

Function calling

Copy page

Enable models to fetch data and take actions.

Function calling provides a powerful and flexible way for OpenAI models to interface with your code or external services. This guide will explain how to connect the models to your own custom code to fetch data or take action.

- Get weather
- Send email
- Search knowledge base

Responses

Overview

Function calling steps

Defining functions

Handling function calls

Additional configs

Streaming

Function calling example with get_weather functionpython

```
1 from openai import OpenAI
2
3 client = OpenAI()
4
5 tools = [{
6     "type": "function",
7     "name": "get_weather",
8     "description": "Get current temperature for a given location.",
9     "parameters": {
10        "type": "object",
11        "properties": {
12            "location": {
13                "type": "string",
14                "description": "City and country e.g. Bogotá, Colombia"
15            }
16        },
17        "required": [
18            "location"
19        ],
20        "additionalProperties": False
21    }
22 }]
23
24
25 response = client.responses.create(
26     model="gpt-4.1",
27     input=[{"role": "user", "content": "What is the weather like in Paris today?"}],
28     tools=tools
29 )
30
31 print(response.output)
```

Output

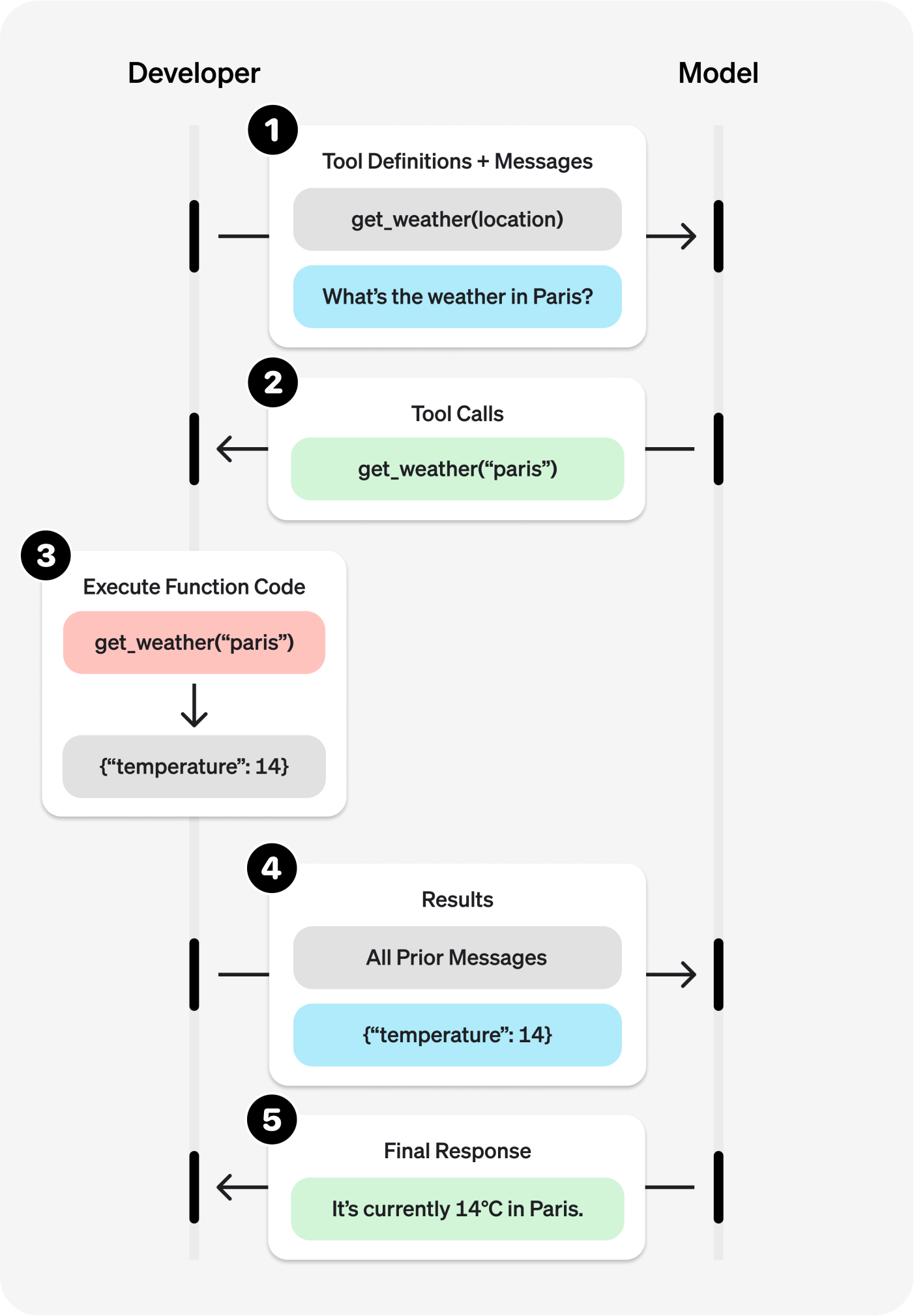
```
1 [{
2     "type": "function_call",
3     "id": "fc_12345xyz",
4     "call_id": "call_12345xyz",
5     "name": "get_weather",
6     "arguments": "{\"location\":\"Paris, France\"}"
7 }]
```

