Function calling

Learn how to connect large language models to external tools.

Introduction

Function calling allows you to connect models like `gpt-4o` to external tools and systems. This is useful for many things such as empowering AI assistants with capabilities, or building deep integrations between your applications and the models.

In August 2024, we launched Structured Outputs. When you turn it on by setting `strict: true`, in your function definition, Structured Outputs ensures that the arguments generated by the model for a function call exactly match the JSON Schema you provided in the function definition.

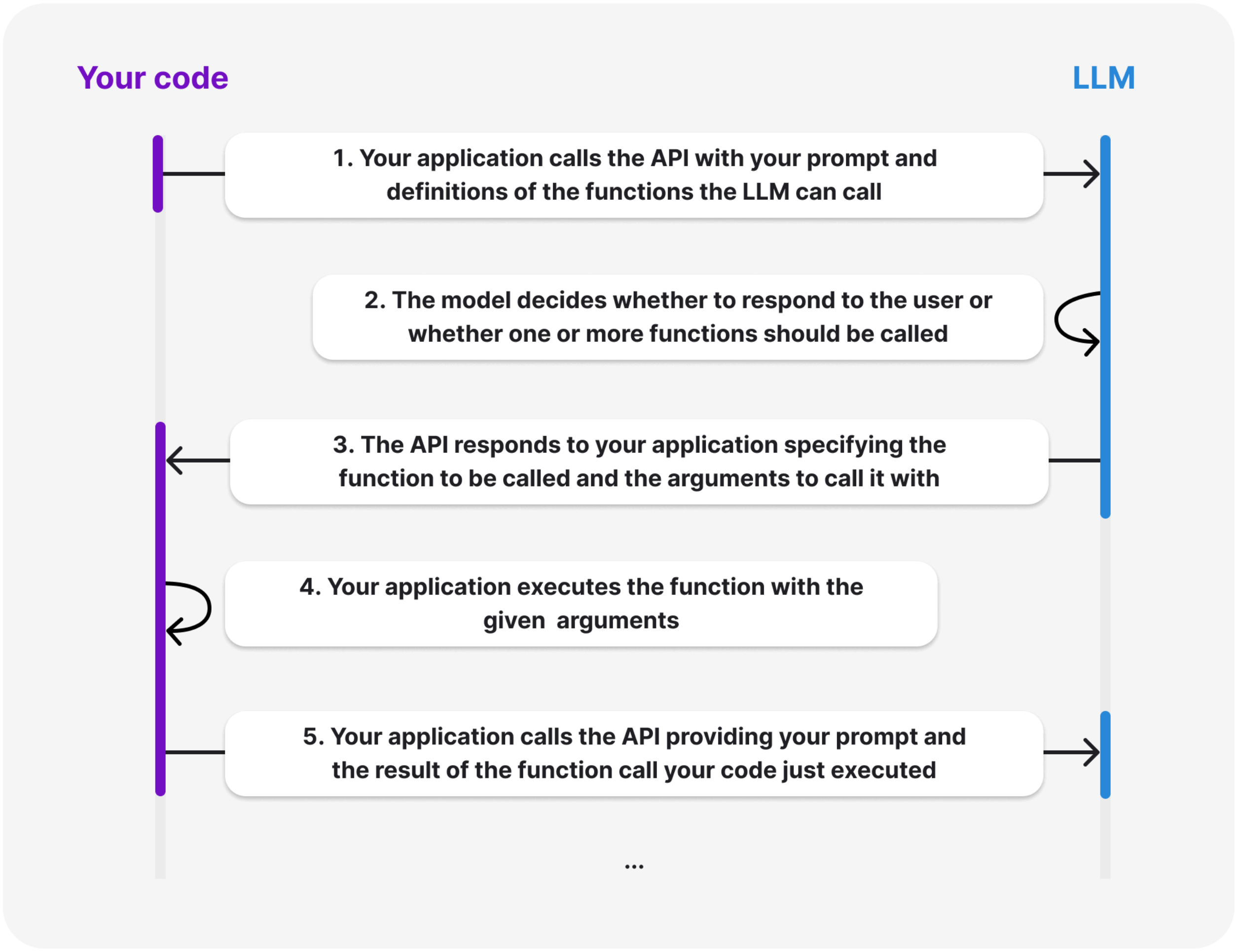
As an alternative to function calling you can instead constrain the model's regular output to match a JSON Schema of your choosing. Learn more about when to use function calling vs when to control the model's normal output by using `response_format`.

Example use cases

Function calling is useful for a large number of use cases, such as:

Enabling assistants to fetch data: an AI assistant needs to fetch the latest customer data from an internal system when a user asks “what are my recent orders?” before it can generate the response to the user
Enabling assistants to take actions: an AI assistant needs to schedule meetings based on user preferences and calendar availability.
Enabling assistants to perform computation: a math tutor assistant needs to perform a math computation.
Building rich workflows: a data extraction pipeline that fetches raw text, then converts it to structured data and saves it in a database.
Modifying your applications' UI: you can use function calls that update the UI based on user input, for example, rendering a pin on a map.

