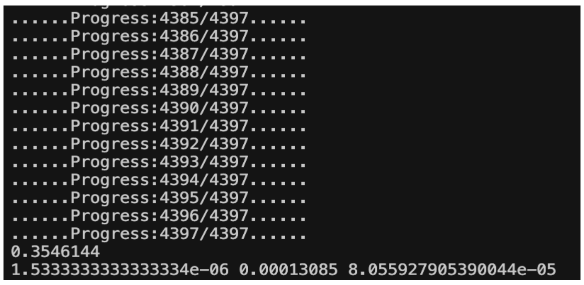
Name: Nguyễn Nam Quân
ID: 2312568
Class: 23APCS1

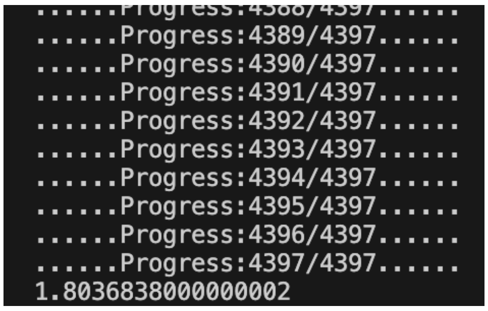
WEEK 10 + 11 Reports

Week 10: Reports of the running time of all the algorithms.

1. **Dijkstra**:
   Time for running all_ pairs : 0.35 minutes (
   Time for running path k importances: 1.8158150999999998 minutes
   Time to run each one to another time:
   + Average: 8.126068910620877e-05 minutes.
   + Fastest: 1.7166666666666666e-06 minutes
   + Slowest: 0.0007252666666666667 minutes.

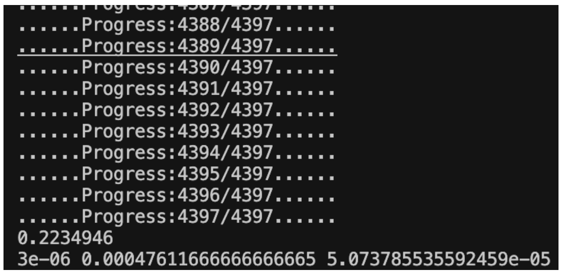


```
......Progress:4385/4397......
......Progress:4386/4397......
......Progress:4387/4397......
......Progress:4388/4397......
......Progress:4389/4397......
......Progress:4390/4397......
......Progress:4391/4397......
......Progress:4392/4397......
......Progress:4393/4397......
......Progress:4394/4397......
......Progress:4395/4397......
......Progress:4396/4397......
......Progress:4397/4397......
0.3546144
1.5333333333333334e-06 0.00013085 8.055927905390044e-05
```

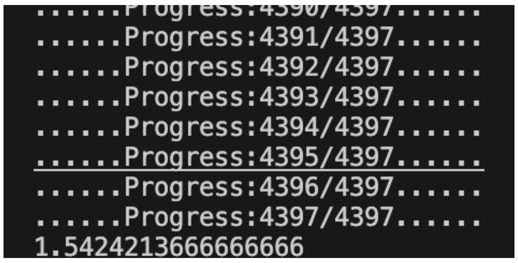


```
......Progress:4388/4397......
......Progress:4389/4397......
......Progress:4390/4397......
......Progress:4391/4397......
......Progress:4392/4397......
......Progress:4393/4397......
......Progress:4394/4397......
......Progress:4395/4397......
......Progress:4396/4397......
......Progress:4397/4397......
1.8036838000000002
```

2. **Esposo-Pape** (Main replacement for dijkstra)
   => Good replacement for dijkstra for large data handling.
   Time for running all_ pairs: 0.22 minutes (All pairs)
   Time for running path k importances: 1.52774145 minutes
   Time to run each one to another time: 5.723333333333333e-05 minutes
   + Average: 4.974724433325747e-05 minutes.
   + Fastest: 2.9666666666666665e-06 minutes
   + Slowest: 0.00018994999999999998 minutes.



