

reset\_current: If true, the span will be reset as the current span.

"""

pass

NoOpSpan

Bases: Span[TSpanData]

Source code in src/agents/tracing/spans.py

class NoOpSpan(Span[TSpanData]):

\_\_slots\_\_ = ("\_span\_data", "\_prev\_span\_token")

def \_\_init\_\_(self, span\_data: TSpanData):

self.\_span\_data = span\_data

self.\_prev\_span\_token: contextvars.Token[Span[TSpanData] | None] | None = None

@property

def trace\_id(self) -> str:

return "no-op"

@property

def span\_id(self) -> str:

return "no-op"

@property

def span\_data(self) -> TSpanData:

return self.\_span\_data

@property

def parent\_id(self) -> str | None:

return None

def start(self, mark\_as\_current: bool = False):

if mark\_as\_current:

self.\_prev\_span\_token = Scope.set\_current\_span(self)

def finish(self, reset\_current: bool = False) -> None:

if reset\_current and self.\_prev\_span\_token is not None:

Scope.reset\_current\_span(self.\_prev\_span\_token)

self.\_prev\_span\_token = None

def \_\_enter\_\_(self) -> Span[TSpanData]:

self.start(mark\_as\_current=True)

return self

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

reset\_current = True

if exc\_type is GeneratorExit:

logger.debug("GeneratorExit, skipping span reset")

reset\_current = False

self.finish(reset\_current=reset\_current)

def set\_error(self, error: SpanError) -> None:

pass

@property

def error(self) -> SpanError | None:

return None

def export(self) -> dict[str, Any] | None:

return None

@property

def started\_at(self) -> str | None:

return None

@property

def ended\_at(self) -> str | None:

return None

SpanImpl

Bases: Span[TSpanData]

Source code in src/agents/tracing/spans.py

class SpanImpl(Span[TSpanData]):

\_\_slots\_\_ = (

"\_trace\_id",

"\_span\_id",

"\_parent\_id",

"\_started\_at",

"\_ended\_at",

"\_error",

"\_prev\_span\_token",

"\_processor",

"\_span\_data",

)

def \_\_init\_\_(

self,

trace\_id: str,

span\_id: str | None,

parent\_id: str | None,

processor: TracingProcessor,

span\_data: TSpanData,

):

self.\_trace\_id = trace\_id

self.\_span\_id = span\_id or util.gen\_span\_id()

self.\_parent\_id = parent\_id

self.\_started\_at: str | None = None

self.\_ended\_at: str | None = None

self.\_processor = processor

self.\_error: SpanError | None = None

self.\_prev\_span\_token: contextvars.Token[Span[TSpanData] | None] | None = None

self.\_span\_data = span\_data

@property

def trace\_id(self) -> str:

return self.\_trace\_id

@property

def span\_id(self) -> str:

return self.\_span\_id

@property

def span\_data(self) -> TSpanData:

return self.\_span\_data

@property

def parent\_id(self) -> str | None:

return self.\_parent\_id

def start(self, mark\_as\_current: bool = False):

if self.started\_at is not None:

logger.warning("Span already started")

return

self.\_started\_at = util.time\_iso()

self.\_processor.on\_span\_start(self)

if mark\_as\_current:

self.\_prev\_span\_token = Scope.set\_current\_span(self)

def finish(self, reset\_current: bool = False) -> None:

if self.ended\_at is not None:

logger.warning("Span already finished")

return

self.\_ended\_at = util.time\_iso()

self.\_processor.on\_span\_end(self)

if reset\_current and self.\_prev\_span\_token is not None:

Scope.reset\_current\_span(self.\_prev\_span\_token)

self.\_prev\_span\_token = None

def \_\_enter\_\_(self) -> Span[TSpanData]:

self.start(mark\_as\_current=True)

return self

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

reset\_current = True

if exc\_type is GeneratorExit:

logger.debug("GeneratorExit, skipping span reset")

reset\_current = False

self.finish(reset\_current=reset\_current)

def set\_error(self, error: SpanError) -> None:

self.\_error = error

@property

def error(self) -> SpanError | None:

return self.\_error

@property

def started\_at(self) -> str | None:

return self.\_started\_at

@property

def ended\_at(self) -> str | None:

return self.\_ended\_at

def export(self) -> dict[str, Any] | None:

return {

"object": "trace.span",

"id": self.span\_id,

"trace\_id": self.trace\_id,

"parent\_id": self.\_parent\_id,

"started\_at": self.\_started\_at,

"ended\_at": self.\_ended\_at,

"span\_data": self.span\_data.export(),

"error": self.\_error,

}

Processor interface

TracingProcessor

Bases: ABC

Interface for processing spans.

Source code in src/agents/tracing/processor\_interface.py

class TracingProcessor(abc.ABC):

"""Interface for processing spans."""

@abc.abstractmethod

def on\_trace\_start(self, trace: "Trace") -> None:

"""Called when a trace is started.

Args:

trace: The trace that started.

"""

pass

@abc.abstractmethod

def on\_trace\_end(self, trace: "Trace") -> None:

"""Called when a trace is finished.

Args:

trace: The trace that started.

"""

pass

@abc.abstractmethod

def on\_span\_start(self, span: "Span[Any]") -> None:

"""Called when a span is started.

Args:

span: The span that started.

"""

pass

@abc.abstractmethod

def on\_span\_end(self, span: "Span[Any]") -> None:

"""Called when a span is finished. Should not block or raise exceptions.

Args:

span: The span that finished.

"""

pass

@abc.abstractmethod

def shutdown(self) -> None:

"""Called when the application stops."""

pass

@abc.abstractmethod

def force\_flush(self) -> None:

"""Forces an immediate flush of all queued spans/traces."""

pass

on\_trace\_start abstractmethod

on\_trace\_start(trace: Trace) -> None

Called when a trace is started.

Parameters:

Name Type Description Default

trace Trace The trace that started. required

Source code in src/agents/tracing/processor\_interface.py

@abc.abstractmethod

def on\_trace\_start(self, trace: "Trace") -> None:

"""Called when a trace is started.

Args:

trace: The trace that started.

"""

pass

on\_trace\_end abstractmethod

on\_trace\_end(trace: Trace) -> None

Called when a trace is finished.

Parameters:

Name Type Description Default

trace Trace The trace that started. required

Source code in src/agents/tracing/processor\_interface.py

@abc.abstractmethod

def on\_trace\_end(self, trace: "Trace") -> None:

"""Called when a trace is finished.

Args:

trace: The trace that started.

"""

pass

on\_span\_start abstractmethod

on\_span\_start(span: Span[Any]) -> None

Called when a span is started.

Parameters:

Name Type Description Default

span Span[Any] The span that started. required

Source code in src/agents/tracing/processor\_interface.py

@abc.abstractmethod

def on\_span\_start(self, span: "Span[Any]") -> None:

"""Called when a span is started.

Args:

span: The span that started.

"""

pass

on\_span\_end abstractmethod

on\_span\_end(span: Span[Any]) -> None

Called when a span is finished. Should not block or raise exceptions.

Parameters:

Name Type Description Default

span Span[Any] The span that finished. required

Source code in src/agents/tracing/processor\_interface.py

@abc.abstractmethod

def on\_span\_end(self, span: "Span[Any]") -> None:

"""Called when a span is finished. Should not block or raise exceptions.

Args:

span: The span that finished.

"""

pass

shutdown abstractmethod

shutdown() -> None

Called when the application stops.

Source code in src/agents/tracing/processor\_interface.py

@abc.abstractmethod

def shutdown(self) -> None:

"""Called when the application stops."""

pass

force\_flush abstractmethod

force\_flush() -> None

Forces an immediate flush of all queued spans/traces.

Source code in src/agents/tracing/processor\_interface.py

@abc.abstractmethod

def force\_flush(self) -> None:

"""Forces an immediate flush of all queued spans/traces."""

pass

TracingExporter

Bases: ABC

Exports traces and spans. For example, could log them or send them to a backend.

Source code in src/agents/tracing/processor\_interface.py

class TracingExporter(abc.ABC):

"""Exports traces and spans. For example, could log them or send them to a backend."""

@abc.abstractmethod

def export(self, items: list["Trace | Span[Any]"]) -> None:

"""Exports a list of traces and spans.

Args:

items: The items to export.

"""

pass

export abstractmethod

export(items: list[Trace | Span[Any]]) -> None

Exports a list of traces and spans.

Parameters:

Name Type Description Default

items list[Trace | Span[Any]] The items to export. required

Source code in src/agents/tracing/processor\_interface.py

@abc.abstractmethod

def export(self, items: list["Trace | Span[Any]"]) -> None:

"""Exports a list of traces and spans.

Args:

items: The items to export.

"""

pass

Processors

ConsoleSpanExporter

Bases: TracingExporter

Prints the traces and spans to the console.

Source code in src/agents/tracing/processors.py

class ConsoleSpanExporter(TracingExporter):

"""Prints the traces and spans to the console."""

def export(self, items: list[Trace | Span[Any]]) -> None:

for item in items:

if isinstance(item, Trace):

print(f"[Exporter] Export trace\_id={item.trace\_id}, name={item.name}, ")

else:

print(f"[Exporter] Export span: {item.export()}")

BackendSpanExporter

Bases: TracingExporter

Source code in src/agents/tracing/processors.py

class BackendSpanExporter(TracingExporter):

def \_\_init\_\_(

self,

api\_key: str | None = None,

organization: str | None = None,

project: str | None = None,

endpoint: str = "https://api.openai.com/v1/traces/ingest",

max\_retries: int = 3,

base\_delay: float = 1.0,

max\_delay: float = 30.0,

):

"""

Args:

api\_key: The API key for the "Authorization" header. Defaults to

`os.environ["OPENAI\_API\_KEY"]` if not provided.

organization: The OpenAI organization to use. Defaults to

`os.environ["OPENAI\_ORG\_ID"]` if not provided.

project: The OpenAI project to use. Defaults to

`os.environ["OPENAI\_PROJECT\_ID"]` if not provided.

endpoint: The HTTP endpoint to which traces/spans are posted.

max\_retries: Maximum number of retries upon failures.

base\_delay: Base delay (in seconds) for the first backoff.

max\_delay: Maximum delay (in seconds) for backoff growth.

"""

self.\_api\_key = api\_key

self.\_organization = organization

self.\_project = project

self.endpoint = endpoint

self.max\_retries = max\_retries

self.base\_delay = base\_delay

self.max\_delay = max\_delay

# Keep a client open for connection pooling across multiple export calls

self.\_client = httpx.Client(timeout=httpx.Timeout(timeout=60, connect=5.0))

def set\_api\_key(self, api\_key: str):

"""Set the OpenAI API key for the exporter.

Args:

api\_key: The OpenAI API key to use. This is the same key used by the OpenAI Python

client.

"""

# We're specifically setting the underlying cached property as well

self.\_api\_key = api\_key

self.api\_key = api\_key

@cached\_property

def api\_key(self):

return self.\_api\_key or os.environ.get("OPENAI\_API\_KEY")

@cached\_property

def organization(self):

return self.\_organization or os.environ.get("OPENAI\_ORG\_ID")

@cached\_property

def project(self):

return self.\_project or os.environ.get("OPENAI\_PROJECT\_ID")

def export(self, items: list[Trace | Span[Any]]) -> None:

if not items:

return

if not self.api\_key:

logger.warning("OPENAI\_API\_KEY is not set, skipping trace export")

return

data = [item.export() for item in items if item.export()]

payload = {"data": data}

headers = {

"Authorization": f"Bearer {self.api\_key}",

"Content-Type": "application/json",

"OpenAI-Beta": "traces=v1",

}

# Exponential backoff loop

attempt = 0

delay = self.base\_delay

while True:

attempt += 1

try:

response = self.\_client.post(url=self.endpoint, headers=headers, json=payload)

# If the response is successful, break out of the loop

if response.status\_code < 300:

logger.debug(f"Exported {len(items)} items")

return

# If the response is a client error (4xx), we wont retry

if 400 <= response.status\_code < 500:

logger.error(

f"[non-fatal] Tracing client error {response.status\_code}: {response.text}"

)

return

# For 5xx or other unexpected codes, treat it as transient and retry

logger.warning(

f"[non-fatal] Tracing: server error {response.status\_code}, retrying."

)

except httpx.RequestError as exc:

# Network or other I/O error, we'll retry

logger.warning(f"[non-fatal] Tracing: request failed: {exc}")

# If we reach here, we need to retry or give up

if attempt >= self.max\_retries:

logger.error("[non-fatal] Tracing: max retries reached, giving up on this batch.")

return

# Exponential backoff + jitter

sleep\_time = delay + random.uniform(0, 0.1 \* delay) # 10% jitter

time.sleep(sleep\_time)

delay = min(delay \* 2, self.max\_delay)

def close(self):

"""Close the underlying HTTP client."""

self.\_client.close()

\_\_init\_\_

\_\_init\_\_(

api\_key: str | None = None,

organization: str | None = None,

project: str | None = None,

endpoint: str = "https://api.openai.com/v1/traces/ingest",

max\_retries: int = 3,

base\_delay: float = 1.0,

max\_delay: float = 30.0,

)

Parameters:

Name Type Description Default

api\_key str | None The API key for the "Authorization" header. Defaults to os.environ["OPENAI\_API\_KEY"] if not provided. None

organization str | None The OpenAI organization to use. Defaults to os.environ["OPENAI\_ORG\_ID"] if not provided. None

project str | None The OpenAI project to use. Defaults to os.environ["OPENAI\_PROJECT\_ID"] if not provided. None

endpoint str The HTTP endpoint to which traces/spans are posted. 'https://api.openai.com/v1/traces/ingest'

max\_retries int Maximum number of retries upon failures. 3

base\_delay float Base delay (in seconds) for the first backoff. 1.0

max\_delay float Maximum delay (in seconds) for backoff growth. 30.0

Source code in src/agents/tracing/processors.py

def \_\_init\_\_(

self,

api\_key: str | None = None,

organization: str | None = None,

project: str | None = None,

endpoint: str = "https://api.openai.com/v1/traces/ingest",

max\_retries: int = 3,

base\_delay: float = 1.0,

max\_delay: float = 30.0,

):

"""

Args:

api\_key: The API key for the "Authorization" header. Defaults to

`os.environ["OPENAI\_API\_KEY"]` if not provided.

organization: The OpenAI organization to use. Defaults to

`os.environ["OPENAI\_ORG\_ID"]` if not provided.

project: The OpenAI project to use. Defaults to

`os.environ["OPENAI\_PROJECT\_ID"]` if not provided.

endpoint: The HTTP endpoint to which traces/spans are posted.

max\_retries: Maximum number of retries upon failures.

base\_delay: Base delay (in seconds) for the first backoff.

max\_delay: Maximum delay (in seconds) for backoff growth.