Experiment with function calling and generate function schemas in the Playground!

Overview

You can give the model access to your own custom code through **function calling**. Based on the system prompt and messages, the model may decide to call these functions — **instead of (or in addition to) generating text or audio**.

You'll then execute the function code, send back the results, and the model will incorporate them into its final response.



Function calling has two primary use cases:

Fetching Data	Retrieve up-to-date information to incorporate into the model's response (RAG). Useful for searching knowledge bases and retrieving specific data from APIs (e.g. current weather data).
Taking Action	Perform actions like submitting a form, calling APIs, modifying application state (UI/frontend or backend), or taking agentic workflow actions (like handing off the conversation).

Sample function

Let's look at the steps to allow a model to use a real `get_weather` function defined below:

Sample `get_weather` function implemented in your codebasepython ↕

```
1 import requests
2
```

```
3 def get_weather(latitude, longitude):
4     response = requests.get(f"https://api.open-meteo.com/v1/forecast?latitude={latitud
5     data = response.json()
6     return data['current']['temperature_2m']
```

Unlike the diagram earlier, this function expects precise `latitude` and `longitude` instead of a general `location` parameter. (However, our models can automatically determine the coordinates for many locations!)

Function calling steps

- 1 Call model with **functions defined** – along with your system and user messages.

Step 1: Call model with get_weather tool definedpython ↕

```
1 from openai import OpenAI
2 import json
3
4 client = OpenAI()
5
6 tools = [{
7     "type": "function",
8     "name": "get_weather",
9     "description": "Get current temperature for provided coordinates in celsius.",
10    "parameters": {
11        "type": "object",
12        "properties": {
13            "latitude": {"type": "number"},
14            "longitude": {"type": "number"}
15        },
16        "required": ["latitude", "longitude"],
17        "additionalProperties": False
18    },
19    "strict": True
20 }]
21
22
23 input_messages = [{"role": "user", "content": "What's the weather like in Paris today
24
25 response = client.responses.create(
26     model="gpt-4.1",
27     input=input_messages,
28     tools=tools,
29 )
```

- 2 Model decides to call function(s) – model returns the **name** and **input arguments**.

response.output

```
1 [{
2     "type": "function_call",
3     "id": "fc_12345xyz",
4     "call_id": "call_12345xyz",
5     "name": "get_weather",
6     "arguments": "{\"latitude\":48.8566,\"longitude\":2.3522}"
7 }]
```

- 3 Execute function code – parse the model's response and **handle function calls**.

Step 3: Execute get_weather functionpython ↕

```
1 tool_call = response.output[0]
2 args = json.loads(tool_call.arguments)
3
4 result = get_weather(args["latitude"], args["longitude"])
```

- 4 Supply model with results – so it can incorporate them into its final response.