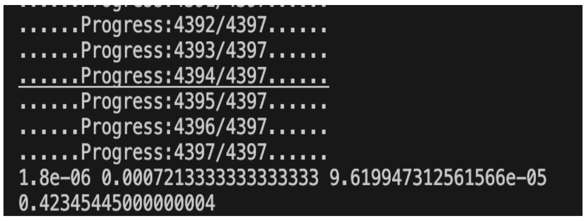
```
......Progress:4387/4397......
......Progress:4388/4397......
......Progress:4389/4397......
......Progress:4390/4397......
......Progress:4391/4397......
......Progress:4392/4397......
......Progress:4393/4397......
......Progress:4394/4397......
......Progress:4395/4397......
......Progress:4396/4397......
......Progress:4397/4397......
0.2234946
3e-06 0.00047611666666666665 5.073785535592459e-05
```



```
......Progress:4390/4397......
......Progress:4391/4397......
......Progress:4392/4397......
......Progress:4393/4397......
......Progress:4394/4397......
......Progress:4395/4397......
......Progress:4396/4397......
......Progress:4397/4397......
1.5424213666666666
```

3. **Floyd_warshall**: As the complexity is O(n^3) This algorithm takes a lot of time to run.
   Time for running all_ pairs: > 1h
   Time for running path k importances: > 1h

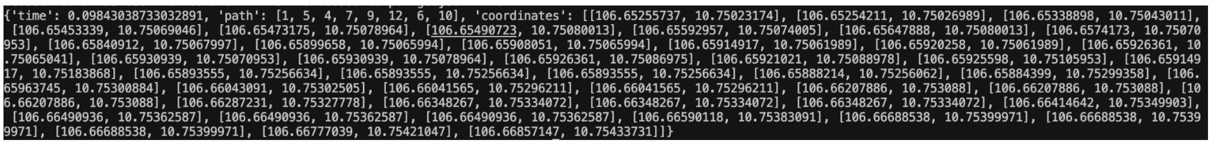
4. **BFS**:
   (Mainly from dijkstra's algo so do not have better performance)
   Time for running all_ pairs : 0.41319075 Minutes
   Time for running path k importances: about 2 minutes
   Time to run each one to another time: 5.7233333333333333e-05 minutes
   + Average: 9.687130619361675e-05 minutes.
   + Fastest: 1.6833333333333335e-06 minutes
   + Slowest: 0.00017383333333333332 minutes.



```
......Progress:4392/4397......
......Progress:4393/4397......
......Progress:4394/4397......
......Progress:4395/4397......
......Progress:4396/4397......
......Progress:4397/4397......
1.8e-06 0.0007213333333333333 9.619947312561566e-05
0.42345445000000004
```

5. **Astar**: This is an algorithms for a single pair
   Each one to another time: 3.4833333333333336e-06 (Better performance compare to Dijkstra or Esposo-Pape with the same point)

   It is hard to use A Star for one to all the other stops as the heuristic_estimate will not be efficient.



