UNIVERSITY OF SCIENCE,

HO CHI MINH

DIJKSTRA' s APPLICATION IN BUSMAP AND AI MODELS

Using AI-Models to integrate function calling with efficiency and good practices
Handling users' queries by natural language in English
Handling users' queries with our documents

TECHNICAL REPORT
W11

submitted for the Solo Project in Semester 2 – First Year

IT DEPARTMENT

Computer Science

by

Phan Tuan Kiet

Full name: Phan Tuan Kiet

ID: 23125062

Class: 23APCS2

Tasks achieved: 01/01

Lecturer:
Professor Dinh Ba Tien (PhD)
Teaching Assistants:
Mr. Ho Tuan Thanh (MSc)
Mr. Nguyen Le Hoang Dung (MSc)

2024

Contents

I) Structure

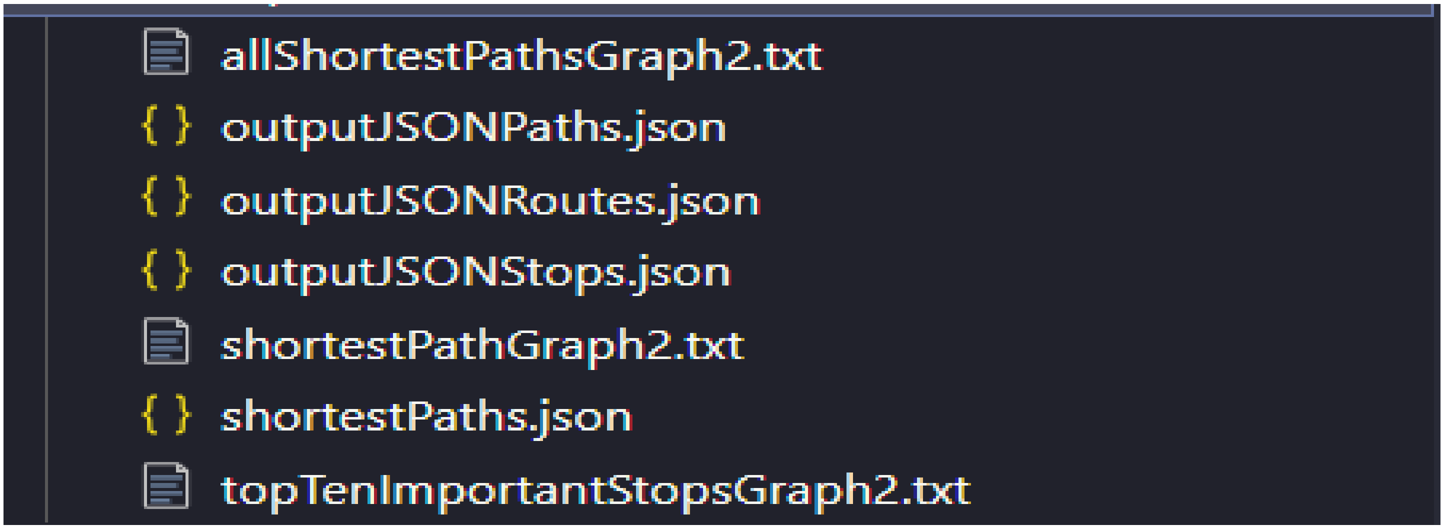
* Main source code – using Gorilla for function calls:

- JSON files: Containing JSON files for data handling.

- Output: The result of our AI – function calling:

+ Query using natural language with var, stop, and path.

+ Query using natural language with the graph (for example I want to find the shortest path from 3 to 7276).



* Notes that these files are created by AI calling to functions itself, not interpreted by the program's owner.

- Six file .py:

+ graph2.py: Our graph contains vertices and edges. Deeper details are covered in the previous report.

+ LLM.py: Contains our main model – which is Gorilla (our main topic today).

+ main.py: For user interface and handling queries from users.

+ path.py: Handling paths.json. Deeper details are covered in the previous report.

+ stop.py: Handling stops.json. Deeper details are covered in the previous report.

+ vars.py: Handling vars.json. Deeper details are covered in the previous report.

* AI_Model_LocalGPT_Method1 folder (second method – not in use but is mentioned to see the difference between an AI model trained for function calling and the one trained for reading our documents through the vector store and API key – Llama-2 model):

- There are a lot of files in this folder and the structure is quite complex, however, we just need to focus on three things:
+ SOURCE_DOCUMENTS: The file that we will put in here, for example, I put my technical report in W07.

+ ingest.py: We will need to run it so that our model (Llama 2) will read out documents and then handle our queries.

+ run_localGPT.py: We will run our program here using some terminal code. Overall progress will be running the ingest.py (so that Llama 2 will read our documents), then input the queries.

II) Methods and Comments

A) Comments

- We can see from the beginning that there are many large language models to choose from, however, not every language model is free. Therefore I was looking for the free ones that have enough ability, though not strong compared to the expensive ones.

- Our objective is to make the AI model understand our data (documents, program. etc) and then handle our queries and talk with users → find the model that is pre-trained to interact with our documents and data.

