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:

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

Note

To prevent infinite loops, the framework automatically resets `tool_choice` to "auto" after a tool call. This behavior is configurable via `agent.reset_tool_choice`. 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.