Function calling

Learn how to connect large language models to external tools.

Introduction

In an API call, you can describe functions and have the model intelligently choose to output a JSON object containing arguments to call one or many functions. The Chat Completions API does not call the function; instead, the model generates JSON that you can use to call the function in your code.

The latest models (`gpt-3.5-turbo-0125` and `gpt-4-turbo-preview`) have been trained to both detect when a function should to be called (depending on the input) and to respond with JSON that adheres to the function signature more closely than previous models. With this capability also comes potential risks. We strongly recommend building in user confirmation flows before taking actions that impact the world on behalf of users (sending an email, posting something online, making a purchase, etc).

> This guide is focused on function calling with the Chat Completions API, for details on function calling in the Assistants API, please see the Assistants Tools page.

Common use cases

Function calling allows you to more reliably get structured data back from the model. For example, you can:

- Create assistants that answer questions by calling external APIs
  - e.g. define functions like `send_email(to: string, body: string)`, or `get_current_weather(location: string, unit: 'celsius' | 'fahrenheit')`
- Convert natural language into API calls
  - e.g. convert "Who are my top customers?" to `get_customers(min_revenue: int, created_before: string, limit: int)` and call your internal API
- Extract structured data from text
  - e.g. define a function called `extract_data(name: string, birthday: string)`, or `sql_query(query: string)`

...and much more!

The basic sequence of steps for function calling is as follows:

1. Call the model with the user query and a set of functions defined in the functions parameter.
2. The model can choose to call one or more functions; if so, the content will be a stringified JSON object adhering to your custom schema (note: the model may hallucinate parameters).
3. Parse the string into JSON in your code, and call your function with the provided arguments if they exist.
4. Call the model again by appending the function response as a new message, and let the model summarize the results back to the user.

Supported models

Not all model versions are trained with function calling data. Function calling is supported with the following models: `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`

In addition, parallel function calls is supported on the following models: `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-0125`, and `gpt-3.5-turbo-1106`.

Function calling behavior

The default behavior for `tool_choice` is `tool_choice: "auto"`. This lets the model decide whether to call functions and, if so, which functions to call.

We offer three ways to customize the default behavior depending on your use case:

1. To force the model to always call one or more functions, you can set `tool_choice: "required"`. The model will then select which function(s) to call.
2. To force the model to call only one specific function, you can set `tool_choice: {"type": "function", "function": {"name": "my_function"}}`.
3. To disable function calling and force the model to only generate a user-facing message, you can set `tool_choice: "none"`.

Parallel function calling

Parallel function calling is the model's ability to perform multiple function calls together, allowing the effects and results of these function calls to be resolved in parallel. This is especially useful if functions take a long time, and reduces round trips with the API. 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, each with an `id`. To respond to these function calls, add 3 new messages to the conversation, each containing the result of one function call, with a `tool_call_id` referencing the `id` from `tool_calls`.

In this example, we define a single function `get_current_weather`. The model calls the function multiple times, and after sending the function response back to the model, we let it decide the next step. It responded with a user-facing message which was telling the user the temperature in San Francisco, Tokyo, and Paris. Depending on the query, it may choose to call a function again.

Example invoking multiple function calls in one response

python

```python
from openai import OpenAI
import json

client = OpenAI()

# Example dummy function hard coded to return the same weather
# In production, this could be your backend API or an external API
def get_current_weather(location, unit="fahrenheit"):
    """Get the current weather in a given location"""
    if "tokyo" in location.lower():
        return json.dumps({"location": "Tokyo", "temperature": "10", "unit": unit})
    elif "san francisco" in location.lower():
        return json.dumps({"location": "San Francisco", "temperature": "72", "unit": un
    elif "paris" in location.lower():
        return json.dumps({"location": "Paris", "temperature": "22", "unit": unit})
    else:
        return json.dumps({"location": location, "temperature": "unknown"})

def run_conversation():
    # Step 1: send the conversation and available functions to the model
    messages = [{"role": "user", "content": "What's the weather like in San Francisco,
    tools = [
        {
            "type": "function",
            "function": {
                "name": "get_current_weather",
                "description": "Get the current weather in a given location",
                "parameters": {
                    "type": "object",
                    "properties": {
                        "location": {
                            "type": "string",
                            "description": "The city and state, e.g. San Francisco, CA"
                        },
                        "unit": {"type": "string", "enum": ["celsius", "fahrenheit"]},
                    },
                    "required": ["location"],
                },
            },
        }
    ]
    response = client.chat.completions.create(
        model="gpt-3.5-turbo-0125",
        messages=messages,
        tools=tools,
        tool_choice="auto",  # auto is default, but we'll be explicit
    )
    response_message = response.choices[0].message
    tool_calls = response_message.tool_calls
    # Step 2: check if the model wanted to call a function
    if tool_calls:
        # Step 3: call the function
        # Note: the JSON response may not always be valid; be sure to handle errors
        available_functions = {
            "get_current_weather": get_current_weather,
        }  # only one function in this example, but you can have multiple
        messages.append(response_message)  # extend conversation with assistant's reply
        # Step 4: send the info for each function call and function response to the mod
        for tool_call in tool_calls:
            function_name = tool_call.function.name
            function_to_call = available_functions[function_name]
            function_args = json.loads(tool_call.function.arguments)
            function_response = function_to_call(
                location=function_args.get("location"),
                unit=function_args.get("unit"),
            )
            messages.append(
                {
                    "tool_call_id": tool_call.id,
                    "role": "tool",
                    "name": function_name,
                    "content": function_response,
                }
            )  # extend conversation with function response
        second_response = client.chat.completions.create(
            model="gpt-3.5-turbo-0125",
            messages=messages,
        )  # get a new response from the model where it can see the function response


        return second_response
print(run_conversation())
```

You can find more examples of function calling in the OpenAI Cookbook:

Function calling

Learn from more examples demonstrating function calling

Tokens

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 running into context limits, we suggest limiting the number of functions or the length of documentation 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.