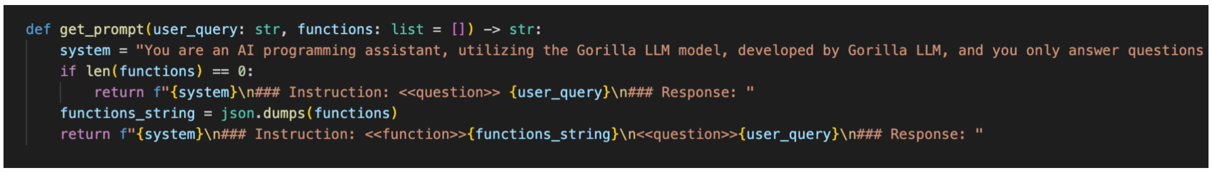
```
{'time': 0.09843038733032891, 'path': [1, 5, 4, 7, 9, 12, 6, 10], 'coordinates': [[106.65255737, 10.75023174], [106.65254211, 10.75026989], [106.65338898, 10.75043011],
 [106.65453339, 10.75069046], [106.65473175, 10.75078964], [106.65490723, 10.75080013], [106.65592957, 10.75074005], [106.65647888, 10.75080013], [106.6574173, 10.75070
953], [106.65840912, 10.75067997], [106.65899658, 10.75065994], [106.65908051, 10.75065994], [106.65914917, 10.75061989], [106.65920258, 10.75061989], [106.65926361, 10
.75065041], [106.65930939, 10.75070953], [106.65930939, 10.75078964], [106.65926361, 10.75086975], [106.65921021, 10.75088978], [106.65925598, 10.75105953], [106.659149
17, 10.75183868], [106.65893555, 10.75256634], [106.65893555, 10.75256634], [106.65893555, 10.75256634], [106.65888214, 10.75256062], [106.65884399, 10.75299358], [106.
65963745, 10.75300884], [106.66043091, 10.75302505], [106.66041565, 10.75296211], [106.66041565, 10.75296211], [106.66207886, 10.753088], [106.66207886, 10.753088], [10
6.66207886, 10.753088], [106.66287231, 10.75327778], [106.66348267, 10.75334072], [106.66348267, 10.75334072], [106.66348267, 10.75334072], [106.66414642, 10.75349903],
 [106.66490936, 10.75362587], [106.66490936, 10.75362587], [106.66490936, 10.75362587], [106.66590118, 10.75383091], [106.66688538, 10.75399971], [106.66688538, 10.7539
9971], [106.66688538, 10.75399971], [106.66777039, 10.75421047], [106.66857147, 10.75433731]]}
```

This is an example result from this algorithm.

Week 11: Using the Godrilla model to deal with function calling problems.

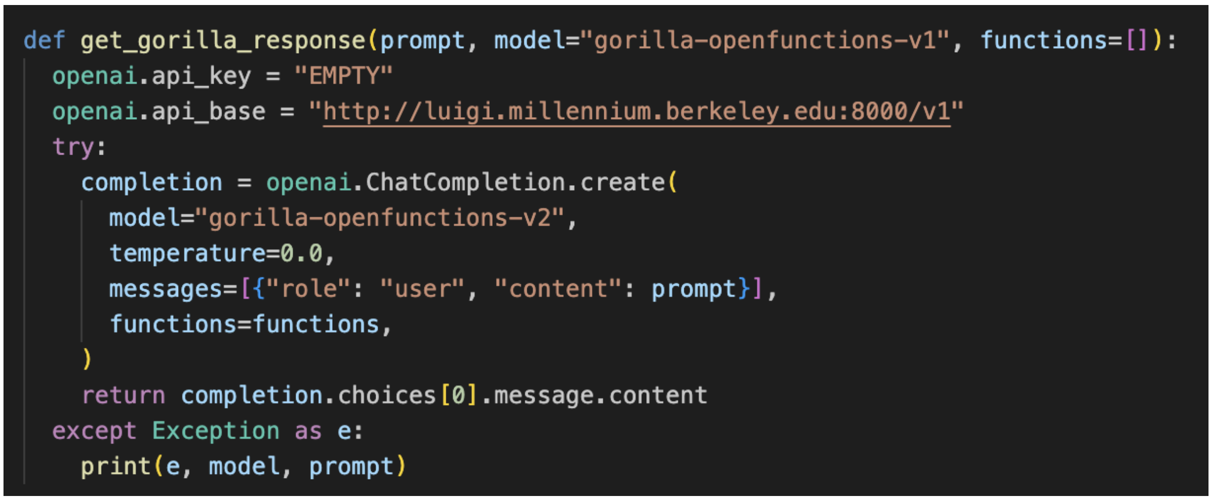
Using the godzilla open ai model for transforming the query to a prompt which is a mix of the system prompt and functions to create an instructed prompt to boost the performance of the model. Declaring the description and the api_call in the function map so that the model can understand and generate the function as we want.

