The Power of OpenAI's Function Calling in Language Learning Models: A Comprehensive Guide
Transforming Data Pipelines with OpenAI's Function Calling Feature: Implementing an Email Sending Workflow Using PostgreSQL and FastAPI

Luís Roque
Towards Data Science

Luís Roque
Published in

Towards Data Science
·
11 min read
·
Jun 22

Introduction

The exciting world of AI has taken another leap forward by the introduction of function calling capabilities in OpenAI's latest Large Language Models (LLMs). This new feature enhances the interaction between humans and AI, transforming it from a simple question-and-answer format to a more dynamic and active dialogue.

But what exactly are these function calling capabilities? At their core, they allow the LLM to call predefined functions during a conversation based on the input instructions. This could be anything from sending an email to fetching data from a database based on the context of the conversation.
The benefits and applications of using function calling capabilities are vast. It significantly increases the dynamic utility of AI in various applications, from customer service to how we build data pipelines.

Imagine a customer service AI that can answer questions and perform actions such as booking appointments, sending information to an email address, or updating customer details in real time. Or consider a data pipeline where the LLM agent can fetch, update, and manipulate data on command.

In the upcoming sections, we will explore these applications further and provide a step-by-step guide on leveraging this new capability by building an email-sending pipeline.
Figure 1: LLMs are becoming intelligent agents that we can work with (image source)

As always, the code is available on my Github.
Understanding the New Function Calling Capability and Comparing It to LangChain

We want to understand what OpenAI's newly introduced function calling feature brings to the AI race. For that, let's understand what differentiates it from other tools and frameworks in the market, like LangChain. We already introduced LangChain in the first article of this series. It is a popular framework for developing AI-powered applications.

The function calling feature and LangChain bring unique advantages and capabilities to the table and are built around making AI more usable, versatile, and dynamic.

We already know that the function calling feature adds an extra layer of interactivity to AI applications. It enables the model to call predefined functions within a conversation, enhancing the LLM's dynamism and reactivity. This new feature can potentially simplify the process of adding functionality to AI applications. Developers would need to define these functions, and the model could then execute them as part of the conversation, depending on the context. The primary advantage here is its direct integration with the OpenAI models, which facilitates ease of use, quick setup, and a lower learning curve for developers familiar with the OpenAI ecosystem.

On the other hand, LangChain offers a comprehensive and versatile framework designed for developing more complex, data-aware, and agentic AI applications. It allows a language model to interact with its environment and make decisions based on high-level directives. Its modules provide abstractions and standard interfaces for building applications, including models, prompts, memory, indexes, chains, agents, and callbacks.

LangChain's approach facilitates building applications where an LLM can persist its state across different interactions, sequence and chain calls to different utilities, and even interact with external data sources. We can see these as function calling capabilities on steroids. Thus, it is particularly useful for developers building complex, multi-step applications that leverage language models. The downside of the added complexity is that it creates a steeper learning curve to use it fully.
Use Case — Transforming Data Pipelines

In my view, data pipelines are among the most exciting application areas for the new function calling capabilities in LLMs. Data pipelines are a critical component of any data-driven organization that collects, processes, and distributes data. Typically, these processes are static, predefined, and require manual intervention for any changes or updates. This is where the dynamic behavior of an LLM that we discussed above creates an opportunity.

Traditionally, database querying requires specific knowledge of query languages like SQL. With LLMs' ability to call functions, services, and databases directly, users can retrieve data conversationally without the need for explicit query formulation. An LLM could translate a user's request into a database query, fetch the data, and return it in a user-friendly format, all in real time. This feature could democratize data access across different roles within an organization.

Another aspect that could change is data transformation. It often requires separate data cleaning and processing steps before analysis. An LLM could streamline this process by performing data cleaning and manipulation tasks interactively based on the user's instructions. Moreover, manipulating real-time data during a conversation allows for more exploratory and iterative data analysis.

A third use case is data monitoring. It involves regular checks to ensure the accuracy and consistency of data in a data pipeline. With LLMs, monitoring tasks can become more interactive and efficient. For instance, an LLM can alert users to data inconsistencies during conversations and take corrective actions immediately.

Finally, the LLMs can also automate the creation and distribution of data reports. Users can instruct the LLM to generate a report based on specific criteria, and the LLM can fetch the data, create the report, and even send it to the relevant recipients.

Building an Email Sending Pipeline with OpenAI Function Calling Capabilities

We aim to create a pipeline that sends an email to a user. While this may sound straightforward, the beauty of this process lies in the interplay of the different components controlled by the LLM. The AI model doesn't merely generate an email body; it dynamically interacts with a database to retrieve user information, composes a contextually appropriate email, and then instructs a service to send it.

Our pipeline consists of three main components: PostgreSQL, FastAPI, and the OpenAI LLM. We use PostgreSQL to store our user data. This data includes user names and their associated email addresses. It serves as our source of truth for user information. FastAPI is a modern, high-performance web framework for building APIs with Python. We use a FastAPI service to simulate the process of sending an email. When the service receives a request to send an email, it returns a response confirming the email has been sent. The LLM serves as the orchestrator of the entire process. It controls the conversation, determines the necessary actions based on the context, interacts with the PostgreSQL database to fetch user information, crafts an email message, and instructs the FastAPI service to send the email.

