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  ①
async def fetch_weather(location: Location) -> str:
    ②
    """Fetch the weather for a given location.

    Args:
        location: The location to fetch the weather for.
    """
    # In real life, we'd fetch the weather from a weather API
    return "sunny"


@function_tool(name_override="fetch_data")  ③
def read_file(ctx: RunContextWrapper[Any], path: str, directory: str | None = None) -> str:
    """Read the contents of a file.

    Args:
        path: The path to the file to read.
        directory: The directory to read the file from.
    """
    # In real life, we'd read the file from the file system
    return "<file contents>"


agent = Agent(
    name="Assistant",
    tools=[fetch_weather, read_file],  ④
)
```

```python
for tool in agent.tools:
    if isinstance(tool, FunctionTool):
        print(tool.name)
        print(tool.description)
        print(json.dumps(tool.params_json_schema, indent=2))
        print()
```

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

Expand to see output

```
fetch_weather
Fetch the weather for a given location.
{
"$defs": {
  "Location": {
    "properties": {
      "lat": {
        "title": "Lat",
        "type": "number"
      },
      "long": {
        "title": "Long",
        "type": "number"
      }
    },
    "required": [
      "lat",
      "long"
    ],
    "title": "Location",
    "type": "object"
  }
},
"properties": {
  "location": {
    "$ref": "#/$defs/Location",
    "description": "The location to fetch the weather for."
  }
},
"required": [
  "location"
],
"title": "fetch_weather_args",
"type": "object"
}

fetch_data
Read the contents of a file.
{
"properties": {
  "path": {
    "description": "The path to the file to read.",
    "title": "Path",
    "type": "string"
  },
  "directory": {
    "anyOf": [
      {
        "type": "string"
      },
      {
        "type": "null"
      }
    ],
    "default": null,
    "description": "The directory to read the file from.",
    "title": "Directory"
  }
},
"required": [
  "path"
],
"title": "fetch_data_args",
"type": "object"
}
```

Custom function tools

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

```python
from typing import Any

from pydantic import BaseModel

from agents import RunContextWrapper, FunctionTool



def do_some_work(data: str) -> str:
    return "done"


class FunctionArgs(BaseModel):
    username: str
    age: int


async def run_function(ctx: RunContextWrapper[Any], args: str) -> str:
    parsed = FunctionArgs.model_validate_json(args)
    return do_some_work(data=f"{parsed.username} is {parsed.age} years old")
```

```
tool = FunctionTool(
    name="process_user",
    description="Processes extracted user data",
    params_json_schema=FunctionArgs.model_json_schema(),
    on_invoke_tool=run_function,
)
```

Automatic argument and docstring parsing

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

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

The code for the schema extraction lives in `agents.function_schema`.

Agents as tools

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

```
from agents import Agent, Runner
import asyncio

spanish_agent = Agent(
    name="Spanish agent",
    instructions="You translate the user's message to Spanish",
)

french_agent = Agent(
    name="French agent",
    instructions="You translate the user's message to French",
)

orchestrator_agent = Agent(
    name="orchestrator_agent",
    instructions=(
        "You are a translation agent. You use the tools given to you to
translate."
        "If asked for multiple translations, you call the relevant tools."
    ),
    tools=[
        spanish_agent.as_tool(
            tool_name="translate_to_spanish",
            tool_description="Translate the user's message to Spanish",
        ),
        french_agent.as_tool(
            tool_name="translate_to_french",
            tool_description="Translate the user's message to French",
        ),
    ],
)

async def main():
    result = await Runner.run(orchestrator_agent, input="Say 'Hello, how are you?'
in Spanish.")
    print(result.final_output)
```

Customizing tool-agents

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

```
@function_tool
async def run_my_agent() -> str:
  """A tool that runs the agent with custom configs".

    agent = Agent(name="My agent", instructions="...")

    result = await Runner.run(
        agent,
        input="...",
        max_turns=5,
        run_config=...
    )

    return str(result.final_output)
```

Handling errors in function tools

When you create a function tool via `@function_tool`, you can pass a `failure_error_function`. This is a function that provides an error response to the LLM in case the tool call crashes.

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

If you are manually creating a `FunctionTool` object, then you must handle errors inside the `on_invoke_tool` function.