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, and 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).

If you only want the model to produce JSON, see our docs on structured outputs.

Get weather | Send email | Search knowledge base

Function calling example with get_weather function

python

```python
from openai import OpenAI

client = OpenAI()

tools = [{
    "type": "function",
    "function": {
        "name": "get_weather",
        "description": "Get current temperature for a given location.",
        "parameters": {
            "type": "object",
            "properties": {
                "location": {
                    "type": "string",
                    "description": "City and country e.g. Bogotá, Colombia"
                }
            },
            "required": [
                "location"
            ],
            "additionalProperties": False
        },
        "strict": True
    }
```

```
}]

completion = client.chat.completions.create(
    model="gpt-4o",
    messages=[{"role": "user", "content": "What is the weather like in Paris today?"}],
    tools=tools
)

print(completion.choices[0].message.tool_calls)
```

Output

```
[{
    "id": "call_12345xyz",
    "type": "function",
    "function": {
        "name": "get_weather",
        "arguments": "{\"location\":\"Paris, France\"}"
    }
}]
```

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

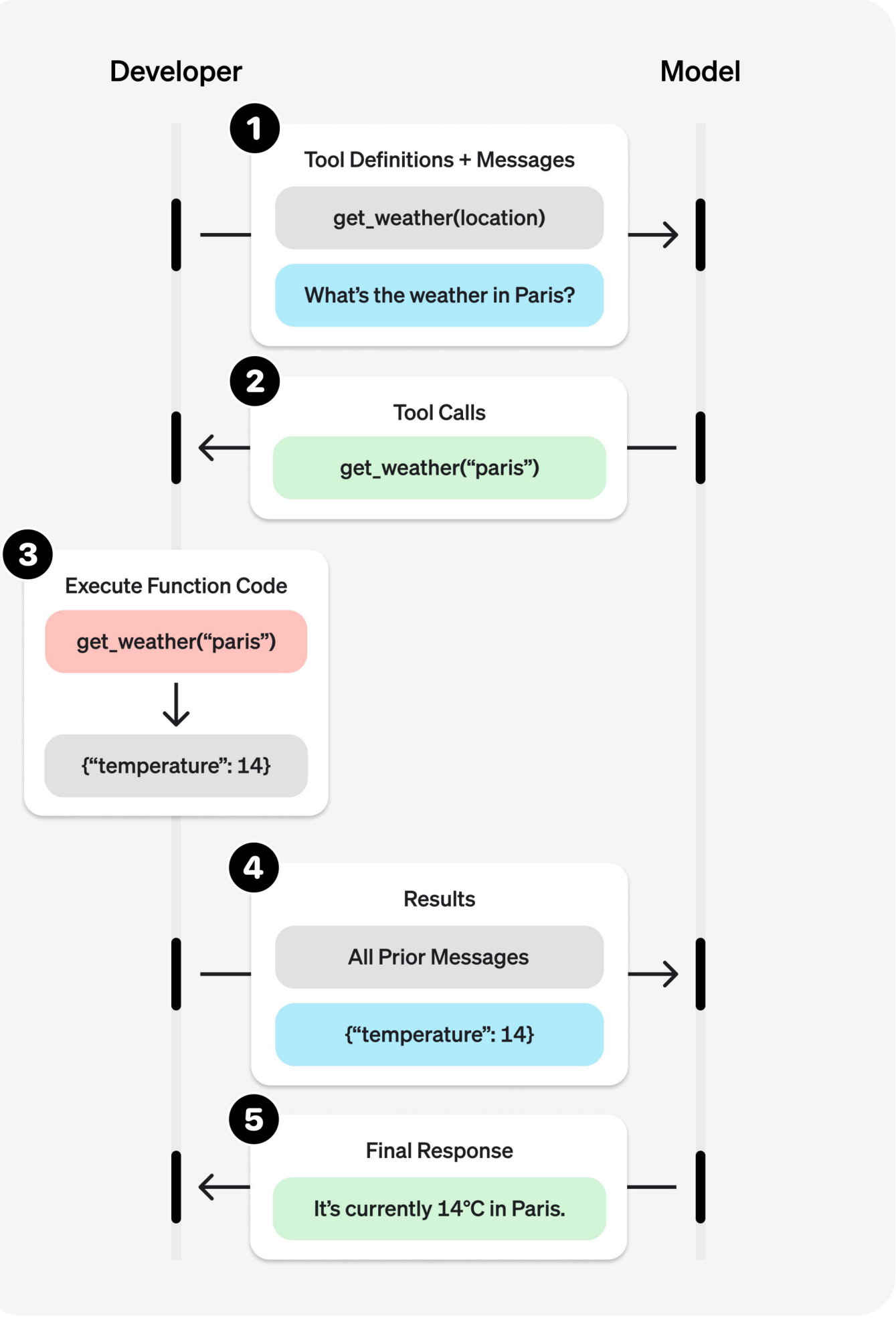
Overview

You can extend the capabilities of OpenAI models by giving them access to `tools`, which can have one of two forms:

Function Calling	Developer-defined code.
Hosted Tools	OpenAI-built tools. (*e.g. file search, code interpreter*) Only available in the Assistants API.

This guide will cover how you can give the model access to your own functions 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.



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 codebase — python

```python
import requests

def get_weather(latitude, longitude):
    response = requests.get(f"https://api.open-meteo.com/v1/forecast?latitude={latitude}&long
    data = response.json()
    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 defined — python

```python
from openai import OpenAI
import json

client = OpenAI()

tools = [{
    "type": "function",
    "function": {
        "name": "get_weather",
        "description": "Get current temperature for provided coordinates in celsius.",
        "parameters": {
            "type": "object",
            "properties": {
                "latitude": {"type": "number"},
                "longitude": {"type": "number"}
            },
            "required": ["latitude", "longitude"],
            "additionalProperties": False
        },
        "strict": True
    }
}]