"""

self.\_api\_key = api\_key

self.\_organization = organization

self.\_project = project

self.endpoint = endpoint

self.max\_retries = max\_retries

self.base\_delay = base\_delay

self.max\_delay = max\_delay

# Keep a client open for connection pooling across multiple export calls

self.\_client = httpx.Client(timeout=httpx.Timeout(timeout=60, connect=5.0))

set\_api\_key

set\_api\_key(api\_key: str)

Set the OpenAI API key for the exporter.

Parameters:

Name Type Description Default

api\_key str The OpenAI API key to use. This is the same key used by the OpenAI Python client. required

Source code in src/agents/tracing/processors.py

def set\_api\_key(self, api\_key: str):

"""Set the OpenAI API key for the exporter.

Args:

api\_key: The OpenAI API key to use. This is the same key used by the OpenAI Python

client.

"""

# We're specifically setting the underlying cached property as well

self.\_api\_key = api\_key

self.api\_key = api\_key

close

close()

Close the underlying HTTP client.

Source code in src/agents/tracing/processors.py

def close(self):

"""Close the underlying HTTP client."""

self.\_client.close()

BatchTraceProcessor

Bases: TracingProcessor

Some implementation notes: 1. Using Queue, which is thread-safe. 2. Using a background thread to export spans, to minimize any performance issues. 3. Spans are stored in memory until they are exported.

Source code in src/agents/tracing/processors.py

class BatchTraceProcessor(TracingProcessor):

"""Some implementation notes:

1. Using Queue, which is thread-safe.

2. Using a background thread to export spans, to minimize any performance issues.

3. Spans are stored in memory until they are exported.

"""

def \_\_init\_\_(

self,

exporter: TracingExporter,

max\_queue\_size: int = 8192,

max\_batch\_size: int = 128,

schedule\_delay: float = 5.0,

export\_trigger\_ratio: float = 0.7,

):

"""

Args:

exporter: The exporter to use.

max\_queue\_size: The maximum number of spans to store in the queue. After this, we will

start dropping spans.

max\_batch\_size: The maximum number of spans to export in a single batch.

schedule\_delay: The delay between checks for new spans to export.

export\_trigger\_ratio: The ratio of the queue size at which we will trigger an export.

"""

self.\_exporter = exporter

self.\_queue: queue.Queue[Trace | Span[Any]] = queue.Queue(maxsize=max\_queue\_size)

self.\_max\_queue\_size = max\_queue\_size

self.\_max\_batch\_size = max\_batch\_size

self.\_schedule\_delay = schedule\_delay

self.\_shutdown\_event = threading.Event()

# The queue size threshold at which we export immediately.

self.\_export\_trigger\_size = int(max\_queue\_size \* export\_trigger\_ratio)

# Track when we next \*must\* perform a scheduled export

self.\_next\_export\_time = time.time() + self.\_schedule\_delay

self.\_shutdown\_event = threading.Event()

self.\_worker\_thread = threading.Thread(target=self.\_run, daemon=True)

self.\_worker\_thread.start()

def on\_trace\_start(self, trace: Trace) -> None:

try:

self.\_queue.put\_nowait(trace)

except queue.Full:

logger.warning("Queue is full, dropping trace.")

def on\_trace\_end(self, trace: Trace) -> None:

# We send traces via on\_trace\_start, so we don't need to do anything here.

pass

def on\_span\_start(self, span: Span[Any]) -> None:

# We send spans via on\_span\_end, so we don't need to do anything here.

pass

def on\_span\_end(self, span: Span[Any]) -> None:

try:

self.\_queue.put\_nowait(span)

except queue.Full:

logger.warning("Queue is full, dropping span.")

def shutdown(self, timeout: float | None = None):

"""

Called when the application stops. We signal our thread to stop, then join it.

"""

self.\_shutdown\_event.set()

self.\_worker\_thread.join(timeout=timeout)

def force\_flush(self):

"""

Forces an immediate flush of all queued spans.

"""

self.\_export\_batches(force=True)

def \_run(self):

while not self.\_shutdown\_event.is\_set():

current\_time = time.time()

queue\_size = self.\_queue.qsize()

# If it's time for a scheduled flush or queue is above the trigger threshold

if current\_time >= self.\_next\_export\_time or queue\_size >= self.\_export\_trigger\_size:

self.\_export\_batches(force=False)

# Reset the next scheduled flush time

self.\_next\_export\_time = time.time() + self.\_schedule\_delay

else:

# Sleep a short interval so we don't busy-wait.

time.sleep(0.2)

# Final drain after shutdown

self.\_export\_batches(force=True)

def \_export\_batches(self, force: bool = False):

"""Drains the queue and exports in batches. If force=True, export everything.

Otherwise, export up to `max\_batch\_size` repeatedly until the queue is empty or below a

certain threshold.

"""

while True:

items\_to\_export: list[Span[Any] | Trace] = []

# Gather a batch of spans up to max\_batch\_size

while not self.\_queue.empty() and (

force or len(items\_to\_export) < self.\_max\_batch\_size

):

try:

items\_to\_export.append(self.\_queue.get\_nowait())

except queue.Empty:

# Another thread might have emptied the queue between checks

break

# If we collected nothing, we're done

if not items\_to\_export:

break

# Export the batch

self.\_exporter.export(items\_to\_export)

\_\_init\_\_

\_\_init\_\_(

exporter: TracingExporter,

max\_queue\_size: int = 8192,

max\_batch\_size: int = 128,

schedule\_delay: float = 5.0,

export\_trigger\_ratio: float = 0.7,

)

Parameters:

Name Type Description Default

exporter TracingExporter The exporter to use. required

max\_queue\_size int The maximum number of spans to store in the queue. After this, we will start dropping spans. 8192

max\_batch\_size int The maximum number of spans to export in a single batch. 128

schedule\_delay float The delay between checks for new spans to export. 5.0

export\_trigger\_ratio float The ratio of the queue size at which we will trigger an export. 0.7

Source code in src/agents/tracing/processors.py

def \_\_init\_\_(

self,

exporter: TracingExporter,

max\_queue\_size: int = 8192,

max\_batch\_size: int = 128,

schedule\_delay: float = 5.0,

export\_trigger\_ratio: float = 0.7,

):

"""

Args:

exporter: The exporter to use.

max\_queue\_size: The maximum number of spans to store in the queue. After this, we will

start dropping spans.

max\_batch\_size: The maximum number of spans to export in a single batch.

schedule\_delay: The delay between checks for new spans to export.

export\_trigger\_ratio: The ratio of the queue size at which we will trigger an export.

"""

self.\_exporter = exporter

self.\_queue: queue.Queue[Trace | Span[Any]] = queue.Queue(maxsize=max\_queue\_size)

self.\_max\_queue\_size = max\_queue\_size

self.\_max\_batch\_size = max\_batch\_size

self.\_schedule\_delay = schedule\_delay

self.\_shutdown\_event = threading.Event()

# The queue size threshold at which we export immediately.

self.\_export\_trigger\_size = int(max\_queue\_size \* export\_trigger\_ratio)

# Track when we next \*must\* perform a scheduled export

self.\_next\_export\_time = time.time() + self.\_schedule\_delay

self.\_shutdown\_event = threading.Event()

self.\_worker\_thread = threading.Thread(target=self.\_run, daemon=True)

self.\_worker\_thread.start()

shutdown

shutdown(timeout: float | None = None)

Called when the application stops. We signal our thread to stop, then join it.

Source code in src/agents/tracing/processors.py

def shutdown(self, timeout: float | None = None):

"""

Called when the application stops. We signal our thread to stop, then join it.

"""

self.\_shutdown\_event.set()

self.\_worker\_thread.join(timeout=timeout)

force\_flush

force\_flush()

Forces an immediate flush of all queued spans.

Source code in src/agents/tracing/processors.py

def force\_flush(self):

"""

Forces an immediate flush of all queued spans.

"""

self.\_export\_batches(force=True)

default\_exporter

default\_exporter() -> BackendSpanExporter

The default exporter, which exports traces and spans to the backend in batches.

Source code in src/agents/tracing/processors.py

def default\_exporter() -> BackendSpanExporter:

"""The default exporter, which exports traces and spans to the backend in batches."""

return \_global\_exporter

default\_processor

default\_processor() -> BatchTraceProcessor

The default processor, which exports traces and spans to the backend in batches.

Source code in src/agents/tracing/processors.py

def default\_processor() -> BatchTraceProcessor:

"""The default processor, which exports traces and spans to the backend in batches."""

return \_global\_processor

Setup

SynchronousMultiTracingProcessor

Bases: TracingProcessor

Forwards all calls to a list of TracingProcessors, in order of registration.

Source code in src/agents/tracing/setup.py

class SynchronousMultiTracingProcessor(TracingProcessor):

"""

Forwards all calls to a list of TracingProcessors, in order of registration.

"""

def \_\_init\_\_(self):

# Using a tuple to avoid race conditions when iterating over processors

self.\_processors: tuple[TracingProcessor, ...] = ()

self.\_lock = threading.Lock()

def add\_tracing\_processor(self, tracing\_processor: TracingProcessor):

"""

Add a processor to the list of processors. Each processor will receive all traces/spans.

"""

with self.\_lock:

self.\_processors += (tracing\_processor,)

def set\_processors(self, processors: list[TracingProcessor]):

"""

Set the list of processors. This will replace the current list of processors.

"""

with self.\_lock:

self.\_processors = tuple(processors)

def on\_trace\_start(self, trace: Trace) -> None:

"""

Called when a trace is started.

"""

for processor in self.\_processors:

processor.on\_trace\_start(trace)

def on\_trace\_end(self, trace: Trace) -> None:

"""

Called when a trace is finished.

"""

for processor in self.\_processors:

processor.on\_trace\_end(trace)

def on\_span\_start(self, span: Span[Any]) -> None:

"""

Called when a span is started.

"""

for processor in self.\_processors:

processor.on\_span\_start(span)

def on\_span\_end(self, span: Span[Any]) -> None:

"""

Called when a span is finished.

"""

for processor in self.\_processors:

processor.on\_span\_end(span)

def shutdown(self) -> None:

"""

Called when the application stops.

"""

for processor in self.\_processors:

logger.debug(f"Shutting down trace processor {processor}")

processor.shutdown()

def force\_flush(self):

"""

Force the processors to flush their buffers.

"""

for processor in self.\_processors:

processor.force\_flush()

add\_tracing\_processor

add\_tracing\_processor(tracing\_processor: TracingProcessor)

Add a processor to the list of processors. Each processor will receive all traces/spans.

Source code in src/agents/tracing/setup.py

def add\_tracing\_processor(self, tracing\_processor: TracingProcessor):

"""

Add a processor to the list of processors. Each processor will receive all traces/spans.

"""

with self.\_lock:

self.\_processors += (tracing\_processor,)

set\_processors

set\_processors(processors: list[TracingProcessor])

Set the list of processors. This will replace the current list of processors.

Source code in src/agents/tracing/setup.py

def set\_processors(self, processors: list[TracingProcessor]):

"""

Set the list of processors. This will replace the current list of processors.

"""

with self.\_lock:

self.\_processors = tuple(processors)

on\_trace\_start

on\_trace\_start(trace: Trace) -> None

Called when a trace is started.

Source code in src/agents/tracing/setup.py

def on\_trace\_start(self, trace: Trace) -> None:

"""

Called when a trace is started.

"""

for processor in self.\_processors:

processor.on\_trace\_start(trace)

on\_trace\_end

on\_trace\_end(trace: Trace) -> None

Called when a trace is finished.

Source code in src/agents/tracing/setup.py

def on\_trace\_end(self, trace: Trace) -> None:

"""

Called when a trace is finished.

"""

for processor in self.\_processors:

processor.on\_trace\_end(trace)

on\_span\_start

on\_span\_start(span: Span[Any]) -> None

Called when a span is started.

Source code in src/agents/tracing/setup.py

def on\_span\_start(self, span: Span[Any]) -> None:

"""

Called when a span is started.

"""

for processor in self.\_processors:

processor.on\_span\_start(span)

on\_span\_end

on\_span\_end(span: Span[Any]) -> None

Called when a span is finished.

Source code in src/agents/tracing/setup.py

def on\_span\_end(self, span: Span[Any]) -> None:

"""

Called when a span is finished.

"""

for processor in self.\_processors:

processor.on\_span\_end(span)

shutdown

shutdown() -> None

Called when the application stops.

Source code in src/agents/tracing/setup.py

def shutdown(self) -> None:

"""

Called when the application stops.

"""

for processor in self.\_processors:

logger.debug(f"Shutting down trace processor {processor}")

processor.shutdown()

force\_flush

force\_flush()

Force the processors to flush their buffers.

Source code in src/agents/tracing/setup.py

def force\_flush(self):

"""

Force the processors to flush their buffers.

"""

for processor in self.\_processors:

processor.force\_flush()

TraceProvider

Source code in src/agents/tracing/setup.py

class TraceProvider:

def \_\_init\_\_(self):

self.\_multi\_processor = SynchronousMultiTracingProcessor()

self.\_disabled = os.environ.get("OPENAI\_AGENTS\_DISABLE\_TRACING", "false").lower() in (

"true",

"1",

)

def register\_processor(self, processor: TracingProcessor):

"""

Add a processor to the list of processors. Each processor will receive all traces/spans.