- For this initial thinking, I have found many models that are trained for this:

GPT-NeoX-20B	4.8/5	Free	20 billion parameters, versatile applications
GPT-J	4.9/5	Free	High performance, adaptable to various tasks
LLaMA 2	4.8/5	Free	Multimodal capabilities, collaboration of Meta AI and Microsoft
OPT-175B	4.8/5	Free	175 billion parameters, advanced language understanding
BLOOM	4.7/5	Free	Multilingual support, large-scale collaboration
Baichuan-13B	4.7/5	Free	Bilingual capabilities, 13 billion parameters
CodeGen	4.5/5	Free	Tailored for developers, supports multiple programming languages
BERT	4.2/5	Free	Contextual language understanding, wide application in NLP
T5	4.7/5	Free	Text-to-text framework, versatile for various NLP tasks
Falcon-40B	4.5/5	Free	40 billion parameters, cutting-edge language processing

- However, after running some of the models, I realized that these models are pre-trained, but for a huge amount of data over there (on the internet), so they can do a lot of things, but not really the thing that we want it to do:

+ For example: When a model is not fine-tuned to detect a function call through APIs, when we want it to call the function, it just returns a text. On the other hand, for the ones that are fine-tuned for function-calling like GPT-4 and GPT-3.5, they can detect when a function needs to be called and then output JSON containing arguments to call the function that we want.

+ To see the real difference in this argument, I will mention later on a difference between Llama2 and Gorilla.

- From here, we already know that we need a large-language model that is fine-tuned for function calling. But there is a problem, how can we make the model call the correct function related to our expectations?

→ We need to write function definition and api_call correspondingly with the correct function in our program.

+ But what if our function needs arguments

→ We also need to write a definition for each argument that relates to a specific function.

B) Methods

- As I mentioned above, we will focus on the pre-trained and fine-tuned model for function calling for better efficiency and convenience. Therefore, I will focus on the method of implementing the Gorilla model and suitably writing the function definition so that the model can easily realise which function to call.

- However, I will go through the method of implementing Llama 2 to see the picture of LLMs in a broader view.

1) Method 1 (Llama 2)

- Begin by cloning the source code repository from GitHub. Since this model is open-source, no costs are associated with this step.

- Establish a dedicated virtual environment to ensure the integrity of our program, mitigating potential conflicts that may arise from the installation of numerous packages.

- Proceed to set up the LocalGPT framework, configuring it to accommodate our specific requirements and data.

- Supply the necessary documents and data pertinent to the model.

- Load the provided data into the system and initiate the model execution.

- Finally, input queries into the model interface to obtain desired outputs.

2) Method 2 (Gorilla – Main method)

- The implementation of Gorilla is quite straightforward, the most complex thing is to write our function definitions so that the model can choose which function to call for, and then we will handle the response given by the model by catching the api_key to call our function, in details:

+ Set up the environment for Gorilla (open-source, no cost needed).

+ Set up the get_prompt function so that the model will return our question and function definitions in the correct form.

+ Get the response by the get_gorilla_response (the Gorilla model will handle this as it is fine-tuned for function-calling, we will just need to write good function definitions, and the model will then handle it).

+ Then, we will parse the response so that we can catch the api_key and then set it up so that the program will call our desired function.

+ Lastly, we just need to input the queries and see our result!

* Notes:

+ We will need to load our graph first when opening the program so that the model can read the data

+ Sometimes the model will call the wrong function. In this case, we need to input a more detailed query:

- For example:

+ Instead of: Give me the stop ID 5 → List out all the stops that have stop ID 5.

+ You can also input a query like this: Today I have a really important job, give me the shortest path from stop 4 to stop 7276.

* Another thing to notice is that I will use the "gorilla-open functions-v1" as gorilla-open functions-v1 is our advanced model that takes in the user's prompt along with MULTIPLE API calls and returns the function with the right arguments.