messages = [{"role": "user", "content": "What's the weather like in Paris today?"}]

completion = client.chat.completions.create(
    model="gpt-4o",
    messages=messages,
    tools=tools,
)
```

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

completion.choices[0].message.tool_calls

```
[{
    "id": "call_12345xyz",
    "type": "function",
    "function": {
      "name": "get_weather",
      "arguments": "{\"latitude\":48.8566,\"longitude\":2.3522}"
    }
}]
```

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

Step 3: Execute get_weather function — python

```python
tool_call = completion.choices[0].message.tool_calls[0]
args = json.loads(tool_call.function.arguments)

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 again — python

```python
messages.append(completion.choices[0].message)  # append model's function call message
messages.append({                               # append result message
    "role": "tool",
    "tool_call_id": tool_call.id,
    "content": str(result)
})

completion_2 = client.chat.completions.create(
    model="gpt-4o",
    messages=messages,
    tools=tools,
)
```

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

completion_2.choices[0].message.content

```
"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 inside a `function` object.

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
`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

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

Generate

Example function schema

```
{
    "type": "function",
    "function": {
        "name": "get_weather",
        "description": "Retrieves current weather for the given location.",
        "parameters": {
            "type": "object",
            "properties": {
                "location": {
                    "type": "string",
                    "description": "City and country e.g. Bogotá, Colombia"
                },
                "units": {
                    "type": "string",
                    "enum": [
                        "celsius",
                        "fahrenheit"
                    ],
                    "description": "Units the temperature will be returned in."
                }
            },
            "required": [
                "location",
                "units"
            ],
            "additionalProperties": false
        },
        "strict": true
    }
}
```

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.

> (Optional) Function calling wth pydantic and zod

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 has an array of `tool_calls`, each with an `id` (used later to submit the function result) and a `function` containing a `name` and JSON-encoded `arguments`.

Sample response with multiple function calls