"""

self.\_multi\_processor.add\_tracing\_processor(processor)

def set\_processors(self, processors: list[TracingProcessor]):

"""

Set the list of processors. This will replace the current list of processors.

"""

self.\_multi\_processor.set\_processors(processors)

def get\_current\_trace(self) -> Trace | None:

"""

Returns the currently active trace, if any.

"""

return Scope.get\_current\_trace()

def get\_current\_span(self) -> Span[Any] | None:

"""

Returns the currently active span, if any.

"""

return Scope.get\_current\_span()

def set\_disabled(self, disabled: bool) -> None:

"""

Set whether tracing is disabled.

"""

self.\_disabled = disabled

def create\_trace(

self,

name: str,

trace\_id: str | None = None,

group\_id: str | None = None,

metadata: dict[str, Any] | None = None,

disabled: bool = False,

) -> Trace:

"""

Create a new trace.

"""

if self.\_disabled or disabled:

logger.debug(f"Tracing is disabled. Not creating trace {name}")

return NoOpTrace()

trace\_id = trace\_id or util.gen\_trace\_id()

logger.debug(f"Creating trace {name} with id {trace\_id}")

return TraceImpl(

name=name,

trace\_id=trace\_id,

group\_id=group\_id,

metadata=metadata,

processor=self.\_multi\_processor,

)

def create\_span(

self,

span\_data: TSpanData,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[TSpanData]:

"""

Create a new span.

"""

if self.\_disabled or disabled:

logger.debug(f"Tracing is disabled. Not creating span {span\_data}")

return NoOpSpan(span\_data)

if not parent:

current\_span = Scope.get\_current\_span()

current\_trace = Scope.get\_current\_trace()

if current\_trace is None:

logger.error(

"No active trace. Make sure to start a trace with `trace()` first"

"Returning NoOpSpan."

)

return NoOpSpan(span\_data)

elif isinstance(current\_trace, NoOpTrace) or isinstance(current\_span, NoOpSpan):

logger.debug(

f"Parent {current\_span} or {current\_trace} is no-op, returning NoOpSpan"

)

return NoOpSpan(span\_data)

parent\_id = current\_span.span\_id if current\_span else None

trace\_id = current\_trace.trace\_id

elif isinstance(parent, Trace):

if isinstance(parent, NoOpTrace):

logger.debug(f"Parent {parent} is no-op, returning NoOpSpan")

return NoOpSpan(span\_data)

trace\_id = parent.trace\_id

parent\_id = None

elif isinstance(parent, Span):

if isinstance(parent, NoOpSpan):

logger.debug(f"Parent {parent} is no-op, returning NoOpSpan")

return NoOpSpan(span\_data)

parent\_id = parent.span\_id

trace\_id = parent.trace\_id

logger.debug(f"Creating span {span\_data} with id {span\_id}")

return SpanImpl(

trace\_id=trace\_id,

span\_id=span\_id,

parent\_id=parent\_id,

processor=self.\_multi\_processor,

span\_data=span\_data,

)

def shutdown(self) -> None:

try:

logger.debug("Shutting down trace provider")

self.\_multi\_processor.shutdown()

except Exception as e:

logger.error(f"Error shutting down trace provider: {e}")

register\_processor

register\_processor(processor: TracingProcessor)

Add a processor to the list of processors. Each processor will receive all traces/spans.

Source code in src/agents/tracing/setup.py

def register\_processor(self, processor: TracingProcessor):

"""

Add a processor to the list of processors. Each processor will receive all traces/spans.

"""

self.\_multi\_processor.add\_tracing\_processor(processor)

set\_processors

set\_processors(processors: list[TracingProcessor])

Set the list of processors. This will replace the current list of processors.

Source code in src/agents/tracing/setup.py

def set\_processors(self, processors: list[TracingProcessor]):

"""

Set the list of processors. This will replace the current list of processors.

"""

self.\_multi\_processor.set\_processors(processors)

get\_current\_trace

get\_current\_trace() -> Trace | None

Returns the currently active trace, if any.

Source code in src/agents/tracing/setup.py

def get\_current\_trace(self) -> Trace | None:

"""

Returns the currently active trace, if any.

"""

return Scope.get\_current\_trace()

get\_current\_span

get\_current\_span() -> Span[Any] | None

Returns the currently active span, if any.

Source code in src/agents/tracing/setup.py

def get\_current\_span(self) -> Span[Any] | None:

"""

Returns the currently active span, if any.

"""

return Scope.get\_current\_span()

set\_disabled

set\_disabled(disabled: bool) -> None

Set whether tracing is disabled.

Source code in src/agents/tracing/setup.py

def set\_disabled(self, disabled: bool) -> None:

"""

Set whether tracing is disabled.

"""

self.\_disabled = disabled

create\_trace

create\_trace(

name: str,

trace\_id: str | None = None,

group\_id: str | None = None,

metadata: dict[str, Any] | None = None,

disabled: bool = False,

) -> Trace

Create a new trace.

Source code in src/agents/tracing/setup.py

def create\_trace(

self,

name: str,

trace\_id: str | None = None,

group\_id: str | None = None,

metadata: dict[str, Any] | None = None,

disabled: bool = False,

) -> Trace:

"""

Create a new trace.

"""

if self.\_disabled or disabled:

logger.debug(f"Tracing is disabled. Not creating trace {name}")

return NoOpTrace()

trace\_id = trace\_id or util.gen\_trace\_id()

logger.debug(f"Creating trace {name} with id {trace\_id}")

return TraceImpl(

name=name,

trace\_id=trace\_id,

group\_id=group\_id,

metadata=metadata,

processor=self.\_multi\_processor,

)

create\_span

create\_span(

span\_data: TSpanData,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[TSpanData]

Create a new span.

Source code in src/agents/tracing/setup.py

def create\_span(

self,

span\_data: TSpanData,

span\_id: str | None = None,

parent: Trace | Span[Any] | None = None,

disabled: bool = False,

) -> Span[TSpanData]:

"""

Create a new span.

"""

if self.\_disabled or disabled:

logger.debug(f"Tracing is disabled. Not creating span {span\_data}")

return NoOpSpan(span\_data)

if not parent:

current\_span = Scope.get\_current\_span()

current\_trace = Scope.get\_current\_trace()

if current\_trace is None:

logger.error(

"No active trace. Make sure to start a trace with `trace()` first"

"Returning NoOpSpan."

)

return NoOpSpan(span\_data)

elif isinstance(current\_trace, NoOpTrace) or isinstance(current\_span, NoOpSpan):

logger.debug(

f"Parent {current\_span} or {current\_trace} is no-op, returning NoOpSpan"

)

return NoOpSpan(span\_data)

parent\_id = current\_span.span\_id if current\_span else None

trace\_id = current\_trace.trace\_id

elif isinstance(parent, Trace):

if isinstance(parent, NoOpTrace):

logger.debug(f"Parent {parent} is no-op, returning NoOpSpan")

return NoOpSpan(span\_data)

trace\_id = parent.trace\_id

parent\_id = None

elif isinstance(parent, Span):

if isinstance(parent, NoOpSpan):

logger.debug(f"Parent {parent} is no-op, returning NoOpSpan")

return NoOpSpan(span\_data)

parent\_id = parent.span\_id

trace\_id = parent.trace\_id

logger.debug(f"Creating span {span\_data} with id {span\_id}")

return SpanImpl(

trace\_id=trace\_id,

span\_id=span\_id,

parent\_id=parent\_id,

processor=self.\_multi\_processor,

span\_data=span\_data,

)

Util

time\_iso

time\_iso() -> str

Returns the current time in ISO 8601 format.

Source code in src/agents/tracing/util.py

def time\_iso() -> str:

"""Returns the current time in ISO 8601 format."""

return datetime.now(timezone.utc).isoformat()

gen\_trace\_id

gen\_trace\_id() -> str

Generates a new trace ID.

Source code in src/agents/tracing/util.py

def gen\_trace\_id() -> str:

"""Generates a new trace ID."""

return f"trace\_{uuid.uuid4().hex}"

gen\_span\_id

gen\_span\_id() -> str

Generates a new span ID.

Source code in src/agents/tracing/util.py

def gen\_span\_id() -> str:

"""Generates a new span ID."""

return f"span\_{uuid.uuid4().hex[:24]}"

gen\_group\_id

gen\_group\_id() -> str

Generates a new group ID.

Source code in src/agents/tracing/util.py

def gen\_group\_id() -> str:

"""Generates a new group ID."""

return f"group\_{uuid.uuid4().hex[:24]}"

Voice:  
Pipeline

VoicePipeline

An opinionated voice agent pipeline. It works in three steps: 1. Transcribe audio input into text. 2. Run the provided workflow, which produces a sequence of text responses. 3. Convert the text responses into streaming audio output.

Source code in src/agents/voice/pipeline.py

class VoicePipeline:

"""An opinionated voice agent pipeline. It works in three steps:

1. Transcribe audio input into text.

2. Run the provided `workflow`, which produces a sequence of text responses.

3. Convert the text responses into streaming audio output.

"""

def \_\_init\_\_(

self,

\*,

workflow: VoiceWorkflowBase,

stt\_model: STTModel | str | None = None,

tts\_model: TTSModel | str | None = None,

config: VoicePipelineConfig | None = None,

):

"""Create a new voice pipeline.

Args:

workflow: The workflow to run. See `VoiceWorkflowBase`.

stt\_model: The speech-to-text model to use. If not provided, a default OpenAI

model will be used.

tts\_model: The text-to-speech model to use. If not provided, a default OpenAI

model will be used.

config: The pipeline configuration. If not provided, a default configuration will be

used.

"""

self.workflow = workflow

self.stt\_model = stt\_model if isinstance(stt\_model, STTModel) else None

self.tts\_model = tts\_model if isinstance(tts\_model, TTSModel) else None

self.\_stt\_model\_name = stt\_model if isinstance(stt\_model, str) else None

self.\_tts\_model\_name = tts\_model if isinstance(tts\_model, str) else None

self.config = config or VoicePipelineConfig()

async def run(self, audio\_input: AudioInput | StreamedAudioInput) -> StreamedAudioResult:

"""Run the voice pipeline.

Args:

audio\_input: The audio input to process. This can either be an `AudioInput` instance,

which is a single static buffer, or a `StreamedAudioInput` instance, which is a

stream of audio data that you can append to.

Returns:

A `StreamedAudioResult` instance. You can use this object to stream audio events and

play them out.

"""

if isinstance(audio\_input, AudioInput):

return await self.\_run\_single\_turn(audio\_input)

elif isinstance(audio\_input, StreamedAudioInput):

return await self.\_run\_multi\_turn(audio\_input)

else:

raise UserError(f"Unsupported audio input type: {type(audio\_input)}")

def \_get\_tts\_model(self) -> TTSModel:

if not self.tts\_model:

self.tts\_model = self.config.model\_provider.get\_tts\_model(self.\_tts\_model\_name)

return self.tts\_model

def \_get\_stt\_model(self) -> STTModel:

if not self.stt\_model:

self.stt\_model = self.config.model\_provider.get\_stt\_model(self.\_stt\_model\_name)

return self.stt\_model

async def \_process\_audio\_input(self, audio\_input: AudioInput) -> str:

model = self.\_get\_stt\_model()

return await model.transcribe(

audio\_input,

self.config.stt\_settings,

self.config.trace\_include\_sensitive\_data,

self.config.trace\_include\_sensitive\_audio\_data,

)

async def \_run\_single\_turn(self, audio\_input: AudioInput) -> StreamedAudioResult:

# Since this is single turn, we can use the TraceCtxManager to manage starting/ending the

# trace

with TraceCtxManager(

workflow\_name=self.config.workflow\_name or "Voice Agent",

trace\_id=None, # Automatically generated

group\_id=self.config.group\_id,

metadata=self.config.trace\_metadata,

disabled=self.config.tracing\_disabled,

):

input\_text = await self.\_process\_audio\_input(audio\_input)

output = StreamedAudioResult(

self.\_get\_tts\_model(), self.config.tts\_settings, self.config

)

async def stream\_events():

try:

async for text\_event in self.workflow.run(input\_text):

await output.\_add\_text(text\_event)

await output.\_turn\_done()

await output.\_done()

except Exception as e:

logger.error(f"Error processing single turn: {e}")

await output.\_add\_error(e)

raise e

output.\_set\_task(asyncio.create\_task(stream\_events()))

return output

async def \_run\_multi\_turn(self, audio\_input: StreamedAudioInput) -> StreamedAudioResult:

with TraceCtxManager(

workflow\_name=self.config.workflow\_name or "Voice Agent",

trace\_id=None,

group\_id=self.config.group\_id,

metadata=self.config.trace\_metadata,

disabled=self.config.tracing\_disabled,

):

output = StreamedAudioResult(

self.\_get\_tts\_model(), self.config.tts\_settings, self.config

)

transcription\_session = await self.\_get\_stt\_model().create\_session(

audio\_input,

self.config.stt\_settings,

self.config.trace\_include\_sensitive\_data,

self.config.trace\_include\_sensitive\_audio\_data,

)

async def process\_turns():

try:

async for input\_text in transcription\_session.transcribe\_turns():

result = self.workflow.run(input\_text)

async for text\_event in result:

await output.\_add\_text(text\_event)

await output.\_turn\_done()

except Exception as e:

logger.error(f"Error processing turns: {e}")

await output.\_add\_error(e)

raise e

finally:

await transcription\_session.close()

await output.\_done()

output.\_set\_task(asyncio.create\_task(process\_turns()))

return output

\_\_init\_\_

\_\_init\_\_(

\*,

workflow: VoiceWorkflowBase,

stt\_model: STTModel | str | None = None,

tts\_model: TTSModel | str | None = None,

config: VoicePipelineConfig | None = None,

)

Create a new voice pipeline.