III) Noticable performance of the Gorilla model

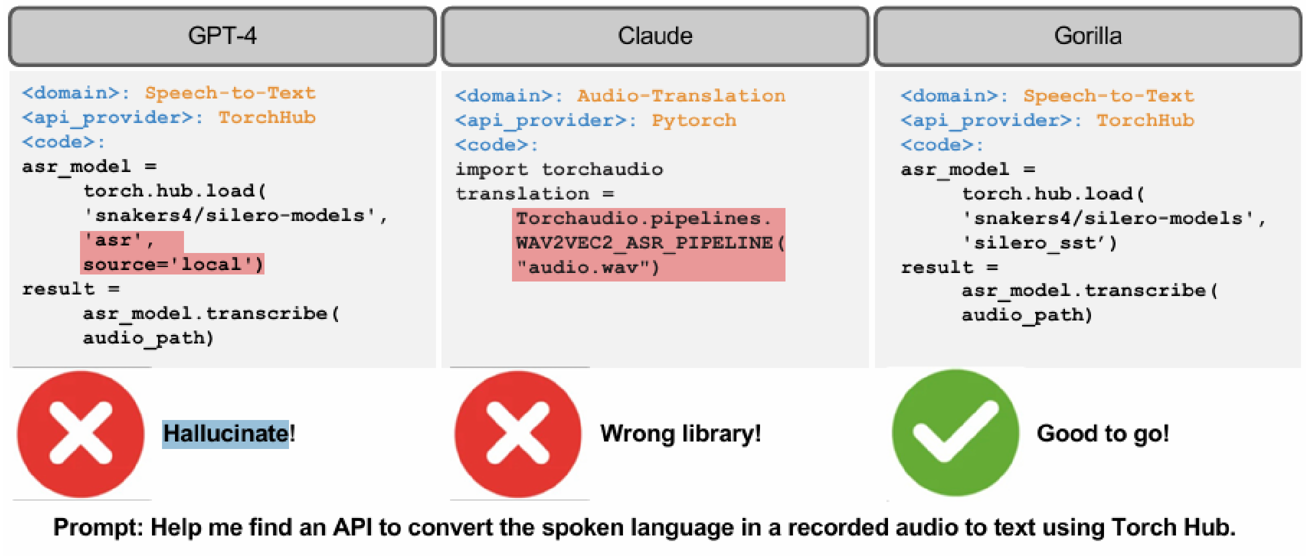


Figure 1: **Examples of API calls**. Example API calls generated by GPT-4 [29], Claude [3], and Gorilla for the given prompt. In this example, GPT-4 presents a model that doesn't exist, and Claude picks an incorrect library. In contrast, our Gorilla model can identify the task correctly and suggest a fully-qualified API call.

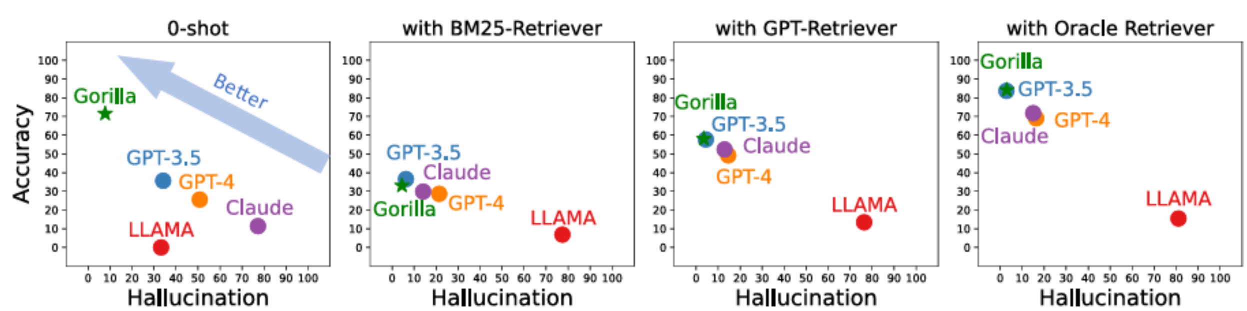
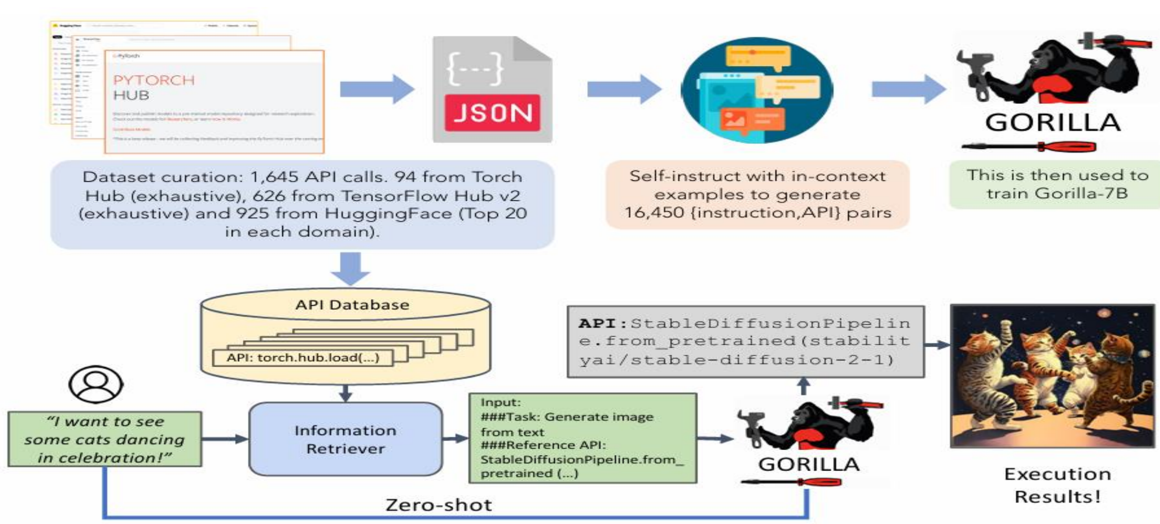


Figure 2: **Accuracy (vs) hallucination** in four settings, that is, *zero-shot* (i.e., without any retriever), and *with retrievers*. BM25 and GPT are commonly used retrievers and the oracle retriever returns relevant documents at 100%, indicating an upper bound. Higher in the graph (higher accuracy) and to the left is better (lower hallucination). Across the entire dataset, our model, Gorilla, improves accuracy while reducing hallucination.

- How is a Gorilla made?



- How APIs are called?