```
[
    {
        "id": "call_12345xyz",
        "type": "function",
        "function": {
            "name": "get_weather",
            "arguments": "{\"location\":\"Paris, France\"}"
        }
    },
    {
        "id": "call_67890abc",
        "type": "function",
        "function": {
            "name": "get_weather",
            "arguments": "{\"location\":\"Bogotá, Colombia\"}"
        }
    },
    {
        "id": "call_99999def",
        "type": "function",
        "function": {
            "name": "send_email",
            "arguments": "{\"to\":\"bob@email.com\",\"body\":\"Hi bob\"}"
        }
    }
]
```

Execute function calls and append results

python

```python
for tool_call in completion.choices[0].message.tool_calls:
    name = tool_call.function.name
    args = json.loads(tool_call.function.arguments)

    result = call_function(name, args)
    messages.append({
        "role": "tool",
        "tool_call_id": tool_call.id,
        "content": result
    })
```

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

Execute function calls and append results

```python
def call_function(name, args):
    if name == "get_weather":
        return get_weather(**args)
    if name == "send_email":
        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 `messages`, you can send them back to the model to get a final response.

Send results back to model

```python
completion = client.chat.completions.create(
    model="gpt-4o",
    messages=messages,
    tools=tools,
)
```

Final response

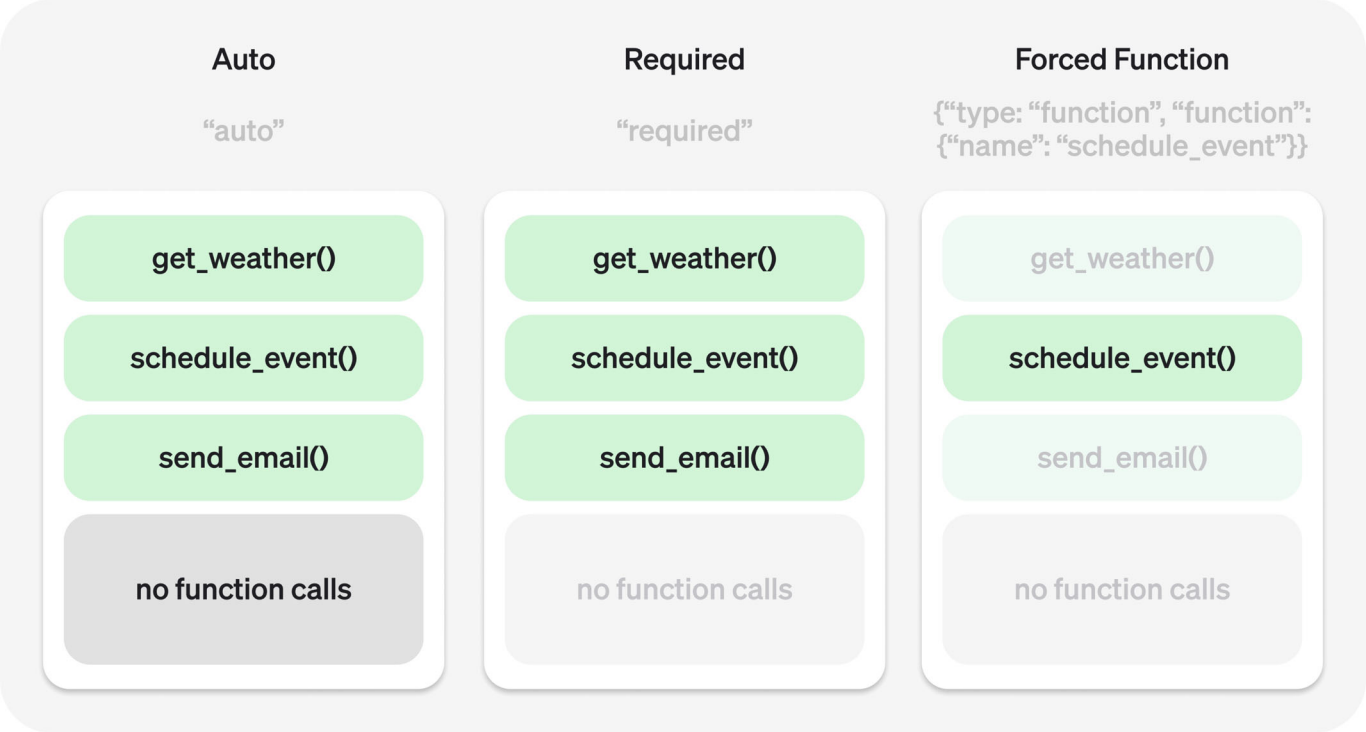
```
"It's about 15°C in Paris, 18°C in Bogotá, and I've sent that email to Bob."
```