Step 4: Supply result and call model againpython

```
1 input_messages.append(tool_call) # append model's function call message
2 input_messages.append({          # append result message
3     "type": "function_call_output",
4     "call_id": tool_call.call_id,
5     "output": str(result)
6 })
7
8 response_2 = client.responses.create(
9     model="gpt-4.1",
10    input=input_messages,
11    tools=tools,
12 )
13 print(response_2.output_text)
```

5 **Model responds** – incorporating the result in its output.

response_2.output_text

```
"The current temperature in Paris is 14°C (57.2°F)."
```

Defining functions

Functions can be set in the `tools` parameter of each API request.

A function is defined by its schema, which informs the model what it does and what input arguments it expects. It comprises the following fields:

FIELD	DESCRIPTION
type	This should always be function
name	The function's name (e.g. get_weather)
description	Details on when and how to use the function
parameters	JSON schema defining the function's input arguments
strict	Whether to enforce strict mode for the function call

Take a look at this example or generate your own below (or in our [Playground](#)).

 Generate

Example function schema

```
1 {
2     "type": "function",
3     "function": {
4         "name": "get_weather",
5         "description": "Retrieves current weather for the given location.",
6         "parameters": {
7             "type": "object",
8             "properties": {
9                 "location": {
10                    "type": "string",
11                    "description": "City and country e.g. Bogotá, Colombia"
12                },
13                "units": {
14                    "type": "string",
15                    "enum": [
16                        "celsius",
17                        "fahrenheit"
18                    ],
19                    "description": "Units the temperature will be returned in."
20                }
21            },
22            "required": [
23
```

```
24         "location",
25         "units"
26     ],
27     "additionalProperties": false
28 },
29     "strict": true
30 }
}
```

Because the `parameters` are defined by a [JSON schema](#), you can leverage many of its rich features like property types, enums, descriptions, nested objects, and, recursive objects.

Best practices for defining functions

- 1 **Write clear and detailed function names, parameter descriptions, and instructions.**
 - **Explicitly describe the purpose of the function and each parameter** (and its format), and what the output represents.
 - **Use the system prompt to describe when (and when not) to use each function.** Generally, tell the model *exactly* what to do.
 - **Include examples and edge cases**, especially to rectify any recurring failures. (**Note:** Adding examples may hurt performance for [reasoning models](#).)
- 2 **Apply software engineering best practices.**
 - **Make the functions obvious and intuitive.** ([principle of least surprise](#))
 - **Use enums** and object structure to make invalid states unrepresentable. (e.g. `toggle_light(on: bool, off: bool)` allows for invalid calls)
 - **Pass the intern test.** Can an intern/human correctly use the function given nothing but what you gave the model? (If not, what questions do they ask you? Add the answers to the prompt.)
- 3 **Offload the burden from the model and use code where possible.**
 - **Don't make the model fill arguments you already know.** For example, if you already have an `order_id` based on a previous menu, don't have an `order_id` param – instead, have no params `submit_refund()` and pass the `order_id` with code.
 - **Combine functions that are always called in sequence.** For example, if you always call `mark_location()` after `query_location()`, just move the marking logic into the query function call.
- 4 **Keep the number of functions small for higher accuracy.**
 - **Evaluate your performance** with different numbers of functions.
 - **Aim for fewer than 20 functions** at any one time, though this is just a soft suggestion.
- 5 **Leverage OpenAI resources.**
 - **Generate and iterate on function schemas** in the [Playground](#).
 - **Consider [fine-tuning](#)** to increase function calling accuracy for large numbers of functions or difficult tasks. ([cookbook](#))

Token Usage

Under the hood, functions are injected into the system message in a syntax the model has been trained on. This means functions count against the model's context limit and are billed as input tokens. If you run into token limits, we suggest limiting the number of functions or the length of the descriptions you provide for function parameters.

It is also possible to use [fine-tuning](#) to reduce the number of tokens used if you have many functions defined in your tools specification.

Handling function calls

When the model calls a function, you must execute it and return the result. Since model responses can include zero, one, or multiple calls, it is best practice to assume there are several.