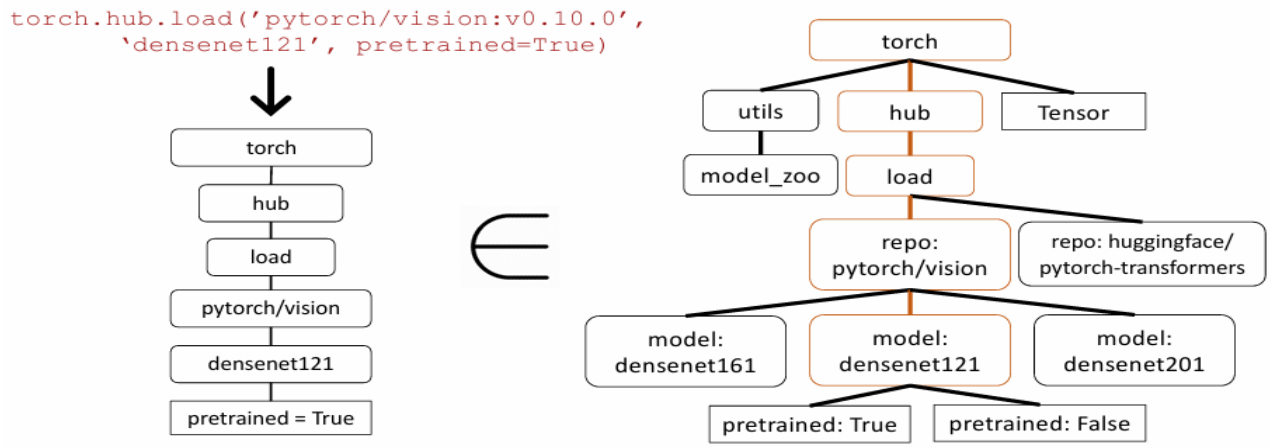


Figure 4: **AST Sub-Tree Matching to evaluate API calls.** On the left is an API call returned by Gorilla. We first build the associated API tree. We then compare this to our dataset, to see if the API dataset has a subtree match. In the above example, the matching subtree is highlighted in brown, signifying that the API call is indeed correct. `Pretrained=True` is an optional argument.

- AST Accuracy on API call

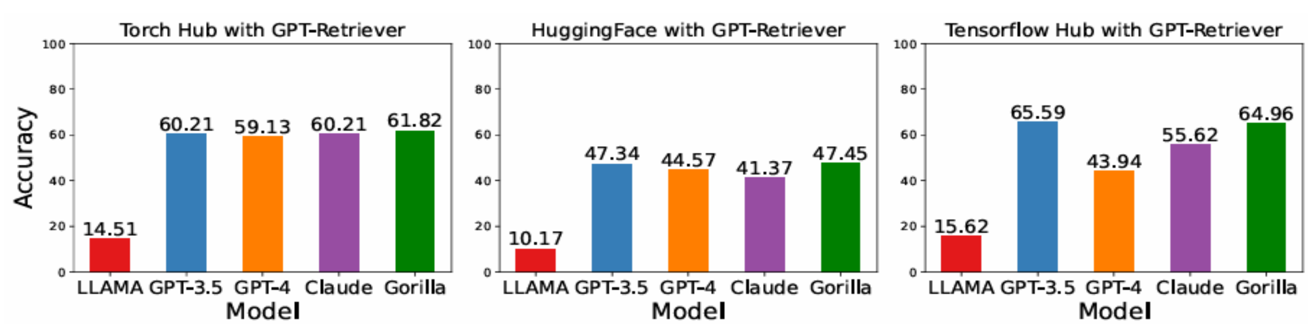


Figure 5: **Accuracy with GPT-retriever.** Gorilla outperforms on Torch Hub and Hugging-Face while matching performance on Tensorflow Hub for all existing SoTA LLMs - closed source, and open source.

- Comparison to other LLMs on Torch Hub, HuggingFace, and Tensorflow Hub APIs

+ Finetuning without Retrieval

Table 1: **Evaluating LLMs on Torch Hub, HuggingFace, and Tensorflow Hub APIs**

LLM (retriever)	TorchHub			HuggingFace			TensorFlow Hub		
	overall ↑	hallu ↓	err ↓	overall ↑	hallu ↓	err ↓	overall ↑	hallu ↓	err ↓
LLAMA (0-shot)	0	100	0	0.00	97.57	2.43	0	100	0
GPT-3.5 (0-shot)	48.38	18.81	32.79	16.81	35.73	47.46	41.75	47.88	10.36
GPT-4 (0-shot)	38.70	36.55	24.7	19.80	37.16	43.03	18.20	78.65	3.13
Claude (0-shot)	18.81	65.59	15.59	6.19	77.65	16.15	9.19	88.46	2.33
Gorilla (0-shot)	**59.13**	**6.98**	33.87	**71.68**	**10.95**	17.36	**83.79**	**5.40**	10.80
LLAMA (BM-25)	8.60	76.88	14.51	3.00	77.99	19.02	8.90	77.37	13.72
GPT-3.5 (BM-25)	38.17	6.98	54.83	**17.26**	8.30	74.44	**54.16**	3.64	42.18
GPT-4 (BM-25)	35.48	11.29	53.22	16.48	15.93	67.59	34.01	37.08	28.90
Claude (BM-25)	39.78	5.37	54.83	14.60	15.82	69.58	35.18	21.16	43.64
Gorilla (BM-25)	**40.32**	**4.30**	55.37	17.03	**6.42**	76.55	41.89	**2.77**	55.32
LLAMA (GPT-Index)	14.51	75.8	9.67	10.18	75.66	14.20	15.62	77.66	6.71
GPT-3.5 (GPT-Index)	60.21	1.61	38.17	29.08	7.85	44.80	**65.59**	3.79	30.50
GPT-4 (GPT-Index)	59.13	1.07	39.78	44.58	11.18	44.25	43.94	31.53	24.52
Claude (GPT-Index)	60.21	3.76	36.02	41.37	18.81	39.82	55.62	16.20	28.17
Gorilla (GPT-Index)	**61.82**	**0**	38.17	**47.46**	**8.19**	44.36	64.96	**2.33**	32.70
LLAMA (Oracle)	16.12	79.03	4.83	17.70	77.10	5.20	12.55	87.00	0.43
GPT-3.5 (Oracle)	66.31	1.60	32.08	89.71	6.64	3.65	**95.03**	**0.29**	4.67
GPT-4 (Oracle)	66.12	0.53	33.33	85.07	10.62	4.31	55.91	37.95	6.13
Claude (Oracle)	63.44	3.76	32.79	77.21	19.58	3.21	74.74	21.60	3.64
Gorilla (Oracle)	**67.20**	**0**	32.79	**91.26**	**7.08**	1.66	94.16	1.89	3.94