Implementing PostgreSQL Database

The first component of our pipeline is the PostgreSQL database where we store our user data. Setting up a PostgreSQL instance is made easy and reproducible with Docker, a platform that allows us to containerize and isolate our database environment.

You first need to install Docker to set up a PostgreSQL Docker container. Once installed, you can pull the PostgreSQL image and run it as a container. We map the container's port 5432 to the host's port 5432 to access the database. In a production environment, please set your password as an environment variable and do not set them directly in a command as below. We are doing this way to speed up our process.

```
docker run --name user_db -e POSTGRES_PASSWORD=testpass -p 5432:5432 -d postgres
```

With our PostgreSQL instance running, we can now create a database and a table to store our user data. We'll use an initialization script to create a userstable with username and emailcolumns and populate it with some dummy data. This script is placed in a directory which is then mapped to the /docker-entrypoint-initdb.d directory in the container. PostgreSQL

executes any scripts found in this directory upon startup. Here's what the script (user_init.sql) looks like:

```
CREATE DATABASE user_database;
\c user_database;

CREATE TABLE users (
    username VARCHAR(50),
    email VARCHAR(50)
);

INSERT INTO users (username, email) VALUES
    ('user1', 'user1@example.com'),
    ('user2', 'user2@example.com'),
    ('user3', 'user3@example.com'),
    ...
    ('user10', 'user10@example.com');
```

The LLM is capable of understanding SQL commands and can be used to query the PostgreSQL database. When the LLM receives a request that involves fetching user data, it can formulate a SQL query to fetch the necessary data from the database.

For instance, if you ask the LLM to send an email to user10, the LLM can formulate the query :

```
SELECT  email FROM users WHERE username='user10';
```

This allows it to fetch user10 email address from the users table. The LLM can then use this email address to instruct the FastAPI service to send the email.

In the next section, we will guide you through the implementation of the FastAPI service that sends the emails.

Creating the FastAPI Email Service

Our second component is a FastAPI service. This service will simulate the process of sending emails. It's a straightforward API that receives a POST request containing the recipient's name, email, and the email body. It will return a response confirming the email was sent. We will again use Docker to ensure our service is isolated and reproducible.

First, you need to install Docker (if not already installed). Then, create a new directory for your FastAPI service and move into it. Here, create a new Python file (e.g., main.py) and add the following code:

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()


class User(BaseModel):
    name: str
```

```
    email: str
    body: str


@app.post("/send_email")
async def send_email(user: User):
    return {
        "message": f"Email successfully sent to {user.name} with email
{user.email}. Email body:\n\n{user.body}"
    }
```

This code defines a FastAPI application with a single endpoint /send_email/. This endpoint accepts POST requests and expects a JSON body containing the recipient's name, email, and the email body.

Next, create a Dockerfile in the same directory with the following content:

```
FROM python:3.9-slim-buster

WORKDIR /app
ADD . /app

RUN pip install --no-cache-dir fastapi uvicorn

EXPOSE 1000

CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "1000"]
```

This Dockerfile instructs Docker to create an image based on thepython:3.9-slim-buster image, a light image ideal for efficiently running python applications. It then copies our main.py file into the /app/ directory in the image.

You can build the Docker image using the command:

```
docker build -t fastapi_email_service .
```

And then run it with:

```
docker run -d -p 1000:1000 fastapi_email_service
```

The LLM interacts with the FastAPI service using a POST request. When the LLM decides to send an email, it generates a function call to the send_email function. The arguments of this function call contain the name, email, and the email body.

The function call is processed by our Python script, which extracts the function arguments and uses them to send a POST request to our FastAPI service. The FastAPI service responds with a message indicating the email was successfully sent.

Now, we have all components of our pipeline. The next section will tie everything together, explaining how the LLM orchestrates the interaction between the PostgreSQL database and the FastAPI service to send an email.
Integrating with OpenAI LLM

The last piece of our pipeline is the OpenAI LLM integration. The LLM serves as the orchestrator, interpreting our commands, querying the database for user information, and instructing the FastAPI service to send emails.

Our script uses OpenAI's API to make chat-based completions with the LLM. Each completion request consists of a series of messages and optionally a list of function specifications that the model could call. We start the conversation with a user message, which provides a prompt to the assistant.

Here's the chat_completion_request function we use to send a request to the API:

```
@retry(wait=wait_random_exponential(min=1, max=40),
stop=stop_after_attempt(3))
def chat_completion_request(messages, functions=None, model=GPT_MODEL):
    headers = {
        "Content-Type": "application/json",
        "Authorization": "Bearer " + openai.api_key,
    }
    json_data = {"model": model, "messages": messages}
    if functions is not None:
        json_data.update({"functions": functions})

    response = requests.post(
        "https://api.openai.com/v1/chat/completions",
        headers=headers,
        json=json_data,
    )
    return response
```

We use the Chat class to manage the conversation history. It has methods to add a new message to the history and display the entire conversation:

```
class Chat:
    def __init__(self):
        self.conversation_history = []

    def add_prompt(self, role, content):
        message = {"role": role, "content": content}
        self.conversation_history.append(message)

    def display_conversation(self):
        for message in self.conversation_history:
            print(f"{message['role']}: {message['content']}")
```

In our use case, the LLM needs to interact with our PostgreSQL database and FastAPI service. We define these functions and include them in our

completion request. Here's how we define our sql_query_email and send_emailfunctions:

```
functions = [
    {
        "name": "send_email",
        "description": "Send a new email",
        "parameters": {
            "type": "object",
            "properties": {
                "to": {
                    "type": "string",
                    "description": "The destination email.",
                },
                "name": {
                    "type": "string",
                    "description": "The name of the person that will
receive the email.",
                },
                "body": {
                    "type": "string",
                    "description": "The body of the email.",
                },
            },
            "required": ["to", "name", "body"],
        },
    },
    {
        "name": "sql_query_email",
        "description": "SQL query to get user emails",
        "parameters": {
            "type": "object",
            "properties": {
                "query": {
                    "type": "string",
                    "description": "The query to get users emails.",
                },
            },
            "required": ["query"],
        },
    },
]
```

When we make a completion request, the LLM responds with its intended actions. If the response includes a function call, our script executes that function. For example, if the LLM decides to call the sql_query_emailfunction, our script retrieves the user's email from the database and then adds the result to the conversation history. When the send_emailfunction is called, our script sends an email using the FastAPI service.

The main loop of our script checks for function calls in the LLM's response and acts accordingly:

```
chat = Chat()
chat.add_prompt("user", "Send an email to user10 saying that he needs to
pay the monthly subscription fee.")
result_query = ''

for i in range(2):
    chat_response = chat_completion_request(
        chat.conversation_history,
        functions=functions
    )
    response_content = chat_response.json()['choices'][0]['message']

    if 'function_call' in response_content:
        if response_content['function_call']['name'] == 'send_email':
            res =
json.loads(response_content['function_call']['arguments'])
            send_email(res['name'], res['to'], res['body'])
            break
        elif response_content['function_call']['name'] ==
'sql_query_email':
            result_query =
query_db(json.loads(response_content['function_call']['arguments'])['query
'])
            chat.add_prompt('user', str(result_query))
    else:
        chat.add_prompt('assistant', response_content['content'])
```

When we run the script, we get the following output:

```
{
  "message": "Email successfully sent to User 10 with email
user10@example.com.",
  "Email body": "\n\nDear User 10, \n\nThis is a reminder that your
monthly subscription fee is due. Please make the payment as soon as
possible to ensure uninterrupted service. Thank you for your cooperation.
\n\nBest regards, \nYour Subscription Service Team"
}
```

Let’s break down what happened for us to get this output. Our prompt was “Send an email to user10 saying that he needs to pay the monthly subscription fee.”. Note that there is no information about the user10email in our message. The LLM identified the missing information and understood that our function query_emailwould allow it to query a database to get the email from that user. After getting the email, it got two things right once again: first, it should generate the body of the email, and second, it should call the send_emailfunction to trigger the email using the FastAPI email service.

Conclusion

This article explored the function calling feature by implementing a case study where the LLM coordinates a pipeline involving a PostgreSQL database and a FastAPI email service. The LLM successfully navigated the task of retrieving a user’s email from the database and instructing the email

service to send a personalized message, all in response to a single prompt.

The implications of function calling in AI models could be enormous, opening up new possibilities for automating and streamlining processes. Data pipelines could change from static and engineering heavy to dynamic entities, allowing non-technical users to quickly get their hands on the latest data using natural language.

Large Language Models Chronicles: Navigating the NLP Frontier

This article belongs to “Large Language Models Chronicles: Navigating the NLP Frontier”, a new weekly series of articles that will explore how to leverage the power of large models for various NLP tasks. By diving into these cutting-edge technologies, we aim to empower developers, researchers, and enthusiasts to harness the potential of NLP and unlock new possibilities.

Articles published so far:

- Summarizing the latest Spotify releases with ChatGPT
- Master Semantic Search at Scale: Index Millions of Documents with Lightning-Fast Inference Times using FAISS and Sentence Transformers
- Unlock the Power of Audio Data: Advanced Transcription and Diarization with Whisper, WhisperX, and PyAnnotate
- Whisper JAX vs PyTorch: Uncovering the Truth about ASR Performance on GPUs
- Vosk for Efficient Enterprise-Grade Speech Recognition: An Evaluation and Implementation Guide
- Testing the Massively Multilingual Speech (MMS) Model that Supports 1162 Languages
- Harnessing the Falcon 40B Model, the Most Powerful Open-Source LLM

Keep in touch: LinkedIn