The lifecycle of a function call


Your code
LLM
1. Your application calls the API with your prompt and definitions of the functions the LLM can call
2. The model decides whether to respond to the user or whether one or more functions should be called
3. The API responds to your application specifying the function to be called and the arguments to call it with
4. Your application executes the function with the given arguments
5. Your application calls the API providing your prompt and the result of the function call your code just executed
...

When you use the OpenAI API with function calling, the model never actually executes functions itself, instead in step 3 the model simply generates parameters that can be used to call your function, which your code can then choose how to handle, likely by calling the indicated function. Your application is always in full control.

How to use function calling

Function calling is supported in both the Chat Completions API, Assistants API, and the Batch API. This guide focuses on function calling using the Chat Completions API. We have a separate guide for function calling using the Assistants API.

For the following example, we are building a conversational assistant which is able to help users with their delivery orders. Rather than requiring your users to interact with a typical form, your user can chat with an AI-powered assistant. In order to make this assistant helpful, we want to give it the ability to look up orders and reply with real data about the user's orders.

Step 1: Pick a function in your codebase that the model should be able to call

The starting point for function calling is choosing a function in your own codebase that you'd like to enable the model to generate arguments for.

For this example, let's imagine you want to allow the model to call the `get_delivery_date` function in your codebase which accepts an `order_id` and queries your database to determine the delivery date for a given package. Your function might look like something like the following.

Python

```python
# This is the function that we want the model to be able to call
def get_delivery_date(order_id: str) -> datetime:
    # Connect to the database
    conn = sqlite3.connect('ecommerce.db')
    cursor = conn.cursor()

    # ...
```

NodeJS - Javascript

```javascript
const getDeliveryDate = async (orderId: string): datetime => {
    const connection = await createConnection({
        host: 'localhost',
        user: 'root',
        // ...
    });
}
```

Step 2: Describe your function to the model so it knows how to call it

Now we know what function we wish to allow the model to call, we will create a “function definition” that describes the function to the model. This definition describes both what the function does (and potentially when it should be called) and what parameters are required to call the function.

The `parameters` section of your function definition should be described using JSON Schema. If and when the model generates a function call, it will use this information to generate arguments according to your provided schema.

In this example it may look like this:

```
{
   "name": "get_delivery_date",
   "description": "Get the delivery date for a customer's order. Call this whenever you need to know the delivery date, for example when a customer asks 'Where is my package'",
   "parameters": {
      "type": "object",
      "properties": {
         "order_id": {
            "type": "string",
            "description": "The customer's order ID.",
         },
      },
      "required": ["order_id"],
      "additionalProperties": false,
   }

}
```

Step 3: Pass your function definitions as available “tools” to the model, along with the messages

Next we need to provide our function definitions within an array of available “tools” when calling the Chat Completions API.

As always, we will provide an array of “messages”, which could for example contain your prompt or a whole back and forth conversation between the user and an assistant.

This example shows how you may call the Chat Completions API providing relevant functions and messages for an assistant that handles customer inquiries for a store.

Python

```
tools = [
    {
        "type": "function",
        "function": {
            "name": "get_delivery_date",
            "description": "Get the delivery date for a customer's order. Call this whenever you need to
know the delivery date, for example when a customer asks 'Where is my package'",
            "parameters": {
                "type": "object",
                "properties": {
                    "order_id": {
                        "type": "string",
                        "description": "The customer's order ID.",
                    },
                },
                "required": ["order_id"],
                "additionalProperties": False,
            },
        }
    }
]

messages = [
    {"role": "system", "content": "You are a helpful customer support assistant. Use the supplied
tools to assist the user."},
    {"role": "user", "content": "Hi, can you tell me the delivery date for my order?"}
]

response = openai.chat.completions.create(
    model="gpt-4o",
    messages=messages,
    tools=tools,

)
```

NodeJS - Javascript

```
const tools = [
  {
    type: "function",
    function: {
      name: "get_delivery_date",
      description: "Get the delivery date for a customer's order. Call this whenever you
need to know the delivery date, for example when a customer asks 'Where is my
package'",
      parameters: {
        type: "object",
        properties: {
          order_id: {
            type: "string",
            description: "The customer's order ID.",
          },
        },
        required: ["order_id"],
        additionalProperties: false,
      },
    }
  }
];

const messages = [
  { role: "system", content: "You are a helpful customer support assistant. Use the
supplied tools to assist the user." },
  { role: "user", content: "Hi, can you tell me the delivery date for my order?" }
];

const response = await openai.chat.completions.create({
  model: "gpt-4o",
  messages: messages,
  tools: tools,
});
```

Step 4: Receive and handle the model response

If the model decides that no function should be called

If the model does not generate a function call, then the response will contain a direct reply to the user in the normal way that Chat Completions does.

For example, in this case `chat_response.choices[0].message` may contain:

Python

```
chat.completionsMessage(content='Hi there! I can help with that. Can you please provide your order ID?', role='assistant', function_call=None, tool_calls=None)
```

Javascript

```
{
  role: 'assistant',
  content: "I'd be happy to help with that. Could you please provide me with your order ID?",
}
```

In an assistant use case you will typically want to show this response to the user and let them respond to it, in which case you will call the API again (with both the latest responses from the assistant and user appended to the `messages`).