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"`
3. **Forced Function:** Call exactly one specific function.
   `tool_choice: {"type": "function", "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.

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

```
{
    "type": "function",
    "function": {
        "name": "get_weather",

```

```
        "description": "Retrieves current weather for the given location.",
        "strict": true,
        "parameters": {
            "type": "object",
            "properties": {
                "location": {
                    "type": "string",
                    "description": "City and country e.g. Bogotá, Colombia"
                },
                "units": {
                    "type": ["string", "null"],
                    "enum": ["celsius", "fahrenheit"],
                    "description": "Units the temperature will be returned in."
                }
            },
            "required": ["location", "units"],
            "additionalProperties": false
        }
    }
}
```

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 chunks with `delta` objects.

Streaming function calls — python

```python
from openai import OpenAI

client = OpenAI()

tools = [{
    "type": "function",
    "function": {
        "name": "get_weather",
        "description": "Get current temperature for a given location.",
        "parameters": {
            "type": "object",
```

```
            "properties": {
                "location": {
                    "type": "string",
                    "description": "City and country e.g. Bogotá, Colombia"
                }
            },
            "required": ["location"],
            "additionalProperties": False
        },
        "strict": True
    }
}]

stream = client.chat.completions.create(
    model="gpt-4o",
    messages=[{"role": "user", "content": "What's the weather like in Paris today?"}],
    tools=tools,
    stream=True
)

for chunk in stream:
    delta = chunk.choices[0].delta
    print(delta.tool_calls)
```

Output delta.tool_calls


```
[{"index": 0, "id": "call_DdmO9pD3xa9XTPNJ32zg2hcA", "function": {"arguments": "", "name": "g
[{"index": 0, "id": null, "function": {"arguments": "{\"", "name": null}, "type": null}]
[{"index": 0, "id": null, "function": {"arguments": "location", "name": null}, "type": null}]
[{"index": 0, "id": null, "function": {"arguments": "\":\"", "name": null}, "type": null}]
[{"index": 0, "id": null, "function": {"arguments": "Paris", "name": null}, "type": null}]
[{"index": 0, "id": null, "function": {"arguments": ",", "name": null}, "type": null}]
[{"index": 0, "id": null, "function": {"arguments": " France", "name": null}, "type": null}]
[{"index": 0, "id": null, "function": {"arguments": "\"}", "name": null}, "type": null}]
null
```


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 the `tool_calls` field of each `delta` will be populated. Each `tool_call` contains the following fields:

FIELD	DESCRIPTION
`index`	Identifies which function call the `delta` is for
`id`	Tool call id.
`function`	Function call delta (`name` and `arguments`)
`type`	Type of `tool_call` (always `function` for function calls)

Many of these fields are only set for the first `delta` of each tool call, like `id`, `function.name`, and `type`.

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

Accumulating tool_call deltas

python

```python
final_tool_calls = {}

for chunk in stream:
    for tool_call in chunk.choices[0].delta.tool_calls or []:
        index = tool_call.index

        if index not in final_tool_calls:
            final_tool_calls[index] = tool_call

        final_tool_calls[index].function.arguments += tool_call.function.arguments
```

Accumulated final_tool_calls[0]

```
{
    "index": 0,
    "id": "call_RzfkBpJgzeR0S242qfvjadNe",
    "function": {
        "name": "get_weather",
        "arguments": "{\"location\":\"Paris, France\"}"
    }
}
```