This is the creating prompts function and gives the model more concepts so that it can realize their familiar formats of answering our query.



```python
def get_prompt(user_query: str, functions: list = []) -> str:
    system = "You are an AI programming assistant, utilizing the Gorilla LLM model, developed by Gorilla LLM, and you only answer questions
    if len(functions) == 0:
        return f"{system}\n### Instruction: <<question>> {user_query}\n### Response: "
    functions_string = json.dumps(functions)
    return f"{system}\n### Instruction: <<function>>{functions_string}\n<<question>>{user_query}\n### Response: "
```

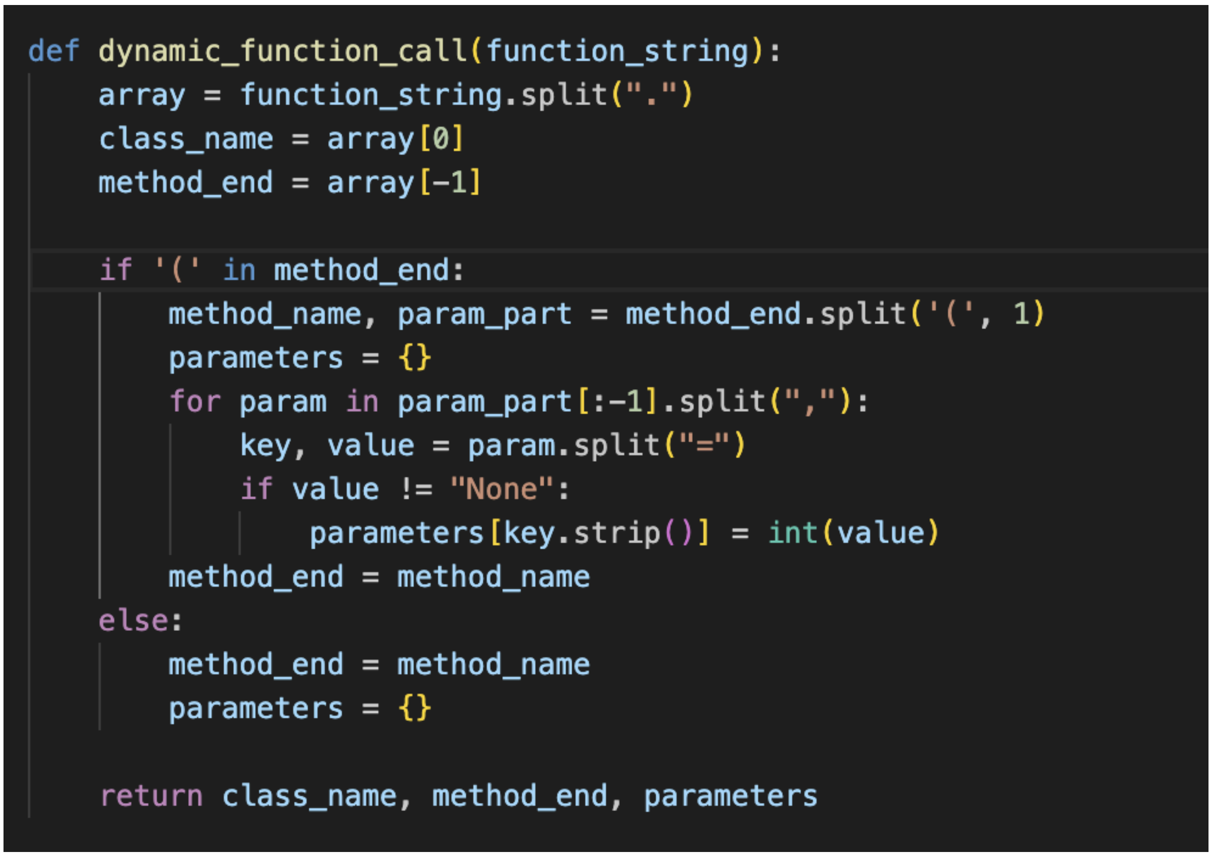
This is the function that uses the prompt to create the response from the query.



```python
def get_gorilla_response(prompt, model="gorilla-openfunctions-v1", functions=[]):
  openai.api_key = "EMPTY"
  openai.api_base = "http://luigi.millennium.berkeley.edu:8000/v1"
  try:
    completion = openai.ChatCompletion.create(
      model="gorilla-openfunctions-v2",
      temperature=0.0,
      messages=[{"role": "user", "content": prompt}],
      functions=functions,
    )
    return completion.choices[0].message.content
  except Exception as e:
    print(e, model, prompt)
```