→ In Table 1, the graph shows that a lightly fine-tuned Gorilla achieves state-of-the-art performance in zero-shot overall modelling, surpassing other models by 20.43% over GPT-4 and 10.75% over ChatGPT. When contrasted with other open-source models like LLAMA, the improvement reaches as high as 83%. These findings suggest quantitatively that fine-tuning outperforms retrieval, at least within the confines of our study.

+ Finetuning with Retrieval

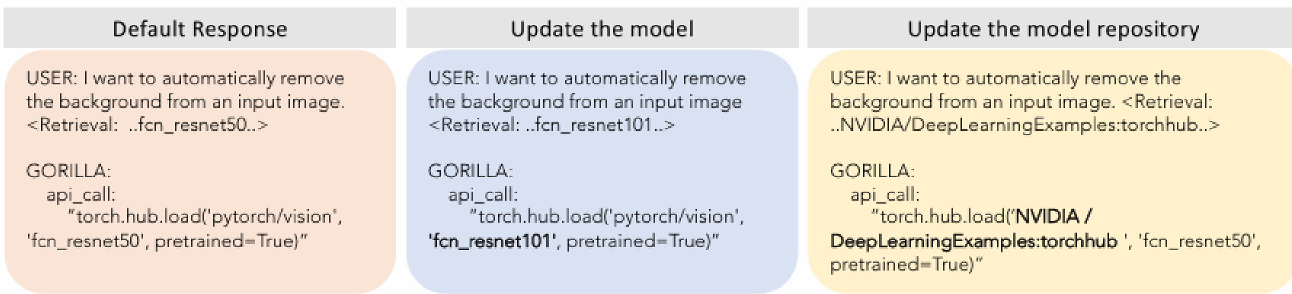


Figure 6: **Gorilla's retriever–aware training enables it to react to changes in the APIs.** The second column demonstrates changes in model upgrading FCN's ResNet–50 backbone to ResNet–101. The third column demonstrate changes in model registry from `pytorch/vision` to `NVIDIA/DeepLearningExamples:torchhub`

Table 2: **Comparison of retrieval techniques**

	Gorilla without Retriever				Gorilla with Oracle retriever			
	zero-shot	BM25	GPT-Index	Oracle	zero-shot	BM25	GPT-Index	Oracle
Torch Hub (overall) ↑	59.13	37.63	60.21	54.83	0	40.32	61.82	67.20
HuggingFace (overall) ↑	71.68	11.28	28.10	45.58	0	17.04	47.46	91.26
TensorHub (overall) ↑	83.79	34.30	52.40	82.91	0	41.89	64.96	94.16
Torch Hub (Hallu) ↓	6.98	11.29	4.30	15.59	100	4.30	0	0
HuggingFace (Hallu) ↓	10.95	46.46	41.48	52.77	99.67	6.42	8.19	7.08
TensorHub (Hallu) ↓	5.40	20.43	19.70	13.28	100	2.77	2.33	1.89

+ What if we have constraints on our APIs?

→ Still good performance

Table 3: **Evaluating LLMs on constraint-aware API invocations**

	GPT-3.5				GPT-4				Gorilla			
	0-shot	BM25	GPT-Index	Oracle	0-shot	BM25	GPT-Index	Oracle	0-shot	BM25	GPT-Index	Oracle
Torch Hub (overall)	**73.94**	62.67	81.69	80.98	62.67	56.33	71.11	69.01	71.83	57.04	71.83	78.16
Torch Hub (Hallu)	19.01	30.98	14.78	14.08	**15.49**	27.46	**14.08**	**9.15**	19.71	39.43	26.05	16.90
Torch Hub (err)	7.04	6.33	3.52	4.92	21.83	16.19	14.78	21.83	8.45	3.52	2.11	4.92
Accuracy const	43.66	**33.80**	**33.09**	69.01	43.66	29.57	29.57	59.15	**47.88**	30.28	26.76	67.60

	LLAMA				Claude			
	0-shot	BM25	GPT-Index	Oracle	0-shot	BM25	GPT-Index	Oracle
Torch Hub (overall)	0	8.45	11.97	19.71	29.92	**81.69**	**82.39**	**81.69**
Torch Hub (Hallu)	100	91.54	88.02	78.87	67.25	**16.19**	15.49	13.38
Torch Hub (err)	0	0	0	1.4	2.81	2.11	2.11	4.92
Accuracy const	0	6.33	3.52	17.60	17.25	29.57	31.69	**69.71**

→ Result: For Table 3, we filtered a subset of the TorchHub dataset that had accuracy defined for at least one dataset in its model card (representing 65.26% of the TorchHub dataset in Table 1). It is noticeable that with constraints, understandably, the accuracy drops across all models, both with and without a retriever. Gorilla can match the performance of the best-performing model, GPT-3.5 when using retrievals (BM25, GPT Index), and it exhibits the highest accuracy in the zero-shot case. This underscores Gorilla's ability to navigate APIs while considering the trade-offs between different constraints.

