Azure functions example

K kristapratico
Jul 21, 2023

Open in Github

This notebook shows how to use the function calling capability with the Azure OpenAI service. Functions allow a caller of chat completions to define capabilities that the model can use to extend its functionality into external tools and data sources.

You can read more about chat functions on OpenAI's blog: https://openai.com/blog/function-calling-and-other-api-updates

NOTE: Chat functions require model versions beginning with gpt-4 and gpt-35-turbo's `-0613` labels. They are not supported by older versions of the models.

Setup

First, we install the necessary dependencies and import the libraries we will be using.

```
! pip install "openai>=1.0.0,<2.0.0"
! pip install python-dotenv
```

```
import os
import openai
```

Cookbook Topics About API Docs Contribute

Authentication

The Azure OpenAI service supports multiple authentication mechanisms that include API keys and Azure Active Directory token credentials.

```
use_azure_active_directory = False  # Set this flag to True if you are using Azure Active Directo
```

Authentication using API key

To set up the OpenAI SDK to use an *Azure API Key*, we need to set `api_key` to a key associated with your endpoint (you can find this key in *"Keys and Endpoints"* under *"Resource Management"* in the Azure Portal). You'll also find the endpoint for your resource here.

```
if not use_azure_active_directory:
    endpoint = os.environ["AZURE_OPENAI_ENDPOINT"]
    api_key = os.environ["AZURE_OPENAI_API_KEY"]

    client = openai.AzureOpenAI(
        azure_endpoint=endpoint,
        api_key=api_key,
        api_version="2023-09-01-preview"
    )
```

Authentication using Azure Active Directory

Let's now see how we can authenticate via Azure Active Directory. We'll start by installing the `azure-identity` library. This library will provide the token credentials we need to authenticate and help us build a token credential provider through the `get_bearer_token_provider` helper function. It's recommended to use `get_bearer_token_provider` over providing a static token to `AzureOpenAI` because this API will automatically cache and refresh tokens for you.

For more information on how to set up Azure Active Directory authentication with Azure OpenAI, see the documentation.

```
! pip install "azure-identity>=1.15.0"
```

```
from azure.identity import DefaultAzureCredential, get_bearer_token_provider

if use_azure_active_directory:
    endpoint = os.environ["AZURE_OPENAI_ENDPOINT"]
    api_key = os.environ["AZURE_OPENAI_API_KEY"]

    client = openai.AzureOpenAI(
        azure_endpoint=endpoint,
        azure_ad_token_provider=get_bearer_token_provider(DefaultAzureCredential(), "https://cogn
        api_version="2023-09-01-preview"
    )
```

> *"Note: the AzureOpenAI infers the following arguments from their corresponding environment variables if they are not provided:"*

- `api_key` from `AZURE_OPENAI_API_KEY`
- `azure_ad_token` from `AZURE_OPENAI_AD_TOKEN`
- `api_version` from `OPENAI_API_VERSION`
- `azure_endpoint` from `AZURE_OPENAI_ENDPOINT`

Deployments

In this section we are going to create a deployment of a GPT model that we can use to call functions.

Deployments: Create in the Azure OpenAI Studio

Let's deploy a model to use with chat completions. Go to https://portal.azure.com, find your Azure OpenAI resource, and then navigate to the Azure OpenAI Studio. Click on the "Deployments" tab and then create a deployment for the model you want to use for chat completions. The deployment name that you give the model will be used in the code below.

```
deployment = "" # Fill in the deployment name from the portal here
```

Functions

With setup and authentication complete, you can now use functions with the Azure OpenAI service. This will be split into a few steps:

1. Define the function(s)
2. Pass function definition(s) into chat completions API
3. Call function with arguments from the response
4. Feed function response back into chat completions API

1. Define the function(s)

A list of functions can be defined, each containing the name of the function, an optional description, and the parameters the function accepts (described as a JSON schema).

```
functions = [
    {
        "name": "get_current_weather",
        "description": "Get the current weather",
        "parameters": {
            "type": "object",
            "properties": {
                "location": {
                    "type": "string",
                    "description": "The city and state, e.g. San Francisco, CA",
                },
                "format": {
                    "type": "string",
                    "enum": ["celsius", "fahrenheit"],
                    "description": "The temperature unit to use. Infer this from the users locati
                },
            },
            "required": ["location"],
        },
    }
]
```

2. Pass function definition(s) into chat completions API

Now we can pass the function into the chat completions API. If the model determines it should call the function, a `finish_reason` of "tool_calls" will be populated on the choice and the details of which function to call and its arguments will be present in the `message`. Optionally, you can set the `tool_choice` keyword argument to force the model to call a particular function (e.g. `{"type": "function", "function": {"name": get_current_weather}}`). By default, this is set to `auto`, allowing the model to choose whether to call the function or not.

```
messages = [
    {"role": "system", "content": "Don't make assumptions about what values to plug into function
    {"role": "user", "content": "What's the weather like today in Seattle?"}
]

chat_completion = client.chat.completions.create(
    model=deployment,
    messages=messages,
    tools=functions,
)
print(chat_completion)
```

3. Call function with arguments from the response

The name of the function call will be one that was provided initially and the arguments will include JSON matching the schema included in the function definition.

```
import json

def get_current_weather(request):
    """
    This function is for illustrative purposes.
    The location and unit should be used to determine weather
    instead of returning a hardcoded response.
    """
    location = request.get("location")
    unit = request.get("unit")
    return {"temperature": "22", "unit": "celsius", "description": "Sunny"}

function_call = chat_completion.choices[0].message.tool_calls[0].function
print(function_call.name)
print(function_call.arguments)

if function_call.name == "get_current_weather":
    response = get_current_weather(json.loads(function_call.arguments))
```

4. Feed function response back into chat completions API

The response from the function should be serialized into a new message with the role set to "function". Now the model will use the response data to formulate its answer.

```
messages.append(
    {
        "role": "function",
        "name": "get_current_weather",
        "content": json.dumps(response)
    }
)

function_completion = client.chat.completions.create(
    model=deployment,
    messages=messages,
    tools=functions,
)

print(function_completion.choices[0].message.content.strip())
```