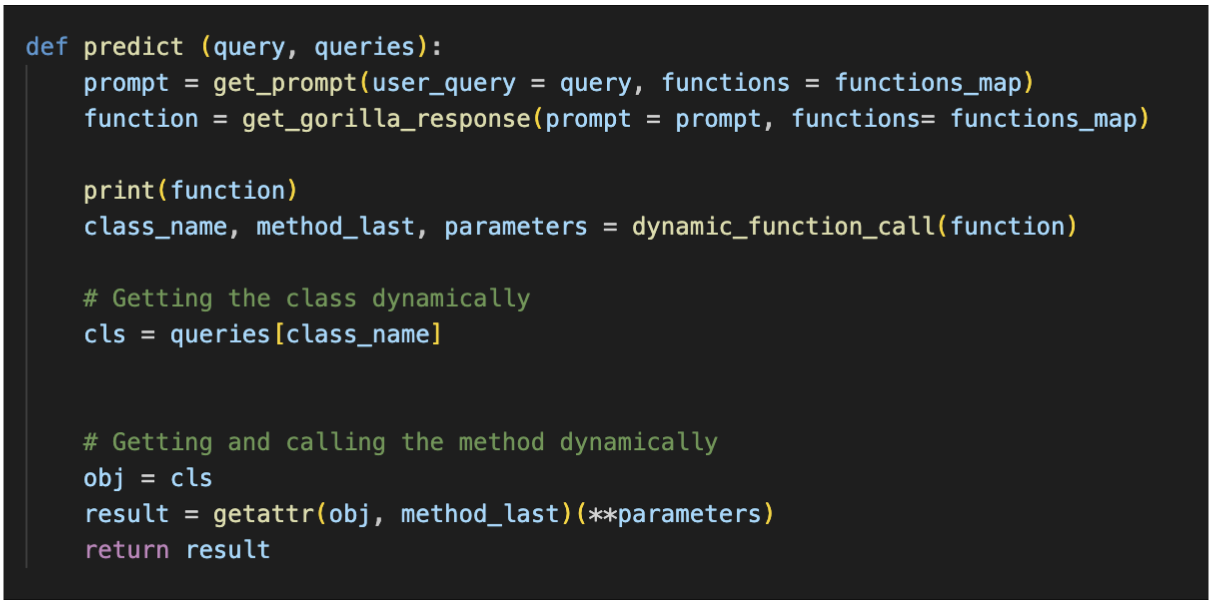
Next, process the output so that it can contain the class and the method of that particular class with the parameters. From there, I try to get the information from the query that I put into the function as a parameter and get the information out to give back to the customer.

This is the handling step from the response by splitting and stripping back to give us the information we need.



```python
def dynamic_function_call(function_string):
    array = function_string.split(".")
    class_name = array[0]
    method_end = array[-1]

    if '(' in method_end:
        method_name, param_part = method_end.split('(', 1)
        parameters = {}
        for param in param_part[:-1].split(","):
            key, value = param.split("=")
            if value != "None":
                parameters[key.strip()] = int(value)
        method_end = method_name
    else:
        method_end = method_name
        parameters = {}

    return class_name, method_end, parameters
```