Parameters:

Name Type Description Default

workflow VoiceWorkflowBase The workflow to run. See VoiceWorkflowBase. required

stt\_model STTModel | str | None The speech-to-text model to use. If not provided, a default OpenAI model will be used. None

tts\_model TTSModel | str | None The text-to-speech model to use. If not provided, a default OpenAI model will be used. None

config VoicePipelineConfig | None The pipeline configuration. If not provided, a default configuration will be used. None

Source code in src/agents/voice/pipeline.py

def \_\_init\_\_(

self,

\*,

workflow: VoiceWorkflowBase,

stt\_model: STTModel | str | None = None,

tts\_model: TTSModel | str | None = None,

config: VoicePipelineConfig | None = None,

):

"""Create a new voice pipeline.

Args:

workflow: The workflow to run. See `VoiceWorkflowBase`.

stt\_model: The speech-to-text model to use. If not provided, a default OpenAI

model will be used.

tts\_model: The text-to-speech model to use. If not provided, a default OpenAI

model will be used.

config: The pipeline configuration. If not provided, a default configuration will be

used.

"""

self.workflow = workflow

self.stt\_model = stt\_model if isinstance(stt\_model, STTModel) else None

self.tts\_model = tts\_model if isinstance(tts\_model, TTSModel) else None

self.\_stt\_model\_name = stt\_model if isinstance(stt\_model, str) else None

self.\_tts\_model\_name = tts\_model if isinstance(tts\_model, str) else None

self.config = config or VoicePipelineConfig()

run async

run(

audio\_input: AudioInput | StreamedAudioInput,

) -> StreamedAudioResult

Run the voice pipeline.

Parameters:

Name Type Description Default

audio\_input AudioInput | StreamedAudioInput The audio input to process. This can either be an AudioInput instance, which is a single static buffer, or a StreamedAudioInput instance, which is a stream of audio data that you can append to. required

Returns:

Type Description

StreamedAudioResult A StreamedAudioResult instance. You can use this object to stream audio events and

StreamedAudioResult play them out.

Source code in src/agents/voice/pipeline.py

async def run(self, audio\_input: AudioInput | StreamedAudioInput) -> StreamedAudioResult:

"""Run the voice pipeline.

Args:

audio\_input: The audio input to process. This can either be an `AudioInput` instance,

which is a single static buffer, or a `StreamedAudioInput` instance, which is a

stream of audio data that you can append to.

Returns:

A `StreamedAudioResult` instance. You can use this object to stream audio events and

play them out.

"""

if isinstance(audio\_input, AudioInput):

return await self.\_run\_single\_turn(audio\_input)

elif isinstance(audio\_input, StreamedAudioInput):

return await self.\_run\_multi\_turn(audio\_input)

else:

raise UserError(f"Unsupported audio input type: {type(audio\_input)}")

Workflow

VoiceWorkflowBase

Bases: ABC

A base class for a voice workflow. You must implement the run method. A "workflow" is any code you want, that receives a transcription and yields text that will be turned into speech by a text-to-speech model. In most cases, you'll create Agents and use Runner.run\_streamed() to run them, returning some or all of the text events from the stream. You can use the VoiceWorkflowHelper class to help with extracting text events from the stream. If you have a simple workflow that has a single starting agent and no custom logic, you can use SingleAgentVoiceWorkflow directly.

Source code in src/agents/voice/workflow.py

class VoiceWorkflowBase(abc.ABC):

"""

A base class for a voice workflow. You must implement the `run` method. A "workflow" is any

code you want, that receives a transcription and yields text that will be turned into speech

by a text-to-speech model.

In most cases, you'll create `Agent`s and use `Runner.run\_streamed()` to run them, returning

some or all of the text events from the stream. You can use the `VoiceWorkflowHelper` class to

help with extracting text events from the stream.

If you have a simple workflow that has a single starting agent and no custom logic, you can

use `SingleAgentVoiceWorkflow` directly.

"""

@abc.abstractmethod

def run(self, transcription: str) -> AsyncIterator[str]:

"""

Run the voice workflow. You will receive an input transcription, and must yield text that

will be spoken to the user. You can run whatever logic you want here. In most cases, the

final logic will involve calling `Runner.run\_streamed()` and yielding any text events from

the stream.

"""

pass

run abstractmethod

run(transcription: str) -> AsyncIterator[str]

Run the voice workflow. You will receive an input transcription, and must yield text that will be spoken to the user. You can run whatever logic you want here. In most cases, the final logic will involve calling Runner.run\_streamed() and yielding any text events from the stream.

Source code in src/agents/voice/workflow.py

@abc.abstractmethod

def run(self, transcription: str) -> AsyncIterator[str]:

"""

Run the voice workflow. You will receive an input transcription, and must yield text that

will be spoken to the user. You can run whatever logic you want here. In most cases, the

final logic will involve calling `Runner.run\_streamed()` and yielding any text events from

the stream.

"""

pass

VoiceWorkflowHelper

Source code in src/agents/voice/workflow.py

class VoiceWorkflowHelper:

@classmethod

async def stream\_text\_from(cls, result: RunResultStreaming) -> AsyncIterator[str]:

"""Wraps a `RunResultStreaming` object and yields text events from the stream."""

async for event in result.stream\_events():

if (

event.type == "raw\_response\_event"

and event.data.type == "response.output\_text.delta"

):

yield event.data.delta

stream\_text\_from async classmethod

stream\_text\_from(

result: RunResultStreaming,

) -> AsyncIterator[str]

Wraps a RunResultStreaming object and yields text events from the stream.

Source code in src/agents/voice/workflow.py

@classmethod

async def stream\_text\_from(cls, result: RunResultStreaming) -> AsyncIterator[str]:

"""Wraps a `RunResultStreaming` object and yields text events from the stream."""

async for event in result.stream\_events():

if (

event.type == "raw\_response\_event"

and event.data.type == "response.output\_text.delta"

):

yield event.data.delta

SingleAgentWorkflowCallbacks

Source code in src/agents/voice/workflow.py

class SingleAgentWorkflowCallbacks:

def on\_run(self, workflow: SingleAgentVoiceWorkflow, transcription: str) -> None:

"""Called when the workflow is run."""

pass

on\_run

on\_run(

workflow: SingleAgentVoiceWorkflow, transcription: str

) -> None

Called when the workflow is run.

Source code in src/agents/voice/workflow.py

def on\_run(self, workflow: SingleAgentVoiceWorkflow, transcription: str) -> None:

"""Called when the workflow is run."""

pass

SingleAgentVoiceWorkflow

Bases: VoiceWorkflowBase

A simple voice workflow that runs a single agent. Each transcription and result is added to the input history. For more complex workflows (e.g. multiple Runner calls, custom message history, custom logic, custom configs), subclass VoiceWorkflowBase and implement your own logic.

Source code in src/agents/voice/workflow.py

class SingleAgentVoiceWorkflow(VoiceWorkflowBase):

"""A simple voice workflow that runs a single agent. Each transcription and result is added to

the input history.

For more complex workflows (e.g. multiple Runner calls, custom message history, custom logic,

custom configs), subclass `VoiceWorkflowBase` and implement your own logic.

"""

def \_\_init\_\_(self, agent: Agent[Any], callbacks: SingleAgentWorkflowCallbacks | None = None):

"""Create a new single agent voice workflow.

Args:

agent: The agent to run.

callbacks: Optional callbacks to call during the workflow.

"""

self.\_input\_history: list[TResponseInputItem] = []

self.\_current\_agent = agent

self.\_callbacks = callbacks

async def run(self, transcription: str) -> AsyncIterator[str]:

if self.\_callbacks:

self.\_callbacks.on\_run(self, transcription)

# Add the transcription to the input history

self.\_input\_history.append(

{

"role": "user",

"content": transcription,

}

)

# Run the agent

result = Runner.run\_streamed(self.\_current\_agent, self.\_input\_history)

# Stream the text from the result

async for chunk in VoiceWorkflowHelper.stream\_text\_from(result):

yield chunk

# Update the input history and current agent

self.\_input\_history = result.to\_input\_list()

self.\_current\_agent = result.last\_agent

\_\_init\_\_

\_\_init\_\_(

agent: Agent[Any],

callbacks: SingleAgentWorkflowCallbacks | None = None,

)

Create a new single agent voice workflow.

Parameters:

Name Type Description Default

agent Agent[Any] The agent to run. required

callbacks SingleAgentWorkflowCallbacks | None Optional callbacks to call during the workflow. None

Source code in src/agents/voice/workflow.py

def \_\_init\_\_(self, agent: Agent[Any], callbacks: SingleAgentWorkflowCallbacks | None = None):

"""Create a new single agent voice workflow.

Args:

agent: The agent to run.

callbacks: Optional callbacks to call during the workflow.

"""

self.\_input\_history: list[TResponseInputItem] = []

self.\_current\_agent = agent

self.\_callbacks = callbacks

Input

AudioInput dataclass

Static audio to be used as input for the VoicePipeline.

Source code in src/agents/voice/input.py

@dataclass

class AudioInput:

"""Static audio to be used as input for the VoicePipeline."""

buffer: npt.NDArray[np.int16 | np.float32]

"""

A buffer containing the audio data for the agent. Must be a numpy array of int16 or float32.

"""

frame\_rate: int = DEFAULT\_SAMPLE\_RATE

"""The sample rate of the audio data. Defaults to 24000."""

sample\_width: int = 2

"""The sample width of the audio data. Defaults to 2."""

channels: int = 1

"""The number of channels in the audio data. Defaults to 1."""

def to\_audio\_file(self) -> tuple[str, io.BytesIO, str]:

"""Returns a tuple of (filename, bytes, content\_type)"""

return \_buffer\_to\_audio\_file(self.buffer, self.frame\_rate, self.sample\_width, self.channels)

def to\_base64(self) -> str:

"""Returns the audio data as a base64 encoded string."""

if self.buffer.dtype == np.float32:

# convert to int16

self.buffer = np.clip(self.buffer, -1.0, 1.0)

self.buffer = (self.buffer \* 32767).astype(np.int16)

elif self.buffer.dtype != np.int16:

raise UserError("Buffer must be a numpy array of int16 or float32")

return base64.b64encode(self.buffer.tobytes()).decode("utf-8")

buffer instance-attribute

buffer: NDArray[int16 | float32]

A buffer containing the audio data for the agent. Must be a numpy array of int16 or float32.

frame\_rate class-attribute instance-attribute

frame\_rate: int = DEFAULT\_SAMPLE\_RATE

The sample rate of the audio data. Defaults to 24000.

sample\_width class-attribute instance-attribute

sample\_width: int = 2

The sample width of the audio data. Defaults to 2.

channels class-attribute instance-attribute

channels: int = 1

The number of channels in the audio data. Defaults to 1.

to\_audio\_file

to\_audio\_file() -> tuple[str, BytesIO, str]

Returns a tuple of (filename, bytes, content\_type)

Source code in src/agents/voice/input.py

def to\_audio\_file(self) -> tuple[str, io.BytesIO, str]:

"""Returns a tuple of (filename, bytes, content\_type)"""

return \_buffer\_to\_audio\_file(self.buffer, self.frame\_rate, self.sample\_width, self.channels)

to\_base64

to\_base64() -> str

Returns the audio data as a base64 encoded string.

Source code in src/agents/voice/input.py

def to\_base64(self) -> str:

"""Returns the audio data as a base64 encoded string."""

if self.buffer.dtype == np.float32:

# convert to int16

self.buffer = np.clip(self.buffer, -1.0, 1.0)

self.buffer = (self.buffer \* 32767).astype(np.int16)

elif self.buffer.dtype != np.int16:

raise UserError("Buffer must be a numpy array of int16 or float32")

return base64.b64encode(self.buffer.tobytes()).decode("utf-8")

StreamedAudioInput

Audio input represented as a stream of audio data. You can pass this to the VoicePipeline and then push audio data into the queue using the add\_audio method.

Source code in src/agents/voice/input.py

class StreamedAudioInput:

"""Audio input represented as a stream of audio data. You can pass this to the `VoicePipeline`

and then push audio data into the queue using the `add\_audio` method.

"""

def \_\_init\_\_(self):

self.queue: asyncio.Queue[npt.NDArray[np.int16 | np.float32]] = asyncio.Queue()

async def add\_audio(self, audio: npt.NDArray[np.int16 | np.float32]):

"""Adds more audio data to the stream.

Args:

audio: The audio data to add. Must be a numpy array of int16 or float32.

"""

await self.queue.put(audio)

add\_audio async

add\_audio(audio: NDArray[int16 | float32])

Adds more audio data to the stream.

Parameters:

Name Type Description Default

audio NDArray[int16 | float32] The audio data to add. Must be a numpy array of int16 or float32. required

Source code in src/agents/voice/input.py

async def add\_audio(self, audio: npt.NDArray[np.int16 | np.float32]):

"""Adds more audio data to the stream.

Args:

audio: The audio data to add. Must be a numpy array of int16 or float32.

"""

await self.queue.put(audio)

Result

StreamedAudioResult

The output of a VoicePipeline. Streams events and audio data as they're generated.

Source code in src/agents/voice/result.py

class StreamedAudioResult:

"""The output of a `VoicePipeline`. Streams events and audio data as they're generated."""

def \_\_init\_\_(

self,

tts\_model: TTSModel,

tts\_settings: TTSModelSettings,

voice\_pipeline\_config: VoicePipelineConfig,

):

"""Create a new `StreamedAudioResult` instance.

Args:

tts\_model: The TTS model to use.

tts\_settings: The TTS settings to use.

voice\_pipeline\_config: The voice pipeline config to use.