IV) Visualization of finetuned LLaMA-based model

- While Gorilla diverges from the GPT model architecture, its operational methodology closely parallels GPT models developed by OpenAI, employing function calls. Nonetheless, empirical evidence indicates Gorilla outperforms GPT in select domains, as demonstrated by the data above.

- The operational framework of Gorilla, despite its departure from the GPT model, remains rooted in the paradigm of a large language model. Its primary function involves processing human-generated data and delivering proficient responses via function calls. Conceptually, Gorilla can be likened to a cognitive entity with which interactions yield informative outputs.

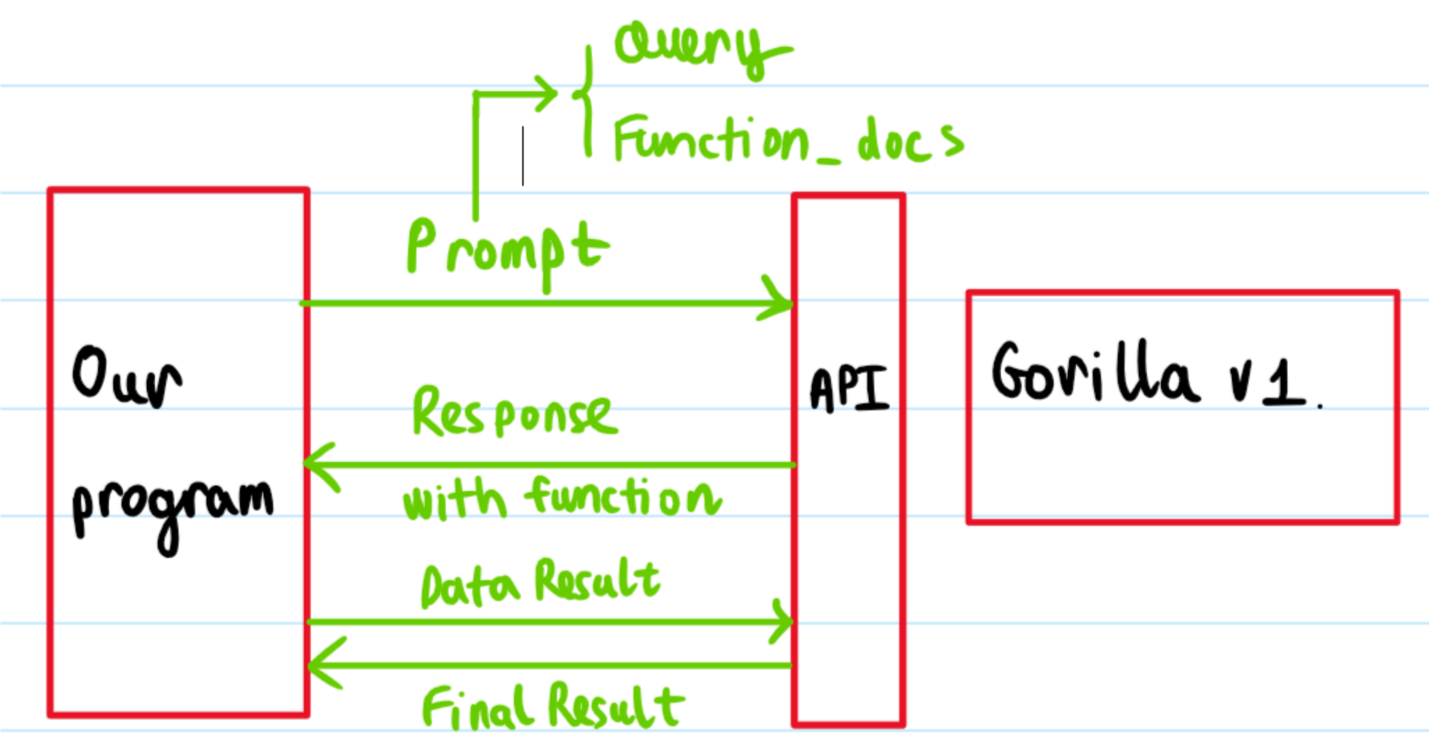
- Nevertheless, the inherent limitations of Large Language Models (LLMs) manifest when tasked with nuanced and precise operations. While LLMs excel at data comprehension and retrieval, executing sophisticated functions necessitates tailored mechanisms. Thus, a model finely tuned for function calling becomes imperative.

- The pivotal challenge arises in orchestrating the interface between LLMs and the requisite function calls. This conundrum underscores the necessity for an 'api_call' mechanism complemented by a comprehensive catalogue of functions.