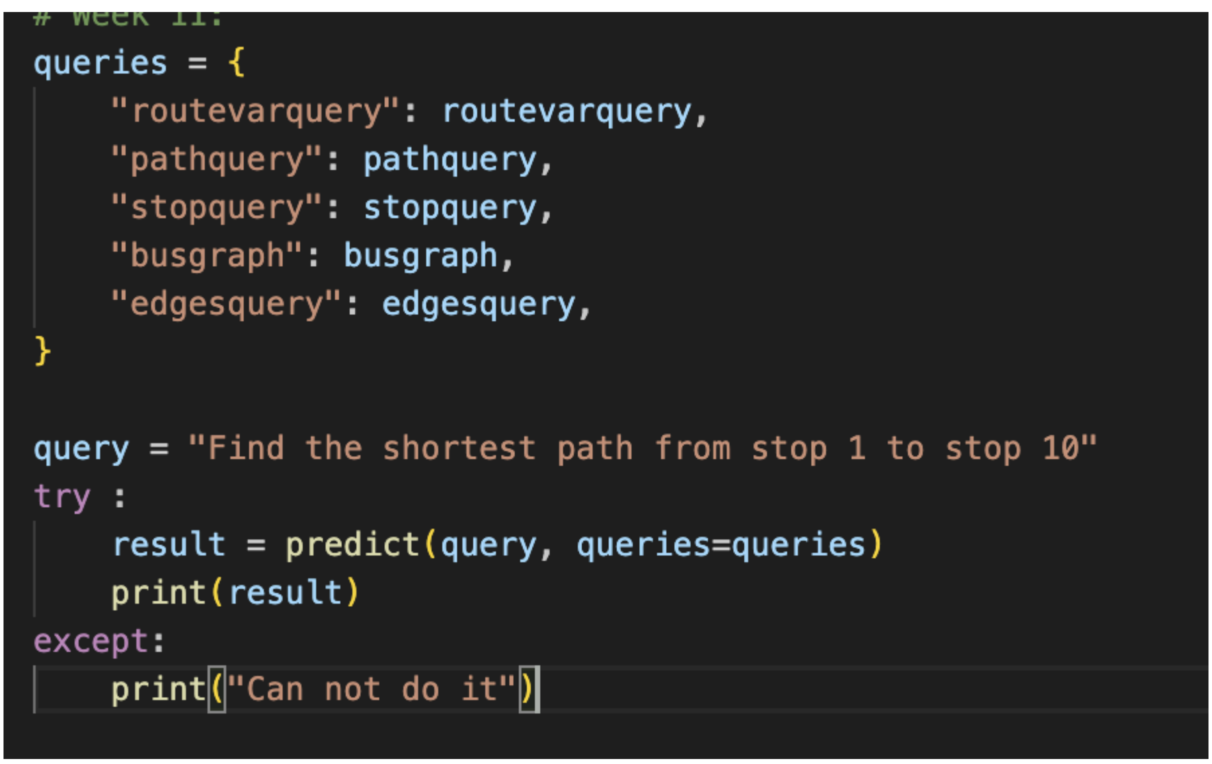
Finally this is the predicted steps and get the information from all the informations that is handle before that.



```python
def predict (query, queries):
    prompt = get_prompt(user_query = query, functions = functions_map)
    function = get_gorilla_response(prompt = prompt, functions= functions_map)

    print(function)
    class_name, method_last, parameters = dynamic_function_call(function)

    # Getting the class dynamically
    cls = queries[class_name]


    # Getting and calling the method dynamically
    obj = cls
    result = getattr(obj, method_last)(**parameters)
    return result
```

It's clearly that from a free model. The performance is mostly based on how the prompt is created and there might be some errors in generating the output text. However it is still a good approach for handling such problems.