Let's assume our user responded with their order id, and we sent the following request to the API.

Python

```
tools = [
    {
        "type": "function",
        "function": {
            "name": "get_delivery_date",
            "description": "Get the delivery date for a customer's order. Call this whenever you need to know the delivery date, for example when a customer asks 'Where is my package'",
            "parameters": {
```

```
            "type": "object",
            "properties": {
                "order_id": {
                    "type": "string",
                    "description": "The customer's order ID."
                }
            },
            "required": ["order_id"],
            "additionalProperties": False
        }
    }
  }
]

messages = []
messages.append({"role": "system", "content": "You are a helpful customer support assistant. Use the
supplied tools to assist the user."})
messages.append({"role": "user", "content": "Hi, can you tell me the delivery date for my order?"})




messages.append({"role": "assistant", "content": "Hi there! I can help with that. Can you please provide
your order ID?"})




messages.append({"role": "user", "content": "i think it is order_12345"})






response = client.chat.completions.create(
  model='gpt-4o',
  messages=messages,
  tools=tools

)
```

Javascript - Node

```
const tools = [
  {
    type: "function",
        function: {
            name: "get_delivery_date",
            description: "Get the delivery date for a customer's order. Call this whenever you
need to know the delivery date, for example when a customer asks 'Where is my
package'",
            parameters: {
                type: "object",
                properties: {
                    order_id: {
                        type: "string",
                        description: "The customer's order ID."
                    }
                },
                required: ["order_id"],
                additionalProperties: false
            }
        }
    }
];

const messages = [];
messages.push({ role: "system", content: "You are a helpful customer support assistant.
Use the supplied tools to assist the user." });
messages.push({ role: "user", content: "Hi, can you tell me the delivery date for my
order?" });
// highlight-start
messages.push({ role: "assistant", content: "Hi there! I can help with that. Can you
please provide your order ID?" });
messages.push({ role: "user", content: "i think it is order_12345" });
// highlight-end

const response = await client.chat.completions.create({
    model: 'gpt-4o',
    messages: messages,
    tools: tools
});
```

If the model generated a function call

If the model generated a function call, it will generate the arguments for the call (based on the `parameters` definition you provided).

Here is an example response showing this:

Python

```
Choice(
    finish_reason='tool_calls',
    index=0,
    logprobs=None,
    message=chat.completionsMessage(
        content=None,
        role='assistant',
        function_call=None,
        tool_calls=[
            chat.completionsMessageToolCall(
                id='call_62136354',
                function=Function(
                    arguments='{"order_id":"order_12345"}',
                    name='get_delivery_date'),
                type='function')
        ])
)
```

Javascript - NodeJS

```
{
    finish_reason: 'tool_calls',
    index: 0,
    logprobs: null,
    message: {
        content: null,
        role: 'assistant',
        function_call: null,
        tool_calls: [
            {
                id: 'call_62136354',
                function: {
                    arguments: '{"order_id":"order_12345"}',
                    name: 'get_delivery_date'
                },
                type: 'function'
            }
        ]
    }
}
```

Handling the model response indicating that a function should be called

Assuming the response indicates that a function should be called, your code will now handle this:

Python

```
# Extract the arguments for get_delivery_date
# Note this code assumes we have already determined that the model generated a function call.
See below for a more production ready example that shows how to check if the model generated
a function call
tool_call = response.choices[0].message.tool_calls[0]
arguments = json.loads(tool_call['function']['arguments'])

order_id = arguments.get('order_id')

# Call the get_delivery_date function with the extracted order_id


delivery_date = get_delivery_date(order_id)
```

NodeJS - Javascript

```
// Extract the arguments for get_delivery_date
// Note this code assumes we have already determined that the model generated a
function call. See below for a more production ready example that shows how to check if
the model generated a function call
const toolCall = response.choices[0].message.tool_calls[0];
const arguments = JSON.parse(toolCall.function.arguments);

const order_id = arguments.order_id;

// Call the get_delivery_date function with the extracted order_id
const delivery_date = get_delivery_date(order_id);
```

Step 5: Provide the function call result back to the model

Now we have executed the function call locally, we need to provide the result of this function call back to the Chat Completions API so the model can generate the actual response that the user should see:

Python

```
# Simulate the order_id and delivery_date
order_id = "order_12345"
delivery_date = datetime.now()

# Simulate the tool call response
response = {
   "choices": [
      {
         "message": {
            "tool_calls": [
               {"id": "tool_call_1"}
            ]
         }
      }
   ]
}

# Create a message containing the result of the function call
function_call_result_message = {
   "role": "tool",
   "content": json.dumps({
      "order_id": order_id,
      "delivery_date": delivery_date.strftime('%Y-%m-%d %H:%M:%S')
   }),
   "tool_call_id": response['choices'][0]['message']['tool_calls'][0]['id']
}

# Prepare the chat completion call payload
completion_payload = {
   "model": "gpt-4o",
   "messages": [
      {"role": "system", "content": "You are a helpful customer support assistant. Use the supplied
tools to assist the user."},
      {"role": "user", "content": "Hi, can you tell me the delivery date for my order?"},
      {"role": "assistant", "content": "Hi there! I can help with that. Can you please provide your
order ID?"},
      {"role": "user", "content": "i think it is order_12345"},
      response['choices'][0]['message'],
        function_call_result_message
    ]
}

# Call the OpenAI API's chat completions endpoint to send the tool call result back to the model
response = openai.chat.completions.create(
    model=completion_payload["model"],
    messages=completion_payload["messages"]
)

# Print the response from the API. In this case it will typically contain a message such as "The
delivery date for your order #12345 is xyz. Is there anything else I can help you with?"

print(response)
```