The response `output` array contains an entry with the `type` having a value of `function_call`. Each entry with a `call_id` (used later to submit the function result), `name`, and JSON-encoded `arguments`.

Sample response with multiple function calls

```
1  [
2    {
3      "id": "fc_12345xyz",
4      "call_id": "call_12345xyz",
5      "type": "function_call",
6      "name": "get_weather",
7      "arguments": "{\"location\":\"Paris, France\"}"
8    },
9    {
10     "id": "fc_67890abc",
11     "call_id": "call_67890abc",
12     "type": "function_call",
13     "name": "get_weather",
14     "arguments": "{\"location\":\"Bogotá, Colombia\"}"
15   },
16   {
17     "id": "fc_99999def",
18     "call_id": "call_99999def",
19     "type": "function_call",
20     "name": "send_email",
21     "arguments": "{\"to\":\"bob@email.com\",\"body\":\"Hi bob\"}"
22   }
23 ]
```

Execute function calls and append resultspython

```
1  for tool_call in response.output:
2    if tool_call.type != "function_call":
3      continue
4
5    name = tool_call.name
6    args = json.loads(tool_call.arguments)
7
8    result = call_function(name, args)
9    input_messages.append({
10      "type": "function_call_output",
11      "call_id": tool_call.call_id,
12      "output": str(result)
13    })
```

In the example above, we have a hypothetical `call_function` to route each call. Here’s a possible implementation:

Execute function calls and append resultspython

```
1  def call_function(name, args):
2    if name == "get_weather":
3      return get_weather(**args)
4    if name == "send_email":
5      return send_email(**args)
```

Formatting results

A result must be a string, but the format is up to you (JSON, error codes, plain text, etc.). The model will interpret that string as needed.

If your function has no return value (e.g. `send_email`), simply return a string to indicate success or failure. (e.g. `"success"`)

Incorporating results into response

After appending the results to your `input` , you can send them back to the model to get a final response.

Send results back to modelpython

```
1 response = client.responses.create(  
2     model="gpt-4.1",  
3     input=input_messages,  
4     tools=tools,  
5 )
```

Final response

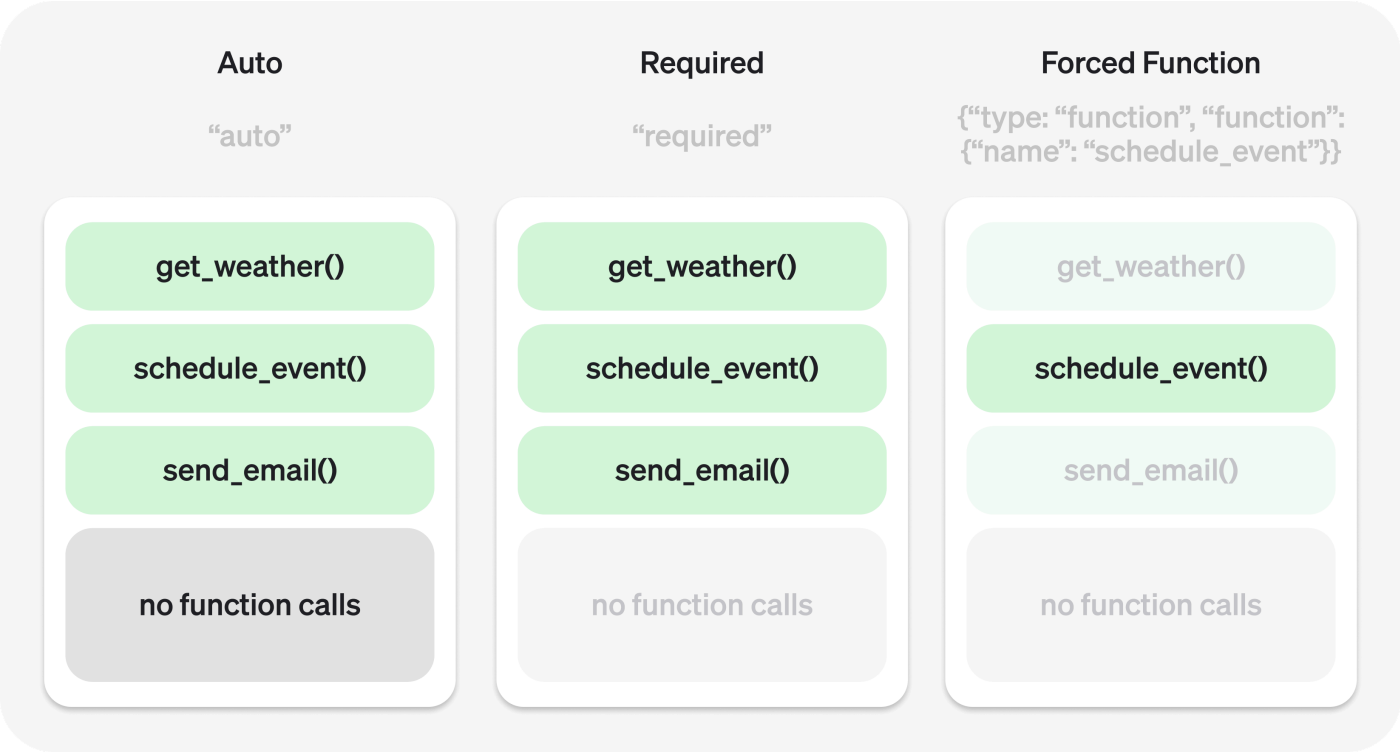
```
"It's about 15°C in Paris, 18°C in Bogotá, and I've sent that email to Bob."
```