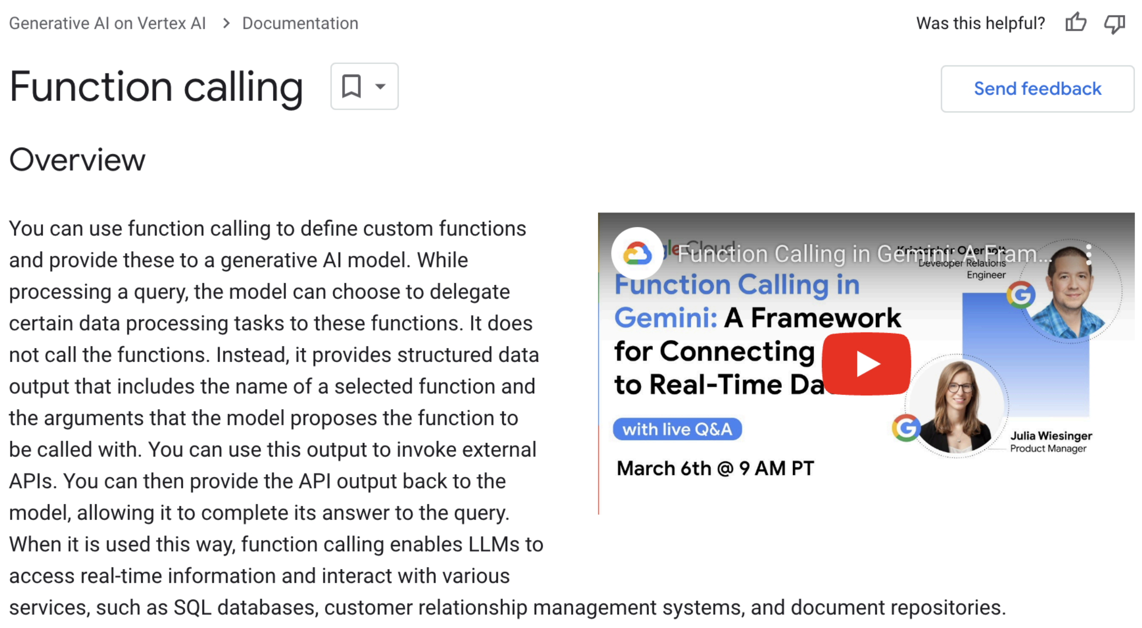
How to use this model.



```python
# week 11:
queries = {
    "routevarquery": routevarquery,
    "pathquery": pathquery,
    "stopquery": stopquery,
    "busgraph": busgraph,
    "edgesquery": edgesquery,
}

query = "Find the shortest path from stop 1 to stop 10"
try :
    result = predict(query, queries=queries)
    print(result)
except:
    print("Can not do it")
```

In addition, there are still a lot more approaches using LLM to call functions like using the Vertex AI from Google, or OpenAI function calling. However it is quite expensive to use, the progress to handle is a bit different and you should identify your account and billing methods as well as key api in order to use it.

From Google Cloud Platform (GCP)


Generative AI on Vertex AI > Documentation

Was this helpful?