Javascript

```
// Simulate the order_id and delivery_date
const order_id = "order_12345";
const delivery_date = moment();

// Simulate the tool call response
const response = {
    choices: [
        {
            message: {
                tool_calls: [
                    { id: "tool_call_1" }
                ]
            }
        }
    ]
};

// Create a message containing the result of the function call
const function_call_result_message = {
    role: "tool",
    content: JSON.stringify({
        order_id: order_id,
        delivery_date: delivery_date.format('YYYY-MM-DD HH:mm:ss')
    }),
    tool_call_id: response.choices[0].message.tool_calls[0].id
};
```

```
// Prepare the chat completion call payload
const completion_payload = {
   model: "gpt-4o",
   messages: [
      { role: "system", content: "You are a helpful customer support assistant. Use the
supplied tools to assist the user." },
      { role: "user", content: "Hi, can you tell me the delivery date for my order?" },
      { role: "assistant", content: "Hi there! I can help with that. Can you please provide
your order ID?" },
      { role: "user", content: "i think it is order_12345" },
      response.choices[0].message,
      function_call_result_message
   ]
};

// Call the OpenAI API's chat completions endpoint to send the tool call result back to the
model
const final_response = await openai.chat.completions.create({
   model: completion_payload.model,
   messages: completion_payload.messages
});

// Print the response from the API. In this case it will typically contain a message such as
"The delivery date for your order #12345 is xyz. Is there anything else I can help you
with?"
console.log(final_response);
```

That’s all you need to give `gpt-4o` access to your functions.

Handling edge cases

We recommend using the SDK to handle the edge cases described below. If for any reason you cannot use the SDK, you should handle these cases in your code.
When you receive a response from the API, if you're not using the SDK, there are a number of edge cases that production code should handle.

In general, the API will return a valid function call, but there are some edge cases when this won’t happen, such as when you have specified `max_tokens` and the model’s response is cut off as a result.

This sample explains them:

Python

```
# Check if the conversation was too long for the context window
if response['choices'][0]['message']['finish_reason'] == "length":
    print("Error: The conversation was too long for the context window.")
    # Handle the error as needed, e.g., by truncating the conversation or asking for clarification
    handle_length_error(response)

# Check if the model's output included copyright material (or similar)
if response['choices'][0]['message']['finish_reason'] == "content_filter":
    print("Error: The content was filtered due to policy violations.")
    # Handle the error as needed, e.g., by modifying the request or notifying the user
    handle_content_filter_error(response)

# Check if the model has made a tool_call. This is the case either if the "finish_reason" is
"tool_calls" or if the "finish_reason" is "stop" and our API request had forced a function call
if (response['choices'][0]['message']['finish_reason'] == "tool_calls" or
    # This handles the edge case where if we forced the model to call one of our functions, the
finish_reason will actually be "stop" instead of "tool_calls"
    (our_api_request_forced_a_tool_call and response['choices'][0]['message']['finish_reason'] ==
"stop")):
    # Handle tool call
    print("Model made a tool call.")
    # Your code to handle tool calls
    handle_tool_call(response)

# Else finish_reason is "stop", in which case the model was just responding directly to the user
elif response['choices'][0]['message']['finish_reason'] == "stop":
    # Handle the normal stop case
    print("Model responded directly to the user.")
    # Your code to handle normal responses
    handle_normal_response(response)

# Catch any other case, this is unexpected
else:
    print("Unexpected finish_reason:", response['choices'][0]['message']['finish_reason'])
    # Handle unexpected cases as needed


    handle_unexpected_case(response)
```

Javascript - Node

```
// Check if the conversation was too long for the context window
if (response.choices[0].message.finish_reason === "length") {
    console.log("Error: The conversation was too long for the context window.");
    // Handle the error as needed, e.g., by truncating the conversation or asking for
clarification
    handleLengthError(response);
}

// Check if the model's output included copyright material (or similar)
if (response.choices[0].message.finish_reason === "content_filter") {
    console.log("Error: The content was filtered due to policy violations.");
    // Handle the error as needed, e.g., by modifying the request or notifying the user
    handleContentFilterError(response);
}

// Check if the model has made a tool_call. This is the case either if the "finish_reason" is
"tool_calls" or if the "finish_reason" is "stop" and our API request had forced a function
call
if (response.choices[0].message.finish_reason === "tool_calls" ||
    (ourApiRequestForcedAToolCall && response.choices[0].message.finish_reason ===
"stop")) {
    // Handle tool call
    console.log("Model made a tool call.");
    // Your code to handle tool calls
    handleToolCall(response);
}

// Else finish_reason is "stop", in which case the model was just responding directly to
the user
else if (response.choices[0].message.finish_reason === "stop") {
    // Handle the normal stop case
    console.log("Model responded directly to the user.");
    // Your code to handle normal responses
    handleNormalResponse(response);
}

// Catch any other case, this is unexpected
else {
    console.log("Unexpected finish_reason:",
response.choices[0].message.finish_reason);
    // Handle unexpected cases as needed
    handleUnexpectedCase(response);
}
```

Function calling with Structured Outputs

By default, when you use function calling, the API will offer best-effort matching for your parameters, which means that occasionally the model may miss parameters or get their types wrong when using complicated schemas.