The workflow:

- When confronted with intricate tasks necessitating functional execution, human operators prompt Large Language Models (LLMs) to act. In the case of Gorilla, for instance, the model's decision-making process hinges upon the provided function documentation to discern the appropriate function invocation. Subsequently, Gorilla furnishes the response accompanied by suitable parameters. Conversely, in the context of GPT, the responsibility falls upon the human operator to invoke the function manually, furnish the data to the LLM, and subsequently enable autonomous function invocation by the LLM in subsequent iterations. Notably, Gorilla streamlines this process by facilitating direct engagement in response handling and subsequent function invocation.

- Visualization:



V) Detailed explanation in the source code

```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-v1",
            temperature=0.0,
            messages=[{"role": "user", "content": prompt}],
            functions=functions,
        )
        response = completion.choices[0].message.content
        return response

    except Exception as e:
        print("Error:", e, model, prompt)
```

*** Notes:** The hardest part is to understand how LLMs which are finetuned for function calling work, then find a suitable model for this purpose. Then the implementation is quite straightforward with writing API and function_docs. Lastly, we will handle the response given by the model

- First, we need to set up the Gorilla model to get the response. We will use gorilla-v1 for parallel function calling, then we will return the response after being defined by Gorilla.

- Then, we will set up the get_prompt function to handle our query for the best result, note that we will have to make our function into to list of JSON strings so that gorilla will handle the prompt easily.

```python
def get_prompt(user_query: str, functions: list = []) -> str:
    if len(functions) == 0:
        return f"USER: <<question>> {user_query}\nASSISTANT: There are no function definitions to call so nothing is done"
    functions_string = json.dumps(functions)
    return f"USER: <<question>> {user_query} <<function>> {functions_string}\nASSISTANT: I am an AI model and I will help you call
```

- In the next step, we will write our function_doc, I will list out one of those for easier explanation:

```json
{
    "name" : "Shortest Path",
    "api_call": "shortest_path",
    "description": "The shortest path between two locations",
    "parameters": [
        {
            "name": "start",
            "description": "Starting location"
        },
        {
            "name": "end",
            "description": "Ending location"
        },
    ]
},
```

+ This is the function call for the shortest path from one stop to another stop. Notice that the sign for the model to call this function is in the “description”. It will understand that this function is used when we need to find the shortest path from one stop to another.

+ For the parameters, we will also need to write the description for each parameter. For this function call, we will need two parameters, one is the starting location and the other one is the ending location.

- Another example of a function call is for the stop query (Example: List out all the stops that have ID x):

```
{
    "name" : "Stop Query",
    "api_call": "stop_query",
    "description": "List out all the stops that have information given by user",
    "parameters": [
        {
            "name": "StopId",
            "description": "The stop_id of the stops"
        },
        {
            "name": "Code",
            "description": "The code of the stops"
        },
        {
            "name": "Name",
            "description": "The name of the stops"
        },
```

+ In this function call, we will need to write the description with more details. As many parameters are required, we must write definitions for them all. However, when we give the query, we can write down any parameter we want.

- The last step is to parse the response given by our model:

+ The response will have the form of api_call(parameters) → We will parse it to take the api_call and parameters, then invoke the function correspondingly.

```python
def parse_response(response: str) -> tuple:
    start_idx = response.find('(')
    end_idx = response.rfind(')')

    if start_idx == -1 or end_idx == -1 or start_idx >= end_idx:
        raise ValueError("Invalid response format")

    api_function = response[:start_idx].strip()

    attributes_str = response[start_idx + 1:end_idx].strip()

    # Splitting attributes_str while considering quotes
    import shlex
    lexer = shlex.shlex(attributes_str, posix=True)
    lexer.whitespace += ','
    lexer.whitespace_split = True
    attributes_list = list(lexer)

    attributes = {}
    for pair in attributes_list:
        pair = pair.strip()
        if '=' in pair:
            key, value = pair.split('=', 1)
            key = key.strip()
            # Check for quoted value and remove quotes
            value = value.strip()
            if value.startswith('"') and value.endswith('"'):
                value = value[1:-1]  # Remove surrounding quotes
            if value != "": attributes[key] = value.lower()
        else:
            raise ValueError(f"Invalid attribute format: {pair}")

    return api_function, attributes
```

→ At the end of the function, we will have the correct function call with corresponding parameters.

VI) User interface – Query handling

- Notes that we will let users choose what area they want to query or ask Gorilla to do. This initial choice will lead to better performance at Gorilla will not have to go through everything, but just the functions in one area.

- The overall view:

```
while True:
    print("--------------------------------AI-MODEL--------------------------------")
    print("1. Query about stops")
    print("2. Query about route_var")
    print("3. Query about paths")
    print("4. Query about graph (All shortest paths - Top 10 stops - Shortest path)")
    print("5. Exit")

    choice = int(input("Enter your choice: "))
```

- Handle queries for stops:

```
if choice == 1:
    stop = StopQuery("stops.json")
    query = input("Enter your query: ")
    for x in function_doc:
        if x["name"] == "Stop Query":
            function_type = x
            break
    prompt = get_prompt(query, functions=[function_type])
    response = get_gorilla_response(prompt)
    print(response)
    function_call, attributes = parse_response(response)
    print(attributes)
    result = stop.searchByABC(**attributes)
    stop.outputAsJSON(result)
```

- As I told above, we will let the model choose a specific area of functions → enhance accuracy.