Additional configurations

Tool choice

By default the model will determine when and how many tools to use. You can force specific behavior with the `tool_choice` parameter.

- 1 **Auto: (Default)** Call zero, one, or multiple functions. `tool_choice: "auto"`
- 2 **Required:** Call one or more functions. `tool_choice: "required"`
- 1 **Forced Function:** Call exactly one specific function.
`tool_choice: {"type": "function", "name": "get_weather"}`



You can also set `tool_choice` to `"none"` to imitate the behavior of passing no functions.

Parallel function calling

The model may choose to call multiple functions in a single turn. You can prevent this by setting `parallel_tool_calls` to `false` , which ensures exactly zero or one tool is called.

Note: Currently, if the model calls multiple functions in one turn then `strict mode` will be disabled for those calls.

Note for `gpt-4.1-nano-2025-04-14` : This snapshot of `gpt-4.1-nano` can sometimes include multiple tools calls for the same tool if parallel tool calls are enabled. It is recommended to disable this feature when using this nano snapshot.

Strict mode

Setting `strict` to `true` will ensure function calls reliably adhere to the function schema, instead of being best effort. We recommend always enabling strict mode.

Under the hood, strict mode works by leveraging our [structured outputs](#) feature and therefore introduces a couple requirements:

- 1 `additionalProperties` must be set to `false` for each object in the `parameters` .
- 2 All fields in `properties` must be marked as `required` .

You can denote optional fields by adding `null` as a `type` option (see example below).

Strict mode enabled

Strict mode disabled

1 {
2 "type": "function",
3 "name": "get_weather",
4 "description": "Retrieves current weather for the given location.",
5 "strict": true,
6 "parameters": {
7 "type": "object",
8 "properties": {
9 "location": {
10 "type": "string",
11 "description": "City and country e.g. Bogotá, Colombia"
12 },
13 "units": {
14 "type": ["string", "null"],
15 "enum": ["celsius", "fahrenheit"],
16 "description": "Units the temperature will be returned in."
17 }
18 },
19 "required": ["location", "units"],
20 "additionalProperties": false
21 }
22 }

All schemas generated in the [playground](#) have strict mode enabled.

While we recommend you enable strict mode, it has a few limitations:

- 1 Some features of JSON schema are not supported. (See [supported schemas](#).)
- 2 Schemas undergo additional processing on the first request (and are then cached). If your schemas vary from request to request, this may result in higher latencies.
- 3 Schemas are cached for performance, and are not eligible for [zero data retention](#).