Structured Outputs is a feature that ensures model outputs for function calls will exactly match your supplied schema.

Structured Outputs for function calling can be enabled with a single parameter, just by supplying `strict: true`.

Curl

```
curl https://api.openai.com/v1/chat/completions/create \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "model": "gpt-4o-2024-08-06",
    "messages": [
      {
        "role": "system",
        "content": "You are a helpful customer support assistant. Use the supplied tools to assist the
user."
      },
      {
        "role": "user",
        "content": "Hi, can you tell me the delivery date for my order #12345?"
      }
    ],
    "tools": [
      {
        "type": "function",
        "function": {
          "name": "get_delivery_date",
          "description": "Get the delivery date for a customer's order. Call this whenever you need to
know the delivery date, for example when a customer asks 'Where is my package'",
          "parameters": {
            "type": "object",
            "properties": {
              "order_id": {
                "type": "string",
                "description": "The customer's order ID."
              }
          },
          "required": ["order_id"],
          "additionalProperties": false
        }
      },
      "strict": true
    }
  ]

}'
```

Python

```
from enum import Enum
from typing import Union
from pydantic import BaseModel
import openai
from openai import OpenAI

client = OpenAI()

class GetDeliveryDate(BaseModel):
    order_id: str

tools = [openai.pydantic_function_tool(GetDeliveryDate)]

messages = []
messages.append({"role": "system", "content": "You are a helpful customer support
assistant. Use the supplied tools to assist the user."})
messages.append({"role": "user", "content": "Hi, can you tell me the delivery date for my
order #12345?"})

response = client.chat.completions.create(
    model='gpt-4o-2024-08-06',
    messages=messages,
    tools=tools
)

print(response.choices[0].message.tool_calls[0].function)
```

NodeJS - Javascript

```
import OpenAI from "openai";
import { z } from "zod";
import { zodFunction } from "openai/helpers/zod";

const OrderParameters = z.object({
  order_id: z.string().describe("The customer's order ID."),
});

const tools = [
  zodFunction({ name: "getDeliveryDate", parameters: OrderParameters }),
];

const messages = [
  {
    role: "system",
    content:
      "You are a helpful customer support assistant. Use the supplied tools to assist the
user.",
  },
  {
    role: "user",
    content: "Hi, can you tell me the delivery date for my order #12345?",
  },
];

const openai = new OpenAI();

const response = await openai.chat.completions.create({
  model: "gpt-4o-2024-08-06",
  messages: messages,
  tools: tools,
});

console.log(response.choices[0].message.tool_calls?.[0].function);
```

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.

As a result, the model will always follow your exact schema, except in a few circumstances:

- When the model's response is cut off (either due to `max_tokens`, `stop tokens`, or maximum context length)
- When a model refusal happens
- When there is a `content_filter` finish reason

Note that the first time you send a request with a new schema using Structured Outputs, there will be additional latency as the schema is processed, but subsequent requests should incur no overhead.

Supported schemas

Function calling with Structured Outputs supports a subset of the JSON Schema language.

For more information on supported schemas, see the Structured Outputs guide.

Customizing function calling behavior

Function calling supports a number of advanced features such as ability to force function calls, parallel function calling and more.

Configuring parallel function calling

Any models released on or after Nov 6, 2023 may by default generate multiple function calls in a single response, indicating that they should be called in parallel.

This is especially useful if executing the given functions takes a long time. For example, the model may call functions to get the weather in 3 different locations at the same time, which will result in a message with 3 function calls in the tool_calls array.

Example response:

Python

```
response = Choice(
    finish_reason='tool_calls',
    index=0,
    logprobs=None,
    message=chat.completionsMessage(
        content=None,
        role='assistant',
        function_call=None,
        tool_calls=[
            chat.completionsMessageToolCall(
                id='call_62136355',
                function=Function(
                    arguments='{"city":"New York"}',
                    name='check_weather'),
                type='function'),
            chat.completionsMessageToolCall(
                id='call_62136356',
                function=Function(
                    arguments='{"city":"London"}',
                    name='check_weather'),
                type='function'),
            chat.completionsMessageToolCall(
                id='call_62136357',
                function=Function(
                    arguments='{"city":"Tokyo"}',
                    name='check_weather'),
                type='function')
        ])
)

# Iterate through tool calls to handle each weather check
for tool_call in response.message.tool_calls:
    arguments = json.loads(tool_call.function.arguments)
    city = arguments['city']
    weather_info = check_weather(city)

    print(f"Weather in {city}: {weather_info}")
```

