Assistants Function Calling

Beta

Copy page

Similar to the Chat Completions API, the Assistants API supports function calling. Function calling allows you to describe functions to the Assistants API and have it intelligently return the functions that need to be called along with their arguments.

Quickstart

In this example, we'll create a weather assistant and define two functions, `get_current_temperature` and `get_rain_probability`, as tools that the Assistant can call. Depending on the user query, the model will invoke parallel function calling if using our latest models released on or after Nov 6, 2023. In our example that uses parallel function calling, we will ask the Assistant what the weather in San Francisco is like today and the chances of rain. We also show how to output the Assistant's response with streaming.

> With the launch of Structured Outputs, you can now use the parameter `strict: true` when using function calling with the Assistants API. For more information, refer to the Function calling guide. Please note that Structured Outputs are not supported in the Assistants API when using vision.

Step 1: Define functions

When creating your assistant, you will first define the functions under the `tools` param of the assistant.

python

```python
from openai import OpenAI
client = OpenAI()

assistant = client.beta.assistants.create(
  instructions="You are a weather bot. Use the provided functions to answer questions.",
  model="gpt-4o",
  tools=[
    {
      "type": "function",
      "function": {
        "name": "get_current_temperature",
        "description": "Get the current temperature for a specific location",
        "parameters": {
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
                "description": "The city and state, e.g., San Francisco, CA"
              },
              "unit": {
                "type": "string",
                "enum": ["Celsius", "Fahrenheit"],
                "description": "The temperature unit to use. Infer this from the user's locati
              }
            },
            "required": ["location", "unit"]
          }
        }
      },
      {
        "type": "function",
        "function": {
          "name": "get_rain_probability",
          "description": "Get the probability of rain for a specific location",
          "parameters": {
            "type": "object",
            "properties": {
              "location": {
                "type": "string",
                "description": "The city and state, e.g., San Francisco, CA"
              }
            },
            "required": ["location"]
          }
        }
      }
    ]
  )
```

Step 2: Create a Thread and add Messages

Create a Thread when a user starts a conversation and add Messages to the Thread as the user asks questions.

python

```python
thread = client.beta.threads.create()
message = client.beta.threads.messages.create(
  thread_id=thread.id,
  role="user",
  content="What's the weather in San Francisco today and the likelihood it'll rain?",
)
```

Step 3: Initiate a Run

When you initiate a Run on a Thread containing a user Message that triggers one or more functions, the Run will enter a `pending` status. After it processes, the run will enter a `requires_action` state which you can verify by checking the Run's `status`. This indicates that you need to run tools and submit their outputs to the Assistant to continue Run execution. In our case, we will see two `tool_calls`, which indicates that the user query resulted in parallel function calling.

Note that a runs expire ten minutes after creation. Be sure to submit your tool outputs before the 10 min mark.

You will see two `tool_calls` within `required_action`, which indicates the user query triggered parallel function calling.

```json
{
  "id": "run_qJL1kI9xxWlfE0z1yfL0fGg9",
  ...
  "status": "requires_action",
  "required_action": {
    "submit_tool_outputs": {
      "tool_calls": [
        {
          "id": "call_FthC9qRpsL5kBpwwyw6c7j4k",
          "function": {
            "arguments": "{"location": "San Francisco, CA"}",
            "name": "get_rain_probability"
          },
          "type": "function"
        },
        {
          "id": "call_RpEDoB8O0FTL9JoKTuCVFOyR",
          "function": {
            "arguments": "{"location": "San Francisco, CA", "unit": "Fahrenheit"}",
            "name": "get_current_temperature"
          },
          "type": "function"
        }
      ]
    },
    ...
    "type": "submit_tool_outputs"
  }
}
```

Run object truncated here for readability

How you initiate a Run and submit `tool_calls` will differ depending on whether you are using streaming or not, although in both cases all `tool_calls` need to be submitted at the same time. You can then complete the Run by submitting the tool outputs from the functions you called. Pass each `tool_call_id` referenced in the `required_action` object to match outputs to each function call.

With streaming Without streaming

For the streaming case, we create an EventHandler class to handle events in the response stream and submit all tool outputs at once with the “submit tool outputs stream” helper in the Python and Node SDKs.

python

```python
from typing_extensions import override
from openai import AssistantEventHandler

class EventHandler(AssistantEventHandler):
    @override
    def on_event(self, event):
      # Retrieve events that are denoted with 'requires_action'
      # since these will have our tool_calls
      if event.event == 'thread.run.requires_action':
        run_id = event.data.id  # Retrieve the run ID from the event data
        self.handle_requires_action(event.data, run_id)

    def handle_requires_action(self, data, run_id):
      tool_outputs = []

      for tool in data.required_action.submit_tool_outputs.tool_calls:
        if tool.function.name == "get_current_temperature":
          tool_outputs.append({"tool_call_id": tool.id, "output": "57"})
        elif tool.function.name == "get_rain_probability":
          tool_outputs.append({"tool_call_id": tool.id, "output": "0.06"})

      # Submit all tool_outputs at the same time
      self.submit_tool_outputs(tool_outputs, run_id)

    def submit_tool_outputs(self, tool_outputs, run_id):
      # Use the submit_tool_outputs_stream helper
      with client.beta.threads.runs.submit_tool_outputs_stream(
        thread_id=self.current_run.thread_id,
        run_id=self.current_run.id,
        tool_outputs=tool_outputs,
        event_handler=EventHandler(),
      ) as stream:
        for text in stream.text_deltas:
          print(text, end="", flush=True)
        print()


with client.beta.threads.runs.stream(
  thread_id=thread.id,
  assistant_id=assistant.id,
  event_handler=EventHandler()
) as stream:
  stream.until_done()
```

Using Structured Outputs

When you enable Structured Outputs by supplying `strict: true`, the OpenAI API will pre-process your supplied schema on your first request, and then use this artifact to constrain the model to your schema.

```python
from openai import OpenAI
client = OpenAI()

assistant = client.beta.assistants.create(
  instructions="You are a weather bot. Use the provided functions to answer questions.",
  model="gpt-4o-2024-08-06",
  tools=[
    {
      "type": "function",
      "function": {
        "name": "get_current_temperature",
        "description": "Get the current temperature for a specific location",
        "parameters": {
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
              "description": "The city and state, e.g., San Francisco, CA"
            },
            "unit": {
              "type": "string",
              "enum": ["Celsius", "Fahrenheit"],
              "description": "The temperature unit to use. Infer this from the user's locati
            }
          },
          "required": ["location", "unit"],
          "additionalProperties": False
        },
        "strict": True
      }
    },
    {
      "type": "function",
      "function": {
        "name": "get_rain_probability",
        "description": "Get the probability of rain for a specific location",
        "parameters": {
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
              "description": "The city and state, e.g., San Francisco, CA"
            }
          },
          "required": ["location"],
          "additionalProperties": False
```

```
            },
            "strict": True
          }
        }
      ]
    )
```