Streaming

Streaming can be used to surface progress by showing which function is called as the model fills its arguments, and even displaying the arguments in real time.

Streaming function calls is very similar to streaming regular responses: you set `stream` to `true` and get different `event` objects.

Streaming function callspython

1 from openai import OpenAI
2
3 client = OpenAI()
4
5 tools = [{
6 "type": "function",
7 "name": "get_weather",
8 "description": "Get current temperature for a given location.",
9 "parameters": {
10 "type": "object",
11 "properties": {
12 "location": {
13 "type": "string",


```
14         "description": "City and country e.g. Bogotá, Colombia"
15     },
16 },
17     "required": [
18         "location"
19     ],
20     "additionalProperties": False
21 }
22 }
23
24 stream = client.responses.create(
25     model="gpt-4.1",
26     input=[{"role": "user", "content": "What's the weather like in Paris today?"}],
27     tools=tools,
28     stream=True
29 )
30
31
32 for event in stream:
33     print(event)
```

Output events 

```
1  {"type": "response.output_item.added", "response_id": "resp_1234xyz", "output_index": 0, "i
2  {"type": "response.function_call_arguments.delta", "response_id": "resp_1234xyz", "item_i
3  {"type": "response.function_call_arguments.delta", "response_id": "resp_1234xyz", "item_i
4  {"type": "response.function_call_arguments.delta", "response_id": "resp_1234xyz", "item_i
5  {"type": "response.function_call_arguments.delta", "response_id": "resp_1234xyz", "item_i
6  {"type": "response.function_call_arguments.delta", "response_id": "resp_1234xyz", "item_i
7  {"type": "response.function_call_arguments.delta", "response_id": "resp_1234xyz", "item_i
8  {"type": "response.function_call_arguments.delta", "response_id": "resp_1234xyz", "item_i
9  {"type": "response.function_call_arguments.done", "response_id": "resp_1234xyz", "item_id
10 {"type": "response.output_item.done", "response_id": "resp_1234xyz", "output_index": 0, "it
```

Instead of aggregating chunks into a single `content` string, however, you're aggregating chunks into an encoded `arguments` JSON object.


When the model calls one or more functions an event of type `response.output_item.added` will be emitted for each function call that contains the following fields:

FIELD	DESCRIPTION
response_id	The id of the response that the function call belongs to
output_index	The index of the output item in the response. This represents the individual function calls in the response.
item	The in-progress function call item that includes a name, arguments and id field

Afterwards you will receive a series of events of type `response.function_call_arguments.delta` which will contain the `delta` of the `arguments` field. These events contain the following fields:

FIELD	DESCRIPTION
response_id	The id of the response that the function call belongs to
item_id	The id of the function call item that the delta belongs to
output_index	The index of the output item in the response. This represents the individual function calls in the response.
delta	The delta of the arguments field.

Below is a code snippet demonstrating how to aggregate the `delta` s into a final `tool_call` object.

Accumulating tool_call deltas python 

```
1  final_tool_calls = {}
2
```

```
3 for event in stream:
4     if event.type == 'response.output_item.added':
5         final_tool_calls[event.output_index] = event.item;
6     elif event.type == 'response.function_call_arguments.delta':
7         index = event.output_index
8
9         if final_tool_calls[index]:
10             final_tool_calls[index].arguments += event.delta
```

Accumulated final_tool_calls[0]

```
1 {
2     "type": "function_call",
3     "id": "fc_1234xyz",
4     "call_id": "call_2345abc",
5     "name": "get_weather",
6     "arguments": "{\"location\":\"Paris, France\"}"
7 }
```

When the model has finished calling the functions an event of type `response.function_call_arguments.done` will be emitted. This event contains the entire function call including the following fields:

FIELD	DESCRIPTION
response_id	The id of the response that the function call belongs to
output_index	The index of the output item in the response. This represents the individual function calls in the response.
item	The function call item that includes a name, arguments and id field.