"""

self.tts\_model = tts\_model

self.tts\_settings = tts\_settings

self.total\_output\_text = ""

self.instructions = tts\_settings.instructions

self.text\_generation\_task: asyncio.Task[Any] | None = None

self.\_voice\_pipeline\_config = voice\_pipeline\_config

self.\_text\_buffer = ""

self.\_turn\_text\_buffer = ""

self.\_queue: asyncio.Queue[VoiceStreamEvent] = asyncio.Queue()

self.\_tasks: list[asyncio.Task[Any]] = []

self.\_ordered\_tasks: list[

asyncio.Queue[VoiceStreamEvent | None]

] = [] # New: list to hold local queues for each text segment

self.\_dispatcher\_task: asyncio.Task[Any] | None = (

None # Task to dispatch audio chunks in order

)

self.\_done\_processing = False

self.\_buffer\_size = tts\_settings.buffer\_size

self.\_started\_processing\_turn = False

self.\_first\_byte\_received = False

self.\_generation\_start\_time: str | None = None

self.\_completed\_session = False

self.\_stored\_exception: BaseException | None = None

self.\_tracing\_span: Span[SpeechGroupSpanData] | None = None

async def \_start\_turn(self):

if self.\_started\_processing\_turn:

return

self.\_tracing\_span = speech\_group\_span()

self.\_tracing\_span.start()

self.\_started\_processing\_turn = True

self.\_first\_byte\_received = False

self.\_generation\_start\_time = time\_iso()

await self.\_queue.put(VoiceStreamEventLifecycle(event="turn\_started"))

def \_set\_task(self, task: asyncio.Task[Any]):

self.text\_generation\_task = task

async def \_add\_error(self, error: Exception):

await self.\_queue.put(VoiceStreamEventError(error))

def \_transform\_audio\_buffer(

self, buffer: list[bytes], output\_dtype: npt.DTypeLike

) -> npt.NDArray[np.int16 | np.float32]:

np\_array = np.frombuffer(b"".join(buffer), dtype=np.int16)

if output\_dtype == np.int16:

return np\_array

elif output\_dtype == np.float32:

return (np\_array.astype(np.float32) / 32767.0).reshape(-1, 1)

else:

raise UserError("Invalid output dtype")

async def \_stream\_audio(

self,

text: str,

local\_queue: asyncio.Queue[VoiceStreamEvent | None],

finish\_turn: bool = False,

):

with speech\_span(

model=self.tts\_model.model\_name,

input=text if self.\_voice\_pipeline\_config.trace\_include\_sensitive\_data else "",

model\_config={

"voice": self.tts\_settings.voice,

"instructions": self.instructions,

"speed": self.tts\_settings.speed,

},

output\_format="pcm",

parent=self.\_tracing\_span,

) as tts\_span:

try:

first\_byte\_received = False

buffer: list[bytes] = []

full\_audio\_data: list[bytes] = []

async for chunk in self.tts\_model.run(text, self.tts\_settings):

if not first\_byte\_received:

first\_byte\_received = True

tts\_span.span\_data.first\_content\_at = time\_iso()

if chunk:

buffer.append(chunk)

full\_audio\_data.append(chunk)

if len(buffer) >= self.\_buffer\_size:

audio\_np = self.\_transform\_audio\_buffer(buffer, self.tts\_settings.dtype)

if self.tts\_settings.transform\_data:

audio\_np = self.tts\_settings.transform\_data(audio\_np)

await local\_queue.put(

VoiceStreamEventAudio(data=audio\_np)

) # Use local queue

buffer = []

if buffer:

audio\_np = self.\_transform\_audio\_buffer(buffer, self.tts\_settings.dtype)

if self.tts\_settings.transform\_data:

audio\_np = self.tts\_settings.transform\_data(audio\_np)

await local\_queue.put(VoiceStreamEventAudio(data=audio\_np)) # Use local queue

if self.\_voice\_pipeline\_config.trace\_include\_sensitive\_audio\_data:

tts\_span.span\_data.output = \_audio\_to\_base64(full\_audio\_data)

else:

tts\_span.span\_data.output = ""

if finish\_turn:

await local\_queue.put(VoiceStreamEventLifecycle(event="turn\_ended"))

else:

await local\_queue.put(None) # Signal completion for this segment

except Exception as e:

tts\_span.set\_error(

{

"message": str(e),

"data": {

"text": text

if self.\_voice\_pipeline\_config.trace\_include\_sensitive\_data

else "",

},

}

)

logger.error(f"Error streaming audio: {e}")

# Signal completion for whole session because of error

await local\_queue.put(VoiceStreamEventLifecycle(event="session\_ended"))

raise e

async def \_add\_text(self, text: str):

await self.\_start\_turn()

self.\_text\_buffer += text

self.total\_output\_text += text

self.\_turn\_text\_buffer += text

combined\_sentences, self.\_text\_buffer = self.tts\_settings.text\_splitter(self.\_text\_buffer)

if len(combined\_sentences) >= 20:

local\_queue: asyncio.Queue[VoiceStreamEvent | None] = asyncio.Queue()

self.\_ordered\_tasks.append(local\_queue)

self.\_tasks.append(

asyncio.create\_task(self.\_stream\_audio(combined\_sentences, local\_queue))

)

if self.\_dispatcher\_task is None:

self.\_dispatcher\_task = asyncio.create\_task(self.\_dispatch\_audio())

async def \_turn\_done(self):

if self.\_text\_buffer:

local\_queue: asyncio.Queue[VoiceStreamEvent | None] = asyncio.Queue()

self.\_ordered\_tasks.append(local\_queue) # Append the local queue for the final segment

self.\_tasks.append(

asyncio.create\_task(

self.\_stream\_audio(self.\_text\_buffer, local\_queue, finish\_turn=True)

)

)

self.\_text\_buffer = ""

self.\_done\_processing = True

if self.\_dispatcher\_task is None:

self.\_dispatcher\_task = asyncio.create\_task(self.\_dispatch\_audio())

await asyncio.gather(\*self.\_tasks)

def \_finish\_turn(self):

if self.\_tracing\_span:

if self.\_voice\_pipeline\_config.trace\_include\_sensitive\_data:

self.\_tracing\_span.span\_data.input = self.\_turn\_text\_buffer

else:

self.\_tracing\_span.span\_data.input = ""

self.\_tracing\_span.finish()

self.\_tracing\_span = None

self.\_turn\_text\_buffer = ""

self.\_started\_processing\_turn = False

async def \_done(self):

self.\_completed\_session = True

await self.\_wait\_for\_completion()

async def \_dispatch\_audio(self):

# Dispatch audio chunks from each segment in the order they were added

while True:

if len(self.\_ordered\_tasks) == 0:

if self.\_completed\_session:

break

await asyncio.sleep(0)

continue

local\_queue = self.\_ordered\_tasks.pop(0)

while True:

chunk = await local\_queue.get()

if chunk is None:

break

await self.\_queue.put(chunk)

if isinstance(chunk, VoiceStreamEventLifecycle):

local\_queue.task\_done()

if chunk.event == "turn\_ended":

self.\_finish\_turn()

break

await self.\_queue.put(VoiceStreamEventLifecycle(event="session\_ended"))

async def \_wait\_for\_completion(self):

tasks: list[asyncio.Task[Any]] = self.\_tasks

if self.\_dispatcher\_task is not None:

tasks.append(self.\_dispatcher\_task)

await asyncio.gather(\*tasks)

def \_cleanup\_tasks(self):

self.\_finish\_turn()

for task in self.\_tasks:

if not task.done():

task.cancel()

if self.\_dispatcher\_task and not self.\_dispatcher\_task.done():

self.\_dispatcher\_task.cancel()

if self.text\_generation\_task and not self.text\_generation\_task.done():

self.text\_generation\_task.cancel()

def \_check\_errors(self):

for task in self.\_tasks:

if task.done():

if task.exception():

self.\_stored\_exception = task.exception()

break

async def stream(self) -> AsyncIterator[VoiceStreamEvent]:

"""Stream the events and audio data as they're generated."""

while True:

try:

event = await self.\_queue.get()

except asyncio.CancelledError:

break

if isinstance(event, VoiceStreamEventError):

self.\_stored\_exception = event.error

logger.error(f"Error processing output: {event.error}")

break

if event is None:

break

yield event

if event.type == "voice\_stream\_event\_lifecycle" and event.event == "session\_ended":

break

self.\_check\_errors()

self.\_cleanup\_tasks()

if self.\_stored\_exception:

raise self.\_stored\_exception

\_\_init\_\_

\_\_init\_\_(

tts\_model: TTSModel,

tts\_settings: TTSModelSettings,

voice\_pipeline\_config: VoicePipelineConfig,

)

Create a new StreamedAudioResult instance.

Parameters:

Name Type Description Default

tts\_model TTSModel The TTS model to use. required

tts\_settings TTSModelSettings The TTS settings to use. required

voice\_pipeline\_config VoicePipelineConfig The voice pipeline config to use. required

Source code in src/agents/voice/result.py

def \_\_init\_\_(

self,

tts\_model: TTSModel,

tts\_settings: TTSModelSettings,

voice\_pipeline\_config: VoicePipelineConfig,

):

"""Create a new `StreamedAudioResult` instance.

Args:

tts\_model: The TTS model to use.

tts\_settings: The TTS settings to use.

voice\_pipeline\_config: The voice pipeline config to use.

"""

self.tts\_model = tts\_model

self.tts\_settings = tts\_settings

self.total\_output\_text = ""

self.instructions = tts\_settings.instructions

self.text\_generation\_task: asyncio.Task[Any] | None = None

self.\_voice\_pipeline\_config = voice\_pipeline\_config

self.\_text\_buffer = ""

self.\_turn\_text\_buffer = ""

self.\_queue: asyncio.Queue[VoiceStreamEvent] = asyncio.Queue()

self.\_tasks: list[asyncio.Task[Any]] = []

self.\_ordered\_tasks: list[

asyncio.Queue[VoiceStreamEvent | None]

] = [] # New: list to hold local queues for each text segment

self.\_dispatcher\_task: asyncio.Task[Any] | None = (

None # Task to dispatch audio chunks in order

)

self.\_done\_processing = False

self.\_buffer\_size = tts\_settings.buffer\_size

self.\_started\_processing\_turn = False

self.\_first\_byte\_received = False

self.\_generation\_start\_time: str | None = None

self.\_completed\_session = False

self.\_stored\_exception: BaseException | None = None

self.\_tracing\_span: Span[SpeechGroupSpanData] | None = None

stream async

stream() -> AsyncIterator[VoiceStreamEvent]

Stream the events and audio data as they're generated.

Source code in src/agents/voice/result.py

async def stream(self) -> AsyncIterator[VoiceStreamEvent]:

"""Stream the events and audio data as they're generated."""

while True:

try:

event = await self.\_queue.get()

except asyncio.CancelledError:

break

if isinstance(event, VoiceStreamEventError):

self.\_stored\_exception = event.error

logger.error(f"Error processing output: {event.error}")

break

if event is None:

break

yield event

if event.type == "voice\_stream\_event\_lifecycle" and event.event == "session\_ended":

break

self.\_check\_errors()

self.\_cleanup\_tasks()

if self.\_stored\_exception:

raise self.\_stored\_exception

Pipeline Config

VoicePipelineConfig dataclass

Configuration for a VoicePipeline.

Source code in src/agents/voice/pipeline\_config.py

@dataclass

class VoicePipelineConfig:

"""Configuration for a `VoicePipeline`."""

model\_provider: VoiceModelProvider = field(default\_factory=OpenAIVoiceModelProvider)

"""The voice model provider to use for the pipeline. Defaults to OpenAI."""

tracing\_disabled: bool = False

"""Whether to disable tracing of the pipeline. Defaults to `False`."""

trace\_include\_sensitive\_data: bool = True

"""Whether to include sensitive data in traces. Defaults to `True`. This is specifically for the

voice pipeline, and not for anything that goes on inside your Workflow."""

trace\_include\_sensitive\_audio\_data: bool = True

"""Whether to include audio data in traces. Defaults to `True`."""

workflow\_name: str = "Voice Agent"

"""The name of the workflow to use for tracing. Defaults to `Voice Agent`."""

group\_id: str = field(default\_factory=gen\_group\_id)

"""

A grouping identifier to use for tracing, to link multiple traces from the same conversation

or process. If not provided, we will create a random group ID.

"""

trace\_metadata: dict[str, Any] | None = None

"""

An optional dictionary of additional metadata to include with the trace.

"""

stt\_settings: STTModelSettings = field(default\_factory=STTModelSettings)

"""The settings to use for the STT model."""

tts\_settings: TTSModelSettings = field(default\_factory=TTSModelSettings)

"""The settings to use for the TTS model."""

model\_provider class-attribute instance-attribute

model\_provider: VoiceModelProvider = field(

default\_factory=OpenAIVoiceModelProvider

)

The voice model provider to use for the pipeline. Defaults to OpenAI.

tracing\_disabled class-attribute instance-attribute

tracing\_disabled: bool = False

Whether to disable tracing of the pipeline. Defaults to False.

trace\_include\_sensitive\_data class-attribute instance-attribute

trace\_include\_sensitive\_data: bool = True

Whether to include sensitive data in traces. Defaults to True. This is specifically for the voice pipeline, and not for anything that goes on inside your Workflow.

trace\_include\_sensitive\_audio\_data class-attribute instance-attribute

trace\_include\_sensitive\_audio\_data: bool = True

Whether to include audio data in traces. Defaults to True.

workflow\_name class-attribute instance-attribute

workflow\_name: str = 'Voice Agent'

The name of the workflow to use for tracing. Defaults to Voice Agent.

group\_id class-attribute instance-attribute

group\_id: str = field(default\_factory=gen\_group\_id)

A grouping identifier to use for tracing, to link multiple traces from the same conversation or process. If not provided, we will create a random group ID.

trace\_metadata class-attribute instance-attribute

trace\_metadata: dict[str, Any] | None = None

An optional dictionary of additional metadata to include with the trace.

stt\_settings class-attribute instance-attribute

stt\_settings: STTModelSettings = field(

default\_factory=STTModelSettings

)

The settings to use for the STT model.

tts\_settings class-attribute instance-attribute

tts\_settings: TTSModelSettings = field(

default\_factory=TTSModelSettings

)

Events

VoiceStreamEvent module-attribute

VoiceStreamEvent: TypeAlias = Union[

VoiceStreamEventAudio,

VoiceStreamEventLifecycle,

VoiceStreamEventError,

]

An event from the VoicePipeline, streamed via StreamedAudioResult.stream().

VoiceStreamEventAudio dataclass

Streaming event from the VoicePipeline

Source code in src/agents/voice/events.py

@dataclass

class VoiceStreamEventAudio:

"""Streaming event from the VoicePipeline"""

data: npt.NDArray[np.int16 | np.float32] | None

"""The audio data."""

type: Literal["voice\_stream\_event\_audio"] = "voice\_stream\_event\_audio"

"""The type of event."""

data instance-attribute

data: NDArray[int16 | float32] | None

The audio data.

type class-attribute instance-attribute

type: Literal["voice\_stream\_event\_audio"] = (

"voice\_stream\_event\_audio"

)

The type of event.