Function calling

Send feedback

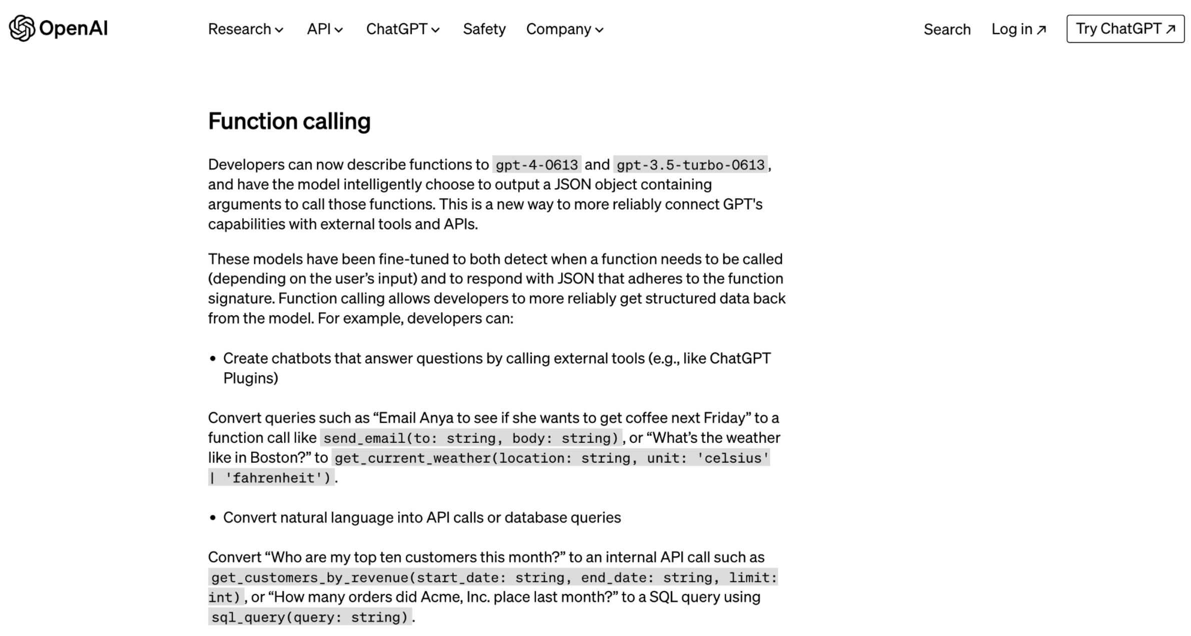
Overview

You can use function calling to define custom functions and provide these to a generative AI model. While processing a query, the model can choose to delegate certain data processing tasks to these functions. It does not call the functions. Instead, it provides structured data output that includes the name of a selected function and the arguments that the model proposes the function to be called with. You can use this output to invoke external APIs. You can then provide the API output back to the model, allowing it to complete its answer to the query. When it is used this way, function calling enables LLMs to access real-time information and interact with various services, such as SQL databases, customer relationship management systems, and document repositories.

Function Calling in Gemini: A Fram...
Function Calling in Gemini: A Framework for Connecting to Real-Time Da
Developer Relations Engineer
with live Q&A
March 6th @ 9 AM PT
Julia Wiesinger
Product Manager

This method uses the description and it can produce a more efficient way into a class without handling the data but to call it right away. However this method is expensive and I have to pay money to use it so I can not try it.

From OpenAI



Function calling

Developers can now describe functions to `gpt-4-0613` and `gpt-3.5-turbo-0613`, and have the model intelligently choose to output a JSON object containing arguments to call those functions. This is a new way to more reliably connect GPT's capabilities with external tools and APIs.

These models have been fine-tuned to both detect when a function needs to be called (depending on the user's input) and to respond with JSON that adheres to the function signature. Function calling allows developers to more reliably get structured data back from the model. For example, developers can:

- Create chatbots that answer questions by calling external tools (e.g., like ChatGPT Plugins)

Convert queries such as "Email Anya to see if she wants to get coffee next Friday" to a function call like `send_email(to: string, body: string)`, or "What's the weather like in Boston?" to `get_current_weather(location: string, unit: 'celsius' | 'fahrenheit')`.

- Convert natural language into API calls or database queries

Convert "Who are my top ten customers this month?" to an internal API call such as `get_customers_by_revenue(start_date: string, end_date: string, limit: int)`, or "How many orders did Acme, Inc. place last month?" to a SQL query using `sql_query(query: string)`.

I have tried to use OpenAI but it requires paying to use, and I have learnt how to use it through this video:
https://youtu.be/aqdWSYWC_LI?si=lN_aL rfJmVUwQAYY

The final solution I want to show is FnCTOD approach using LLama: (One of the newest approach for this field)

FnCTOD:
"A novel approach, FnCTOD, to address zero-shot DST with LLMs. Our method seamlessly integrates DST as a part of the assistant's output during chat completion. Specifically, we treat the schema of each task-oriented dialogue domain as a specific function, and DST for this domain as the process of ``calling" the corresponding function. We thus instruct LLMs to generate function calls along with the response in the assistant's output. To achieve this, we convert the domain schema into function specifications, which include the function's description and required arguments, and incorporate them into the system prompt of the LLM. Additionally, we integrate these function calls into the assistant's output within the dialogue context."
Github: https://github.com/facebookresearch/FnCTOD
Paper: https://arxiv.org/pdf/2402.10466 (16 Feb 2024)

Advancements in language models, particularly Large Language Models (LLMs), have catalyzed a paradigm shift in software development, offering a plethora of innovative avenues for optimizing function calls and enhancing overall performance. These models, endowed with vast linguistic knowledge and learning capabilities, are transforming how developers approach code optimization, ushering in an era of unprecedented efficiency and productivity. LLMs have made a profound impact in the realm of function inlining. Function inlining, the process of replacing a function call with its actual code body, can greatly reduce the overhead associated with function invocation. LLMs excel in identifying opportunities for inlining by scrutinizing code patterns and contextual cues, thereby enabling developers to streamline function calls and expedite program execution.