Javascript - NodeJS

```
const response = {
    finish_reason: 'tool_calls',
    index: 0,
    logprobs: null,
    message: {
        content: null,
        role: 'assistant',
        function_call: null,
          tool_calls: [
            {
              id: 'call_62136355',
              function: {
                arguments: '{"city":"New York"}',
                name: 'check_weather'
              },
              type: 'function'
            },
            {
              id: 'call_62136356',
              function: {
                arguments: '{"city":"London"}',
                name: 'check_weather'
              },
              type: 'function'
            },
            {
              id: 'call_62136357',
              function: {
                arguments: '{"city":"Tokyo"}',
                name: 'check_weather'
              },
              type: 'function'
            }
          ]
        }
      };

      // Iterate through tool calls to handle each weather check
      response.message.tool_calls.forEach(tool_call => {
        const arguments = JSON.parse(tool_call.function.arguments);
        const city = arguments.city;
        check_weather(city).then(weather_info => {
          console.log(`Weather in ${city}: ${weather_info}`);
        });
      });
```
Each function call in the array has a unique `id`.

Once you've executed these function calls in your application, you can provide the result back to the model by adding one new message to the conversation for each function call, each containing the result of one function call, with a `tool_call_id` referencing the `id` from `tool_calls`, for example:

Python

```python
# Assume we have fetched the weather data from somewhere
weather_data = {
    "New York": {"temperature": "22°C", "condition": "Sunny"},
    "London": {"temperature": "15°C", "condition": "Cloudy"},
    "Tokyo": {"temperature": "25°C", "condition": "Rainy"}
}

# Prepare the chat completion call payload with inline function call result creation
completion_payload = {
    "model": "gpt-4o",
    "messages": [
        {"role": "system", "content": "You are a helpful assistant providing weather updates."},
        {"role": "user", "content": "Can you tell me the weather in New York, London, and Tokyo?"},
        # Append the original function calls to the conversation
        response['message'],
        # Include the result of the function calls
        {
            "role": "tool",
            "content": json.dumps({
                "city": "New York",
                "weather": weather_data["New York"]
            }),
            # Here we specify the tool_call_id that this result corresponds to
            "tool_call_id": response['message']['tool_calls'][0]['id']
        },
        {
            "role": "tool",
            "content": json.dumps({
                "city": "London",
                "weather": weather_data["London"]
            }),
            "tool_call_id": response['message']['tool_calls'][1]['id']
        },
        {
            "role": "tool",
            "content": json.dumps({
                "city": "Tokyo",
                "weather": weather_data["Tokyo"]
            }),
            "tool_call_id": response['message']['tool_calls'][2]['id']
        }
    ]
}

# Call the OpenAI API's chat completions endpoint to send the tool call result back to the model
response = openai.chat.completions.create(
    model=completion_payload["model"],
    messages=completion_payload["messages"]
)

# Print the response from the API, which will return something like "In New York the weather is..."

print(response)
```

Javascript - NodeJS

```
// Assume we have fetched the weather data from somewhere
const weather_data = {
    "New York": { "temperature": "22°C", "condition": "Sunny" },
    "London": { "temperature": "15°C", "condition": "Cloudy" },
    "Tokyo": { "temperature": "25°C", "condition": "Rainy" }
};

// Prepare the chat completion call payload with inline function call result creation
const completion_payload = {
    model: "gpt-4o",
    messages: [
        { role: "system", content: "You are a helpful assistant providing weather updates." },
        { role: "user", content: "Can you tell me the weather in New York, London, and
Tokyo?" },
        // Append the original function calls to the conversation
        response.message,
        // Include the result of the function calls
        {
            role: "tool",
            content: JSON.stringify({
                city: "New York",
                weather: weather_data["New York"]
            }),
            // Here we specify the tool_call_id that this result corresponds to
            tool_call_id: response.message.tool_calls[0].id
        },
        {
            role: "tool",
            content: JSON.stringify({
                city: "London",
                weather: weather_data["London"]
            }),
            tool_call_id: response.message.tool_calls[1].id
        },
        {
            role: "tool",
            content: JSON.stringify({
                city: "Tokyo",
                weather: weather_data["Tokyo"]
            }),
            tool_call_id: response.message.tool_calls[2].id
        }
    ]
};

// Call the OpenAI API's chat completions endpoint to send the tool call result back to the
model
const response = await openai.chat.completions.create({
    model: completion_payload.model,
    messages: completion_payload.messages
});

// Print the response from the API, which will return something like "In New York the
weather is..."
console.log(response);
```

You can also disable parallel function calling by setting `parallel_tool_calls: false`.

Parallel function calling and Structured Outputs

When the model outputs multiple function calls via parallel function calling, model outputs may not match strict schemas supplied in tools.

In order to ensure strict schema adherence, disable parallel function calls by supplying `parallel_tool_calls: false`. With this setting, the model will generate one function call at a time.

Configuring function calling behavior using the `tool_choice` parameter

By default, the model is configured to automatically select which functions to call, as determined by the `tool_choice: "auto"` setting.