VoiceStreamEventLifecycle dataclass

Streaming event from the VoicePipeline

Source code in src/agents/voice/events.py

@dataclass

class VoiceStreamEventLifecycle:

"""Streaming event from the VoicePipeline"""

event: Literal["turn\_started", "turn\_ended", "session\_ended"]

"""The event that occurred."""

type: Literal["voice\_stream\_event\_lifecycle"] = "voice\_stream\_event\_lifecycle"

"""The type of event."""

event instance-attribute

event: Literal[

"turn\_started", "turn\_ended", "session\_ended"

]

The event that occurred.

type class-attribute instance-attribute

type: Literal["voice\_stream\_event\_lifecycle"] = (

"voice\_stream\_event\_lifecycle"

)

The type of event.

VoiceStreamEventError dataclass

Streaming event from the VoicePipeline

Source code in src/agents/voice/events.py

@dataclass

class VoiceStreamEventError:

"""Streaming event from the VoicePipeline"""

error: Exception

"""The error that occurred."""

type: Literal["voice\_stream\_event\_error"] = "voice\_stream\_event\_error"

"""The type of event."""

error instance-attribute

error: Exception

The error that occurred.

type class-attribute instance-attribute

type: Literal["voice\_stream\_event\_error"] = (

"voice\_stream\_event\_error"

)

Exceptions

STTWebsocketConnectionError

Bases: AgentsException

Exception raised when the STT websocket connection fails.

Source code in src/agents/voice/exceptions.py

class STTWebsocketConnectionError(AgentsException):

"""Exception raised when the STT websocket connection fails."""

def \_\_init\_\_(self, message: str):

self.message = message

Model

TTSModelSettings dataclass

Settings for a TTS model.

Source code in src/agents/voice/model.py

@dataclass

class TTSModelSettings:

"""Settings for a TTS model."""

voice: (

Literal["alloy", "ash", "coral", "echo", "fable", "onyx", "nova", "sage", "shimmer"] | None

) = None

"""

The voice to use for the TTS model. If not provided, the default voice for the respective model

will be used.

"""

buffer\_size: int = 120

"""The minimal size of the chunks of audio data that are being streamed out."""

dtype: npt.DTypeLike = np.int16

"""The data type for the audio data to be returned in."""

transform\_data: (

Callable[[npt.NDArray[np.int16 | np.float32]], npt.NDArray[np.int16 | np.float32]] | None

) = None

"""

A function to transform the data from the TTS model. This is useful if you want the resulting

audio stream to have the data in a specific shape already.

"""

instructions: str = (

"You will receive partial sentences. Do not complete the sentence just read out the text."

)

"""

The instructions to use for the TTS model. This is useful if you want to control the tone of the

audio output.

"""

text\_splitter: Callable[[str], tuple[str, str]] = get\_sentence\_based\_splitter()

"""

A function to split the text into chunks. This is useful if you want to split the text into

chunks before sending it to the TTS model rather than waiting for the whole text to be

processed.

"""

speed: float | None = None

"""The speed with which the TTS model will read the text. Between 0.25 and 4.0."""

voice class-attribute instance-attribute

voice: (

Literal[

"alloy",

"ash",

"coral",

"echo",

"fable",

"onyx",

"nova",

"sage",

"shimmer",

]

| None

) = None

The voice to use for the TTS model. If not provided, the default voice for the respective model will be used.

buffer\_size class-attribute instance-attribute

buffer\_size: int = 120

The minimal size of the chunks of audio data that are being streamed out.

dtype class-attribute instance-attribute

dtype: DTypeLike = int16

The data type for the audio data to be returned in.

transform\_data class-attribute instance-attribute

transform\_data: (

Callable[

[NDArray[int16 | float32]], NDArray[int16 | float32]

]

| None

) = None

A function to transform the data from the TTS model. This is useful if you want the resulting audio stream to have the data in a specific shape already.

instructions class-attribute instance-attribute

instructions: str = "You will receive partial sentences. Do not complete the sentence just read out the text."

The instructions to use for the TTS model. This is useful if you want to control the tone of the audio output.

text\_splitter class-attribute instance-attribute

text\_splitter: Callable[[str], tuple[str, str]] = (

get\_sentence\_based\_splitter()

)

A function to split the text into chunks. This is useful if you want to split the text into chunks before sending it to the TTS model rather than waiting for the whole text to be processed.

speed class-attribute instance-attribute

speed: float | None = None

The speed with which the TTS model will read the text. Between 0.25 and 4.0.

TTSModel

Bases: ABC

A text-to-speech model that can convert text into audio output.

Source code in src/agents/voice/model.py

class TTSModel(abc.ABC):

"""A text-to-speech model that can convert text into audio output."""

@property

@abc.abstractmethod

def model\_name(self) -> str:

"""The name of the TTS model."""

pass

@abc.abstractmethod

def run(self, text: str, settings: TTSModelSettings) -> AsyncIterator[bytes]:

"""Given a text string, produces a stream of audio bytes, in PCM format.

Args:

text: The text to convert to audio.

Returns:

An async iterator of audio bytes, in PCM format.

"""

pass

model\_name abstractmethod property

model\_name: str

The name of the TTS model.

run abstractmethod

run(

text: str, settings: TTSModelSettings

) -> AsyncIterator[bytes]

Given a text string, produces a stream of audio bytes, in PCM format.

Parameters:

Name Type Description Default

text str The text to convert to audio. required

Returns:

Type Description

AsyncIterator[bytes] An async iterator of audio bytes, in PCM format.

Source code in src/agents/voice/model.py

@abc.abstractmethod

def run(self, text: str, settings: TTSModelSettings) -> AsyncIterator[bytes]:

"""Given a text string, produces a stream of audio bytes, in PCM format.

Args:

text: The text to convert to audio.

Returns:

An async iterator of audio bytes, in PCM format.

"""

pass

StreamedTranscriptionSession

Bases: ABC

A streamed transcription of audio input.

Source code in src/agents/voice/model.py

class StreamedTranscriptionSession(abc.ABC):

"""A streamed transcription of audio input."""

@abc.abstractmethod

def transcribe\_turns(self) -> AsyncIterator[str]:

"""Yields a stream of text transcriptions. Each transcription is a turn in the conversation.

This method is expected to return only after `close()` is called.

"""

pass

@abc.abstractmethod

async def close(self) -> None:

"""Closes the session."""

pass

transcribe\_turns abstractmethod

transcribe\_turns() -> AsyncIterator[str]

Yields a stream of text transcriptions. Each transcription is a turn in the conversation.

This method is expected to return only after close() is called.

Source code in src/agents/voice/model.py

@abc.abstractmethod

def transcribe\_turns(self) -> AsyncIterator[str]:

"""Yields a stream of text transcriptions. Each transcription is a turn in the conversation.

This method is expected to return only after `close()` is called.

"""

pass

close abstractmethod async

close() -> None

Closes the session.

Source code in src/agents/voice/model.py

@abc.abstractmethod

async def close(self) -> None:

"""Closes the session."""

pass

STTModelSettings dataclass

Settings for a speech-to-text model.

Source code in src/agents/voice/model.py

@dataclass

class STTModelSettings:

"""Settings for a speech-to-text model."""

prompt: str | None = None

"""Instructions for the model to follow."""

language: str | None = None

"""The language of the audio input."""

temperature: float | None = None

"""The temperature of the model."""

turn\_detection: dict[str, Any] | None = None

"""The turn detection settings for the model when using streamed audio input."""

prompt class-attribute instance-attribute

prompt: str | None = None

Instructions for the model to follow.

language class-attribute instance-attribute

language: str | None = None

The language of the audio input.

temperature class-attribute instance-attribute

temperature: float | None = None

The temperature of the model.

turn\_detection class-attribute instance-attribute

turn\_detection: dict[str, Any] | None = None

The turn detection settings for the model when using streamed audio input.

STTModel

Bases: ABC

A speech-to-text model that can convert audio input into text.

Source code in src/agents/voice/model.py

class STTModel(abc.ABC):

"""A speech-to-text model that can convert audio input into text."""

@property

@abc.abstractmethod

def model\_name(self) -> str:

"""The name of the STT model."""

pass

@abc.abstractmethod

async def transcribe(

self,

input: AudioInput,

settings: STTModelSettings,

trace\_include\_sensitive\_data: bool,

trace\_include\_sensitive\_audio\_data: bool,

) -> str:

"""Given an audio input, produces a text transcription.

Args:

input: The audio input to transcribe.

settings: The settings to use for the transcription.

trace\_include\_sensitive\_data: Whether to include sensitive data in traces.

trace\_include\_sensitive\_audio\_data: Whether to include sensitive audio data in traces.

Returns:

The text transcription of the audio input.

"""

pass

@abc.abstractmethod

async def create\_session(

self,

input: StreamedAudioInput,

settings: STTModelSettings,

trace\_include\_sensitive\_data: bool,

trace\_include\_sensitive\_audio\_data: bool,

) -> StreamedTranscriptionSession:

"""Creates a new transcription session, which you can push audio to, and receive a stream

of text transcriptions.

Args:

input: The audio input to transcribe.

settings: The settings to use for the transcription.

trace\_include\_sensitive\_data: Whether to include sensitive data in traces.

trace\_include\_sensitive\_audio\_data: Whether to include sensitive audio data in traces.

Returns:

A new transcription session.

"""

pass

model\_name abstractmethod property

model\_name: str

The name of the STT model.

transcribe abstractmethod async

transcribe(

input: AudioInput,

settings: STTModelSettings,

trace\_include\_sensitive\_data: bool,

trace\_include\_sensitive\_audio\_data: bool,

) -> str

Given an audio input, produces a text transcription.

Parameters:

Name Type Description Default

input AudioInput The audio input to transcribe. required

settings STTModelSettings The settings to use for the transcription. required

trace\_include\_sensitive\_data bool Whether to include sensitive data in traces. required

trace\_include\_sensitive\_audio\_data bool Whether to include sensitive audio data in traces. required

Returns:

Type Description

str The text transcription of the audio input.

Source code in src/agents/voice/model.py

@abc.abstractmethod

async def transcribe(

self,

input: AudioInput,

settings: STTModelSettings,

trace\_include\_sensitive\_data: bool,

trace\_include\_sensitive\_audio\_data: bool,

) -> str:

"""Given an audio input, produces a text transcription.

Args:

input: The audio input to transcribe.

settings: The settings to use for the transcription.

trace\_include\_sensitive\_data: Whether to include sensitive data in traces.

trace\_include\_sensitive\_audio\_data: Whether to include sensitive audio data in traces.

Returns:

The text transcription of the audio input.

"""

pass

create\_session abstractmethod async

create\_session(

input: StreamedAudioInput,

settings: STTModelSettings,

trace\_include\_sensitive\_data: bool,

trace\_include\_sensitive\_audio\_data: bool,

) -> StreamedTranscriptionSession

Creates a new transcription session, which you can push audio to, and receive a stream of text transcriptions.

Parameters:

Name Type Description Default

input StreamedAudioInput The audio input to transcribe. required

settings STTModelSettings The settings to use for the transcription. required

trace\_include\_sensitive\_data bool Whether to include sensitive data in traces. required

trace\_include\_sensitive\_audio\_data bool Whether to include sensitive audio data in traces. required

Returns:

Type Description

StreamedTranscriptionSession A new transcription session.

Source code in src/agents/voice/model.py

@abc.abstractmethod

async def create\_session(

self,

input: StreamedAudioInput,

settings: STTModelSettings,

trace\_include\_sensitive\_data: bool,

trace\_include\_sensitive\_audio\_data: bool,

) -> StreamedTranscriptionSession:

"""Creates a new transcription session, which you can push audio to, and receive a stream

of text transcriptions.

Args:

input: The audio input to transcribe.

settings: The settings to use for the transcription.

trace\_include\_sensitive\_data: Whether to include sensitive data in traces.

trace\_include\_sensitive\_audio\_data: Whether to include sensitive audio data in traces.

Returns:

A new transcription session.

"""

pass

VoiceModelProvider

Bases: ABC

The base interface for a voice model provider.

A model provider is responsible for creating speech-to-text and text-to-speech models, given a name.

Source code in src/agents/voice/model.py

class VoiceModelProvider(abc.ABC):

"""The base interface for a voice model provider.

A model provider is responsible for creating speech-to-text and text-to-speech models, given a

name.

"""

@abc.abstractmethod

def get\_stt\_model(self, model\_name: str | None) -> STTModel:

"""Get a speech-to-text model by name.

Args:

model\_name: The name of the model to get.

Returns:

The speech-to-text model.

"""

pass

@abc.abstractmethod

def get\_tts\_model(self, model\_name: str | None) -> TTSModel:

"""Get a text-to-speech model by name."""

get\_stt\_model abstractmethod

get\_stt\_model(model\_name: str | None) -> STTModel

Get a speech-to-text model by name.

Parameters:

Name Type Description Default

model\_name str | None The name of the model to get. required

Returns:

Type Description

STTModel The speech-to-text model.

Source code in src/agents/voice/model.py

@abc.abstractmethod

def get\_stt\_model(self, model\_name: str | None) -> STTModel:

"""Get a speech-to-text model by name.

Args:

model\_name: The name of the model to get.

Returns:

The speech-to-text model.

"""

pass

get\_tts\_model abstractmethod

get\_tts\_model(model\_name: str | None) -> TTSModel

Get a text-to-speech model by name.

Source code in src/agents/voice/model.py

@abc.abstractmethod

def get\_tts\_model(self, model\_name: str | None) -> TTSModel:

"""Get a text-to-speech model by name."""

Utils

get\_sentence\_based\_splitter

get\_sentence\_based\_splitter(

min\_sentence\_length: int = 20,

) -> Callable[[str], tuple[str, str]]

Returns a function that splits text into chunks based on sentence boundaries.

Parameters:

Name Type Description Default

min\_sentence\_length int The minimum length of a sentence to be included in a chunk. 20

Returns:

Type Description

Callable[[str], tuple[str, str]] A function that splits text into chunks based on sentence boundaries.

Source code in src/agents/voice/utils.py

def get\_sentence\_based\_splitter(

min\_sentence\_length: int = 20,

) -> Callable[[str], tuple[str, str]]:

"""Returns a function that splits text into chunks based on sentence boundaries.

Args:

min\_sentence\_length: The minimum length of a sentence to be included in a chunk.

Returns:

A function that splits text into chunks based on sentence boundaries.

"""

def sentence\_based\_text\_splitter(text\_buffer: str) -> tuple[str, str]:

"""

A function to split the text into chunks. This is useful if you want to split the text into

chunks before sending it to the TTS model rather than waiting for the whole text to be

processed.

Args:

text\_buffer: The text to split.

Returns:

A tuple of the text to process and the remaining text buffer.

"""

sentences = re.split(r"(?<=[.!?])\s+", text\_buffer.strip())

if len(sentences) >= 1:

combined\_sentences = " ".join(sentences[:-1])

if len(combined\_sentences) >= min\_sentence\_length:

remaining\_text\_buffer = sentences[-1]

return combined\_sentences, remaining\_text\_buffer

return "", text\_buffer

return sentence\_based\_text\_splitter

OpenAIVoiceModelProvider

OpenAIVoiceModelProvider

Bases: VoiceModelProvider

A voice model provider that uses OpenAI models.

Source code in src/agents/voice/models/openai\_model\_provider.py

class OpenAIVoiceModelProvider(VoiceModelProvider):

"""A voice model provider that uses OpenAI models."""

def \_\_init\_\_(

self,

\*,

api\_key: str | None = None,

base\_url: str | None = None,

openai\_client: AsyncOpenAI | None = None,

organization: str | None = None,

project: str | None = None,

) -> None:

"""Create a new OpenAI voice model provider.

Args:

api\_key: The API key to use for the OpenAI client. If not provided, we will use the

default API key.

base\_url: The base URL to use for the OpenAI client. If not provided, we will use the

default base URL.

openai\_client: An optional OpenAI client to use. If not provided, we will create a new

OpenAI client using the api\_key and base\_url.

organization: The organization to use for the OpenAI client.

project: The project to use for the OpenAI client.

"""

if openai\_client is not None:

assert api\_key is None and base\_url is None, (

"Don't provide api\_key or base\_url if you provide openai\_client"

)

self.\_client: AsyncOpenAI | None = openai\_client

else:

self.\_client = None

self.\_stored\_api\_key = api\_key

self.\_stored\_base\_url = base\_url

self.\_stored\_organization = organization

self.\_stored\_project = project

# We lazy load the client in case you never actually use OpenAIProvider(). Otherwise

# AsyncOpenAI() raises an error if you don't have an API key set.

def \_get\_client(self) -> AsyncOpenAI:

if self.\_client is None:

self.\_client = \_openai\_shared.get\_default\_openai\_client() or AsyncOpenAI(

api\_key=self.\_stored\_api\_key or \_openai\_shared.get\_default\_openai\_key(),

base\_url=self.\_stored\_base\_url,

organization=self.\_stored\_organization,

project=self.\_stored\_project,

http\_client=shared\_http\_client(),

)

return self.\_client

def get\_stt\_model(self, model\_name: str | None) -> STTModel:

"""Get a speech-to-text model by name.

Args:

model\_name: The name of the model to get.

Returns:

The speech-to-text model.

"""

return OpenAISTTModel(model\_name or DEFAULT\_STT\_MODEL, self.\_get\_client())

def get\_tts\_model(self, model\_name: str | None) -> TTSModel:

"""Get a text-to-speech model by name.

Args:

model\_name: The name of the model to get.

Returns:

The text-to-speech model.

"""

return OpenAITTSModel(model\_name or DEFAULT\_TTS\_MODEL, self.\_get\_client())

\_\_init\_\_

\_\_init\_\_(

\*,

api\_key: str | None = None,

base\_url: str | None = None,

openai\_client: AsyncOpenAI | None = None,

organization: str | None = None,

project: str | None = None,

) -> None

Create a new OpenAI voice model provider.

Parameters:

Name Type Description Default

api\_key str | None The API key to use for the OpenAI client. If not provided, we will use the default API key. None

base\_url str | None The base URL to use for the OpenAI client. If not provided, we will use the default base URL. None

openai\_client AsyncOpenAI | None An optional OpenAI client to use. If not provided, we will create a new OpenAI client using the api\_key and base\_url. None

organization str | None The organization to use for the OpenAI client. None

project str | None The project to use for the OpenAI client. None

Source code in src/agents/voice/models/openai\_model\_provider.py

def \_\_init\_\_(

self,

\*,

api\_key: str | None = None,

base\_url: str | None = None,

openai\_client: AsyncOpenAI | None = None,

organization: str | None = None,

project: str | None = None,

) -> None:

"""Create a new OpenAI voice model provider.

Args:

api\_key: The API key to use for the OpenAI client. If not provided, we will use the

default API key.

base\_url: The base URL to use for the OpenAI client. If not provided, we will use the

default base URL.

openai\_client: An optional OpenAI client to use. If not provided, we will create a new

OpenAI client using the api\_key and base\_url.

organization: The organization to use for the OpenAI client.

project: The project to use for the OpenAI client.

"""

if openai\_client is not None:

assert api\_key is None and base\_url is None, (

"Don't provide api\_key or base\_url if you provide openai\_client"

)

self.\_client: AsyncOpenAI | None = openai\_client

else:

self.\_client = None

self.\_stored\_api\_key = api\_key

self.\_stored\_base\_url = base\_url

self.\_stored\_organization = organization

self.\_stored\_project = project

get\_stt\_model

get\_stt\_model(model\_name: str | None) -> STTModel

Get a speech-to-text model by name.

Parameters:

Name Type Description Default

model\_name str | None The name of the model to get. required

Returns:

Type Description

STTModel The speech-to-text model.

Source code in src/agents/voice/models/openai\_model\_provider.py

def get\_stt\_model(self, model\_name: str | None) -> STTModel:

"""Get a speech-to-text model by name.

Args:

model\_name: The name of the model to get.

Returns:

The speech-to-text model.

"""

return OpenAISTTModel(model\_name or DEFAULT\_STT\_MODEL, self.\_get\_client())

get\_tts\_model

get\_tts\_model(model\_name: str | None) -> TTSModel

Get a text-to-speech model by name.

Parameters:

Name Type Description Default

model\_name str | None The name of the model to get. required

Returns:

Type Description

TTSModel The text-to-speech model.

Source code in src/agents/voice/models/openai\_model\_provider.py

def get\_tts\_model(self, model\_name: str | None) -> TTSModel:

"""Get a text-to-speech model by name.

Args:

model\_name: The name of the model to get.

Returns:

The text-to-speech model.

"""

return OpenAITTSModel(model\_name or DEFAULT\_TTS\_MODEL, self.\_get\_client())

OpenAI STT

OpenAISTTTranscriptionSession

Bases: StreamedTranscriptionSession

A transcription session for OpenAI's STT model.

Source code in src/agents/voice/models/openai\_stt.py

class OpenAISTTTranscriptionSession(StreamedTranscriptionSession):

"""A transcription session for OpenAI's STT model."""

def \_\_init\_\_(

self,

input: StreamedAudioInput,

client: AsyncOpenAI,

model: str,

settings: STTModelSettings,

trace\_include\_sensitive\_data: bool,

trace\_include\_sensitive\_audio\_data: bool,

):

self.connected: bool = False

self.\_client = client

self.\_model = model

self.\_settings = settings

self.\_turn\_detection = settings.turn\_detection or DEFAULT\_TURN\_DETECTION

self.\_trace\_include\_sensitive\_data = trace\_include\_sensitive\_data

self.\_trace\_include\_sensitive\_audio\_data = trace\_include\_sensitive\_audio\_data

self.\_input\_queue: asyncio.Queue[npt.NDArray[np.int16 | np.float32]] = input.queue

self.\_output\_queue: asyncio.Queue[str | ErrorSentinel | SessionCompleteSentinel] = (

asyncio.Queue()

)

self.\_websocket: websockets.ClientConnection | None = None

self.\_event\_queue: asyncio.Queue[dict[str, Any] | WebsocketDoneSentinel] = asyncio.Queue()

self.\_state\_queue: asyncio.Queue[dict[str, Any]] = asyncio.Queue()

self.\_turn\_audio\_buffer: list[npt.NDArray[np.int16 | np.float32]] = []

self.\_tracing\_span: Span[TranscriptionSpanData] | None = None

# tasks

self.\_listener\_task: asyncio.Task[Any] | None = None

self.\_process\_events\_task: asyncio.Task[Any] | None = None

self.\_stream\_audio\_task: asyncio.Task[Any] | None = None

self.\_connection\_task: asyncio.Task[Any] | None = None

self.\_stored\_exception: Exception | None = None

def \_start\_turn(self) -> None:

self.\_tracing\_span = transcription\_span(

model=self.\_model,

model\_config={

"temperature": self.\_settings.temperature,

"language": self.\_settings.language,

"prompt": self.\_settings.prompt,

"turn\_detection": self.\_turn\_detection,

},

)

self.\_tracing\_span.start()

def \_end\_turn(self, \_transcript: str) -> None:

if len(\_transcript) < 1:

return

if self.\_tracing\_span:

if self.\_trace\_include\_sensitive\_audio\_data:

self.\_tracing\_span.span\_data.input = \_audio\_to\_base64(self.\_turn\_audio\_buffer)

self.\_tracing\_span.span\_data.input\_format = "pcm"

if self.\_trace\_include\_sensitive\_data:

self.\_tracing\_span.span\_data.output = \_transcript

self.\_tracing\_span.finish()

self.\_turn\_audio\_buffer = []

self.\_tracing\_span = None

async def \_event\_listener(self) -> None:

assert self.\_websocket is not None, "Websocket not initialized"

async for message in self.\_websocket:

try:

event = json.loads(message)

if event.get("type") == "error":

raise STTWebsocketConnectionError(f"Error event: {event.get('error')}")

if event.get("type") in [

"session.updated",

"transcription\_session.updated",

"session.created",

"transcription\_session.created",

]:

await self.\_state\_queue.put(event)

await self.\_event\_queue.put(event)

except Exception as e:

await self.\_output\_queue.put(ErrorSentinel(e))

raise STTWebsocketConnectionError("Error parsing events") from e

await self.\_event\_queue.put(WebsocketDoneSentinel())

async def \_configure\_session(self) -> None:

assert self.\_websocket is not None, "Websocket not initialized"

await self.\_websocket.send(

json.dumps(

{

"type": "transcription\_session.update",

"session": {

"input\_audio\_format": "pcm16",

"input\_audio\_transcription": {"model": self.\_model},

"turn\_detection": self.\_turn\_detection,

},

}

)

)

async def \_setup\_connection(self, ws: websockets.ClientConnection) -> None:

self.\_websocket = ws

self.\_listener\_task = asyncio.create\_task(self.\_event\_listener())

try:

event = await \_wait\_for\_event(

self.\_state\_queue,

["session.created", "transcription\_session.created"],

SESSION\_CREATION\_TIMEOUT,

)

except TimeoutError as e:

wrapped\_err = STTWebsocketConnectionError(

"Timeout waiting for transcription\_session.created event"

)

await self.\_output\_queue.put(ErrorSentinel(wrapped\_err))

raise wrapped\_err from e

except Exception as e:

await self.\_output\_queue.put(ErrorSentinel(e))

raise e

await self.\_configure\_session()

try:

event = await \_wait\_for\_event(

self.\_state\_queue,

["session.updated", "transcription\_session.updated"],

SESSION\_UPDATE\_TIMEOUT,

)

if \_debug.DONT\_LOG\_MODEL\_DATA:

logger.debug("Session updated")

else:

logger.debug(f"Session updated: {event}")

except TimeoutError as e:

wrapped\_err = STTWebsocketConnectionError(

"Timeout waiting for transcription\_session.updated event"

)

await self.\_output\_queue.put(ErrorSentinel(wrapped\_err))

raise wrapped\_err from e

except Exception as e:

await self.\_output\_queue.put(ErrorSentinel(e))

raise

async def \_handle\_events(self) -> None:

while True:

try:

event = await asyncio.wait\_for(

self.\_event\_queue.get(), timeout=EVENT\_INACTIVITY\_TIMEOUT

)

if isinstance(event, WebsocketDoneSentinel):

# processed all events and websocket is done

break

event\_type = event.get("type", "unknown")

if event\_type == "conversation.item.input\_audio\_transcription.completed":

transcript = cast(str, event.get("transcript", ""))

if len(transcript) > 0:

self.\_end\_turn(transcript)

self.\_start\_turn()

await self.\_output\_queue.put(transcript)

await asyncio.sleep(0) # yield control

except asyncio.TimeoutError:

# No new events for a while. Assume the session is done.

break

except Exception as e:

await self.\_output\_queue.put(ErrorSentinel(e))

raise e

await self.\_output\_queue.put(SessionCompleteSentinel())

async def \_stream\_audio(

self, audio\_queue: asyncio.Queue[npt.NDArray[np.int16 | np.float32]]

) -> None:

assert self.\_websocket is not None, "Websocket not initialized"

self.\_start\_turn()

while True:

buffer = await audio\_queue.get()

if buffer is None:

break

self.\_turn\_audio\_buffer.append(buffer)

try:

await self.\_websocket.send(

json.dumps(

{

"type": "input\_audio\_buffer.append",

"audio": base64.b64encode(buffer.tobytes()).decode("utf-8"),

}

)