We offer three ways to customize the default behavior:

To force the model to always call one or more functions, you can set `tool_choice: "required"`. The model will then always select one or more function(s) to call. This is useful for example if you want the model to pick between multiple actions to perform next.
To force the model to call a specific function, you can set `tool_choice: {"type": "function", "function": {"name": "my_function"}}`.
To disable function calling and force the model to only generate a user-facing message, you can either provide no tools, or set `tool_choice: "none"`.

Note that if you do either 1 or 2 (i.e. force the model to call a function) then the subsequent `finish_reason` will be `"stop"` instead of being `"tool_calls"`.

Python

```
from openai import OpenAI

client = OpenAI()

tools = [
    {
        "type": "function",
        "function": {
            "name": "get_weather",
            "strict": True,
            "parameters": {
                "type": "object",
                "properties": {
                    "location": {"type": "string"},
                    "unit": {"type": "string", "enum": ["c", "f"]},
                },
                "required": ["location", "unit"],
                "additionalProperties": False,
            },
        },
    },
    {
        "type": "function",
        "function": {
            "name": "get_stock_price",
            "strict": True,
            "parameters": {
                "type": "object",
                "properties": {
                    "symbol": {"type": "string"},
                },
                "required": ["symbol"],
                "additionalProperties": False,
            },
        },
    },
]

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

print(completion)
```

NodeJS - Javascript

```
import { OpenAI } from "openai";
const openai = new OpenAI();

// Define a set of tools to use
const tools = [
  {
    type: "function",
    function: {
      name: "get_weather",
      strict: true,
      parameters: {
        type: "object",
        properties: {
          location: { type: "string" },
          unit: { type: "string", enum: ["c", "f"] },
        },
        required: ["location", "unit"],
        additionalProperties: false,
      },
    },
  },
  {
    type: "function",
    function: {
      name: "get_stock_price",
      strict: true,
      parameters: {
        type: "object",
        properties: {
          symbol: { type: "string" },
        },
        required: ["symbol"],
        additionalProperties: false,
      },
    },
  },
];

// Call the OpenAI API's chat completions endpoint
const response = await openai.chat.completions.create({
  model: "gpt-4o",
  messages: [
    {
      role: "user",
      content: "Can you tell me the weather in Tokyo?",
    },
  ],
  // highlight-start
  tool_choice: "required",
  // highlight-end
  tools,
});

// Print the response from the API
console.log(response);
```

Understanding 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.

Tips and best practices

Turn on Structured Outputs by setting `strict: "true"`

When Structured Outputs is turned on, the arguments generated by the model for function calls will reliably match the JSON Schema that you provide.

If you are not using Structured Outputs, then the structure of arguments is not guaranteed to be correct, so we recommend the use of a validation library like Pydantic to first verify the arguments prior to using them.

Name functions intuitively, with detailed descriptions

If you find the model does not generate calls to the correct functions, you may need to update your function names and descriptions so the model more clearly understands when it should select each function. Avoid using abbreviations or acronyms to shorten function and argument names.

You can also include detailed descriptions for when a tool should be called. For complex functions, you should include descriptions for each of the arguments to help the model know what it needs to ask the user to collect that argument.

Name function parameters intuitively, with detailed descriptions

Use clear and descriptive names for function parameters. For example, specify the expected format for a date parameter (e.g., YYYY-mm-dd or dd/mm/yy) in the description.

Consider providing additional information about how and when to call functions in your system message

Providing clear instructions in your system message can significantly improve the model's function calling accuracy. For example, guide the model with things like, "Use check_order_status when the user inquires about the status of their order, such as 'Where is my order?' or 'Has my order shipped yet?'". Provide context for complex scenarios, like "Before scheduling a meeting with schedule_meeting, check the user's calendar for availability using check_availability to avoid conflicts."

Use enums for function arguments when possible

If your use case allows, you can use enums to constrain the possible values for arguments. This can help reduce hallucinations.

For example, say you have an AI assistant that helps with ordering a T-shirt. You likely have a fixed set of sizes for the T-shirt, and you might want the model to output in a specific format. If you want the model to output “s”, “m”, “l”, etc for small, medium, and large, then you could provide those values in the enum, for example:

```
{
   "name": "pick_tshirt_size",
   "description": "Call this if the user specifies which size t-shirt they want",
   "parameters": {
      "type": "object",
      "properties": {
         "size": {
            "type": "string",
            "enum": ["s", "m", "l"],
            "description": "The size of the t-shirt that the user would like to order"
         }
      },
      "required": ["size"],
      "additionalProperties": false
   }

}
```

If you don’t constrain the output, a user may say “large” or “L”, and the model may return either value. Your code may expect a specific structure, so it’s important to limit the number of possible formats the model can choose from.

Keep the number of functions low for higher accuracy

We recommend that you use no more than 20 functions in a single tool call. Developers typically see a reduction in the model's ability to select the correct tool once they have between 10-20 tools.

If your use case requires the model to be able to pick between a large number of functions, you may want to explore fine-tuning (learn more) or break out the tools and group them logically to create a multi-agent system.