)

except websockets.ConnectionClosed:

break

except Exception as e:

await self.\_output\_queue.put(ErrorSentinel(e))

raise e

await asyncio.sleep(0) # yield control

async def \_process\_websocket\_connection(self) -> None:

try:

async with websockets.connect(

"wss://api.openai.com/v1/realtime?intent=transcription",

additional\_headers={

"Authorization": f"Bearer {self.\_client.api\_key}",

"OpenAI-Beta": "realtime=v1",

"OpenAI-Log-Session": "1",

},

) as ws:

await self.\_setup\_connection(ws)

self.\_process\_events\_task = asyncio.create\_task(self.\_handle\_events())

self.\_stream\_audio\_task = asyncio.create\_task(self.\_stream\_audio(self.\_input\_queue))

self.connected = True

if self.\_listener\_task:

await self.\_listener\_task

else:

logger.error("Listener task not initialized")

raise AgentsException("Listener task not initialized")

except Exception as e:

await self.\_output\_queue.put(ErrorSentinel(e))

raise e

def \_check\_errors(self) -> None:

if self.\_connection\_task and self.\_connection\_task.done():

exc = self.\_connection\_task.exception()

if exc and isinstance(exc, Exception):

self.\_stored\_exception = exc

if self.\_process\_events\_task and self.\_process\_events\_task.done():

exc = self.\_process\_events\_task.exception()

if exc and isinstance(exc, Exception):

self.\_stored\_exception = exc

if self.\_stream\_audio\_task and self.\_stream\_audio\_task.done():

exc = self.\_stream\_audio\_task.exception()

if exc and isinstance(exc, Exception):

self.\_stored\_exception = exc

if self.\_listener\_task and self.\_listener\_task.done():

exc = self.\_listener\_task.exception()

if exc and isinstance(exc, Exception):

self.\_stored\_exception = exc

def \_cleanup\_tasks(self) -> None:

if self.\_listener\_task and not self.\_listener\_task.done():

self.\_listener\_task.cancel()

if self.\_process\_events\_task and not self.\_process\_events\_task.done():

self.\_process\_events\_task.cancel()

if self.\_stream\_audio\_task and not self.\_stream\_audio\_task.done():

self.\_stream\_audio\_task.cancel()

if self.\_connection\_task and not self.\_connection\_task.done():

self.\_connection\_task.cancel()

async def transcribe\_turns(self) -> AsyncIterator[str]:

self.\_connection\_task = asyncio.create\_task(self.\_process\_websocket\_connection())

while True:

try:

turn = await self.\_output\_queue.get()

except asyncio.CancelledError:

break

if (

turn is None

or isinstance(turn, ErrorSentinel)

or isinstance(turn, SessionCompleteSentinel)

):

self.\_output\_queue.task\_done()

break

yield turn

self.\_output\_queue.task\_done()

if self.\_tracing\_span:

self.\_end\_turn("")

if self.\_websocket:

await self.\_websocket.close()

self.\_check\_errors()

if self.\_stored\_exception:

raise self.\_stored\_exception

async def close(self) -> None:

if self.\_websocket:

await self.\_websocket.close()

self.\_cleanup\_tasks()

OpenAISTTModel

Bases: STTModel

A speech-to-text model for OpenAI.

Source code in src/agents/voice/models/openai\_stt.py

class OpenAISTTModel(STTModel):

"""A speech-to-text model for OpenAI."""

def \_\_init\_\_(

self,

model: str,

openai\_client: AsyncOpenAI,

):

"""Create a new OpenAI speech-to-text model.

Args:

model: The name of the model to use.

openai\_client: The OpenAI client to use.

"""

self.model = model

self.\_client = openai\_client

@property

def model\_name(self) -> str:

return self.model

def \_non\_null\_or\_not\_given(self, value: Any) -> Any:

return value if value is not None else None # NOT\_GIVEN

async def transcribe(

self,

input: AudioInput,

settings: STTModelSettings,

trace\_include\_sensitive\_data: bool,

trace\_include\_sensitive\_audio\_data: bool,

) -> str:

"""Transcribe an audio input.

Args:

input: The audio input to transcribe.

settings: The settings to use for the transcription.

Returns:

The transcribed text.

"""

with transcription\_span(

model=self.model,

input=input.to\_base64() if trace\_include\_sensitive\_audio\_data else "",

input\_format="pcm",

model\_config={

"temperature": self.\_non\_null\_or\_not\_given(settings.temperature),

"language": self.\_non\_null\_or\_not\_given(settings.language),

"prompt": self.\_non\_null\_or\_not\_given(settings.prompt),

},

) as span:

try:

response = await self.\_client.audio.transcriptions.create(

model=self.model,

file=input.to\_audio\_file(),

prompt=self.\_non\_null\_or\_not\_given(settings.prompt),

language=self.\_non\_null\_or\_not\_given(settings.language),

temperature=self.\_non\_null\_or\_not\_given(settings.temperature),

)

if trace\_include\_sensitive\_data:

span.span\_data.output = response.text

return response.text

except Exception as e:

span.span\_data.output = ""

span.set\_error(SpanError(message=str(e), data={}))

raise e

async def create\_session(

self,

input: StreamedAudioInput,

settings: STTModelSettings,

trace\_include\_sensitive\_data: bool,

trace\_include\_sensitive\_audio\_data: bool,

) -> StreamedTranscriptionSession:

"""Create a new transcription session.

Args:

input: The audio input to transcribe.

settings: The settings to use for the transcription.

trace\_include\_sensitive\_data: Whether to include sensitive data in traces.

trace\_include\_sensitive\_audio\_data: Whether to include sensitive audio data in traces.

Returns:

A new transcription session.

"""

return OpenAISTTTranscriptionSession(

input,

self.\_client,

self.model,

settings,

trace\_include\_sensitive\_data,

trace\_include\_sensitive\_audio\_data,

)

\_\_init\_\_

\_\_init\_\_(model: str, openai\_client: AsyncOpenAI)

Create a new OpenAI speech-to-text model.

Parameters:

Name Type Description Default

model str The name of the model to use. required

openai\_client AsyncOpenAI The OpenAI client to use. required

Source code in src/agents/voice/models/openai\_stt.py

def \_\_init\_\_(

self,

model: str,

openai\_client: AsyncOpenAI,

):

"""Create a new OpenAI speech-to-text model.

Args:

model: The name of the model to use.

openai\_client: The OpenAI client to use.

"""

self.model = model

self.\_client = openai\_client

transcribe async

transcribe(

input: AudioInput,

settings: STTModelSettings,

trace\_include\_sensitive\_data: bool,

trace\_include\_sensitive\_audio\_data: bool,

) -> str

Transcribe an audio input.

Parameters:

Name Type Description Default

input AudioInput The audio input to transcribe. required

settings STTModelSettings The settings to use for the transcription. required

Returns:

Type Description

str The transcribed text.

Source code in src/agents/voice/models/openai\_stt.py

async def transcribe(

self,

input: AudioInput,

settings: STTModelSettings,

trace\_include\_sensitive\_data: bool,

trace\_include\_sensitive\_audio\_data: bool,

) -> str:

"""Transcribe an audio input.

Args:

input: The audio input to transcribe.

settings: The settings to use for the transcription.

Returns:

The transcribed text.

"""

with transcription\_span(

model=self.model,

input=input.to\_base64() if trace\_include\_sensitive\_audio\_data else "",

input\_format="pcm",

model\_config={

"temperature": self.\_non\_null\_or\_not\_given(settings.temperature),

"language": self.\_non\_null\_or\_not\_given(settings.language),

"prompt": self.\_non\_null\_or\_not\_given(settings.prompt),

},

) as span:

try:

response = await self.\_client.audio.transcriptions.create(

model=self.model,

file=input.to\_audio\_file(),

prompt=self.\_non\_null\_or\_not\_given(settings.prompt),

language=self.\_non\_null\_or\_not\_given(settings.language),

temperature=self.\_non\_null\_or\_not\_given(settings.temperature),

)

if trace\_include\_sensitive\_data:

span.span\_data.output = response.text

return response.text

except Exception as e:

span.span\_data.output = ""

span.set\_error(SpanError(message=str(e), data={}))

raise e

create\_session async

create\_session(

input: StreamedAudioInput,

settings: STTModelSettings,

trace\_include\_sensitive\_data: bool,

trace\_include\_sensitive\_audio\_data: bool,

) -> StreamedTranscriptionSession

Create a new transcription session.

Parameters:

Name Type Description Default

input StreamedAudioInput The audio input to transcribe. required

settings STTModelSettings The settings to use for the transcription. required

trace\_include\_sensitive\_data bool Whether to include sensitive data in traces. required

trace\_include\_sensitive\_audio\_data bool Whether to include sensitive audio data in traces. required

Returns:

Type Description

StreamedTranscriptionSession A new transcription session.

Source code in src/agents/voice/models/openai\_stt.py

async def create\_session(

self,

input: StreamedAudioInput,

settings: STTModelSettings,

trace\_include\_sensitive\_data: bool,

trace\_include\_sensitive\_audio\_data: bool,

) -> StreamedTranscriptionSession:

"""Create a new transcription session.

Args:

input: The audio input to transcribe.

settings: The settings to use for the transcription.

trace\_include\_sensitive\_data: Whether to include sensitive data in traces.

trace\_include\_sensitive\_audio\_data: Whether to include sensitive audio data in traces.

Returns:

A new transcription session.

"""

return OpenAISTTTranscriptionSession(

input,

self.\_client,

self.model,

settings,

trace\_include\_sensitive\_data,

trace\_include\_sensitive\_audio\_data,

)

OpenAI TTS

OpenAITTSModel

Bases: TTSModel

A text-to-speech model for OpenAI.

Source code in src/agents/voice/models/openai\_tts.py

class OpenAITTSModel(TTSModel):

"""A text-to-speech model for OpenAI."""

def \_\_init\_\_(

self,

model: str,

openai\_client: AsyncOpenAI,

):

"""Create a new OpenAI text-to-speech model.

Args:

model: The name of the model to use.

openai\_client: The OpenAI client to use.

"""

self.model = model

self.\_client = openai\_client

@property

def model\_name(self) -> str:

return self.model

async def run(self, text: str, settings: TTSModelSettings) -> AsyncIterator[bytes]:

"""Run the text-to-speech model.

Args:

text: The text to convert to speech.

settings: The settings to use for the text-to-speech model.

Returns:

An iterator of audio chunks.

"""

response = self.\_client.audio.speech.with\_streaming\_response.create(

model=self.model,

voice=settings.voice or DEFAULT\_VOICE,

input=text,

response\_format="pcm",

extra\_body={

"instructions": settings.instructions,

},

)

async with response as stream:

async for chunk in stream.iter\_bytes(chunk\_size=1024):

yield chunk

\_\_init\_\_

\_\_init\_\_(model: str, openai\_client: AsyncOpenAI)

Create a new OpenAI text-to-speech model.

Parameters:

Name Type Description Default

model str The name of the model to use. required

openai\_client AsyncOpenAI The OpenAI client to use. required

Source code in src/agents/voice/models/openai\_tts.py

def \_\_init\_\_(

self,

model: str,

openai\_client: AsyncOpenAI,

):

"""Create a new OpenAI text-to-speech model.

Args:

model: The name of the model to use.

openai\_client: The OpenAI client to use.

"""

self.model = model

self.\_client = openai\_client

run async

run(

text: str, settings: TTSModelSettings

) -> AsyncIterator[bytes]

Run the text-to-speech model.

Parameters:

Name Type Description Default

text str The text to convert to speech. required

settings TTSModelSettings The settings to use for the text-to-speech model. required

Returns:

Type Description

AsyncIterator[bytes] An iterator of audio chunks.

Source code in src/agents/voice/models/openai\_tts.py

async def run(self, text: str, settings: TTSModelSettings) -> AsyncIterator[bytes]:

"""Run the text-to-speech model.

Args:

text: The text to convert to speech.

settings: The settings to use for the text-to-speech model.

Returns:

An iterator of audio chunks.

"""

response = self.\_client.audio.speech.with\_streaming\_response.create(

model=self.model,

voice=settings.voice or DEFAULT\_VOICE,

input=text,

response\_format="pcm",

extra\_body={

"instructions": settings.instructions,

},

)

async with response as stream:

async for chunk in stream.iter\_bytes(chunk\_size=1024):

yield chunk

Extensions:

Handoff filters

remove\_all\_tools

remove\_all\_tools(

handoff\_input\_data: HandoffInputData,

) -> HandoffInputData

Filters out all tool items: file search, web search and function calls+output.

Source code in src/agents/extensions/handoff\_filters.py

def remove\_all\_tools(handoff\_input\_data: HandoffInputData) -> HandoffInputData:

"""Filters out all tool items: file search, web search and function calls+output."""

history = handoff\_input\_data.input\_history

new\_items = handoff\_input\_data.new\_items

filtered\_history = (

\_remove\_tool\_types\_from\_input(history) if isinstance(history, tuple) else history

)

filtered\_pre\_handoff\_items = \_remove\_tools\_from\_items(handoff\_input\_data.pre\_handoff\_items)

filtered\_new\_items = \_remove\_tools\_from\_items(new\_items)

return HandoffInputData(

input\_history=filtered\_history,

pre\_handoff\_items=filtered\_pre\_handoff\_items,

new\_items=filtered\_new\_items,

)

Handoff prompt

RECOMMENDED\_PROMPT\_PREFIX module-attribute

RECOMMENDED\_PROMPT\_PREFIX = "# System context\nYou are part of a multi-agent system called the Agents SDK, designed to make agent coordination and execution easy. Agents uses two primary abstraction: \*\*Agents\*\* and \*\*Handoffs\*\*. An agent encompasses instructions and tools and can hand off a conversation to another agent when appropriate. Handoffs are achieved by calling a handoff function, generally named `transfer\_to\_<agent\_name>`. Transfers between agents are handled seamlessly in the background; do not mention or draw attention to these transfers in your conversation with the user.\n"

prompt\_with\_handoff\_instructions

prompt\_with\_handoff\_instructions(prompt: str) -> str

Add recommended instructions to the prompt for agents that use handoffs.

Source code in src/agents/extensions/handoff\_prompt.py

def prompt\_with\_handoff\_instructions(prompt: str) -> str:

"""

Add recommended instructions to the prompt for agents that use handoffs.

"""

return f"{RECOMMENDED\_PROMPT\_PREFIX}\n\n{prompt}"