Set up evals to act as an aid in prompt engineering your function definitions and system messages

We recommend for non-trivial uses of function calling that you set up a suite of evals that allow you to measure how frequently the correct function is called or correct arguments are generated for a wide variety of possible user messages. Learn more about setting up evals on the OpenAI Cookbook.

You can then use these to measure whether adjustments to your function definitions and system messages will improve or hurt your integration.

Fine-tuning may help improve accuracy for function calling

Fine-tuning a model can improve performance at function calling for your use case, especially if you have a large number of functions, or complex, nuanced or similar functions.

See our fine-tuning for function calling cookbook for more information.

Fine-tuning for function calling

Learn how to fine-tune a model for function calling

FAQ

How do functions differ from tools?

When using function calling with the OpenAI API, you provide them as `tools`, configure them with `tool_choice` and monitor for `finish_reason: "tool_calls"`.

The parameters named things like `functions` and `function_call` etc are now deprecated.

Should I include function call instructions in the tool specification or in the system prompt?

We recommend including instructions regarding when to call a function in the system prompt, while using the function definition to provide instructions on how to call the function and how to generate the parameters.

Which models support function calling?

Function calling was introduced with the release of `gpt-4-turbo` on June 13, 2023. This includes: `gpt-4o`, `gpt-4o-2024-08-06`, `gpt-4o-2024-05-13`, `gpt-4o-mini`, `gpt-4o-mini-2024-07-18`, `gpt-4-turbo`, `gpt-4-turbo-2024-04-09`, `gpt-4-turbo-preview`, `gpt-4-0125-preview`, `gpt-4-1106-preview`, `gpt-4`, `gpt-4-0613`, `gpt-3.5-turbo`, `gpt-3.5-turbo-0125`, `gpt-3.5-turbo-1106`, and `gpt-3.5-turbo-0613`.

Legacy models released before this date were not trained to support function calling.

Parallel function calling is supported on models released on or after Nov 6, 2023. This includes: `gpt-4o`, `gpt-4o-2024-08-06`, `gpt-4o-2024-05-13`, `gpt-4o-mini`, `gpt-4o-mini-2024-07-18`, `gpt-4-turbo`, `gpt-4-turbo-2024-04-09`, `gpt-4-turbo-preview`, `gpt-4-0125-preview`, `gpt-4-1106-preview`, `gpt-3.5-turbo`, `gpt-3.5-turbo-0125`, and `gpt-3.5-turbo-1106`.

What are some example functions?

Data Retrieval:

> Scenario: A chatbot needs to fetch the latest customer data from an internal system when a user asks “who are my top customers?”
> Implementation: Define a function`get_customers(min_revenue: int, created_before: string, limit: int)` that retrieves customer data from your

internal API. The model can suggest calling this function with the appropriate parameters based on user input.

Task Automation:

Scenario: An assistant bot schedules meetings based on user preferences and calendar availability.
Implementation: Define a function scheduleMeeting(date: str, time: str, participants: list) that interacts with a calendar API. The model can suggest the best times and dates to call this function.

Computational Tasks:

Scenario: A financial application calculates loan payments based on user input.
Implementation: Define a function calculateLoanPayment(principal: float, interestRate: float, term: int) to perform the necessary calculations. The model can provide the input values for this function.

Customer Support:

Scenario: A customer support bot assists users by providing the status of their orders.
Implementation: Define a function getOrderStatus(orderId: str) that retrieves order status information from a database. The model can suggest calling this function with the appropriate order ID parameter based on user input.

Can the model execute functions itself?

No, the model only suggests function calls and generates arguments. Your application handles the execution of the functions based on these suggestions (and returns the results of calling those functions to the model).

What are Structured Outputs?

Structured Outputs, introduced in August 2024, is a feature that ensures that the arguments generated by the model exactly match the provided JSON Schema, enhancing reliability and reducing errors. We recommend its use and it can be enabled by setting `"strict": true`.

Why might I not want to turn on Structured Outputs?

The main reasons to not use Structured Outputs are:

- If you need to use some feature of JSON Schema that is not yet supported (learn more), for example recursive schemas.
- If each of your API requests will include a novel schema (i.e. your schemas are not fixed, but are generated on-demand and rarely repeat), since the first request with a novel JSON Schema will have increased latency as the schema is pre-processed and cached for future generations to constrain the output of the model.

How do I ensure the model calls the correct function?

Use intuitive names and detailed descriptions for functions and parameters. Provide clear guidance in the system message to enhance the model's ability to pick the correct function.

What does Structured Outputs mean for Zero Data Retention?

When Structured Outputs is turned on, schemas provided are not eligible for zero data retention.

Resources

The OpenAI Cookbook has several end-to-end examples to help you implement function calling. In our introductory cookbook how to call functions with chat models, we outline two examples of how the models can use function calling. This one is a great resource to follow as you get started:

Function calling

Learn from more examples demonstrating function calling

You can find more examples to help you get started with function calling in the OpenAI Cookbook.

How to call functions with chat models cookbook

How to use functions with a knowledge base cookbook

Assistants API overview cookbook

Fine-tuning for function calling cookbook