- Then we will get the prompt by putting our query and list of functions into the get_prompt function. After we get the function_call and attributes from the parse_response, we will call our desired function.

```
1. Query about stops
2. Query about route_var
3. Query about paths
4. Query about graph (All shortest paths - Top 10 stops - Shortest path)
5. Exit
Enter your choice: 1
Enter your query: List out all stops have id = 45
stop_query(StopId=45)
{'StopId': '45'}
JSON file created successfully.
```

- Similar job for RouteVarQuery, PathQuery and the graph of stops that we have built in the previous weeks.

VII) Result of Gorilla (main model)

- For StopQuery:

```
1. Query about stops
2. Query about route_var
3. Query about paths
4. Query about graph (All shortest paths - Top 10 stops - Shortest path)
5. Exit
Enter your choice: 1
Enter your query: list out all stops have id = 46
stop_query(StopId=46)
{'StopId': '46'}
JSON file created successfully.
```

output > {} outputJSONStops.json > ...


```
{"_StopId": 46, "_Code": "Q1 063", "_Name": "Nhà thờ Tân Định", "_StopType": "Trụ dừng", "_Zone": "Quận 1", "_Ward": null, "_AddressNo": "298"
{"_StopId": 46, "_Code": "Q1 063", "_Name": "Nhà thờ Tân Định", "_StopType": "Trụ dừng", "_Zone": "Quận 1", "_Ward": null, "_AddressNo": "298"
{"_StopId": 46, "_Code": "Q1 063", "_Name": "Nhà thờ Tân Định", "_StopType": "Trụ dừng", "_Zone": "Quận 1", "_Ward": null, "_AddressNo": "298"
```


- For RouteVarQuery:

```
1. Query about stops
2. Query about route_var
3. Query about paths
4. Query about graph (All shortest paths - Top 10 stops - Shortest path)
5. Exit
Enter your choice: 2
Enter your query: List out all routes have RouteVarName = Lượt về: Bến xe Hậu Nghĩa - Bến xe An Sương
route_var_query(RouteVarName="Lượt về: Bến xe Hậu Nghĩa - Bến xe An Sương")
JSON file created successfully.
```

output > {} outputJSONRoutes.json > ...


```
{"_RouteId": 97, "_RouteVarId": 2, "_RouteVarName": "Lượt về: Bến xe Hậu Nghĩa - Bến xe An Sương", "_RouteVarShortName": "Bến xe An Sương", "
```


- For PathQuery:

```
1. Query about stops
2. Query about route_var
3. Query about paths
4. Query about graph (All shortest paths - Top 10 stops - Shortest path)
5. Exit
Enter your choice: 3
Enter your query: Find all paths have route id = 4
path_query(RouteId=4)
JSON file created successfully.
```

output > {} outputJSONPaths.json > ...


```
{"_lat": [10.76767635, 10.76767635, 10.76767635, 10.7672224, 10.76747322, 10.76772308, 10.76766968, 10.76766968, 10.76766968, 10.76766968, 10.
{"_lat": [10.8437376, 10.84418106, 10.84297943, 10.84253693, 10.84233665, 10.84031296, 10.83889103, 10.83889103, 10.83889103, 10.83889103, 10.
```

- For GraphQuery:

+ Shortest path

```
Enter your choice: 4
Loading graph...
100%|                                                                  | 10243/10243 [00:00<00:00, 9979618.09it/s]
100%|                                                                  | 4397/4397 [00:15<00:00, 288.06it/s]
100%|                                                                  | 297/297 [00:02<00:00, 120.34it/s]
Enter your query (enter exit to Exit): Find the shortest path from stop 1 to stop 7276
shortest_path(start="1", end="7276")
Find shortest path and save successfully.
GeoJSON file created successfully.
```


```
{
    "type": "FeatureCollection",
    "features": [
    {
      "type": "Feature",
      "properties": {},
      "geometry": {
        "coordinates": [[106.65255737,10.75023174],[106.65255737,10.75023174],[106.65255737,10.75023174],[106.65255737,10.75023174],[106.6525
        "type": "LineString"
      }
    }
            ]
}
```


+ All shortest pairs:

```
Enter your query (enter exit to Exit): Give me all shortest paths in the graph
all_shortest_paths(type="ALL")
100%|                                                  | 4397/4397 [00:34<00:00, 127.16it/s]
100%|                                                  | 4397/4397 [00:24<00:00, 180.34it/s]
Save file successfully
```

```
LLM.py    allShortestPathsGraph2.txt ×
output > allShortestPathsGraph2.txt
Start from 1: [1->1:0s; 1->2:46.49182308420526s; 1->3:141.8110834347271s; 1->4:205.8843123936143s; 1->5:153.7689984516102s; 1->6:346.48537232
Start from 2: [2->1:626.1555416358026s; 2->2:0s; 2->3:95.31926035052183s; 2->4:170.34052371000416s; 2->5:118.22520976800007s; 2->6:310.941583
Start from 3: [3->1:530.8362812852807s; 3->2:559.4915712446058s; 3->3:0s; 3->4:75.02126335948233s; 3->5:22.90594941747824s; 3->6:215.62232328
Start from 4: [4->1:799.9674579200771s; 4->2:828.6227478794021s; 4->3:923.942008229924s; 4->4:0s; 4->5:946.847957647402 3s; 4->6:140.601059927
Start from 5: [5->1:507.9303318678026s; 5->2:536.5856218271276s; 5->3:631.9048821776495s; 5->4:52.11531394200408 6s; 5->5:0s; 5->6:192.7163738
Start from 6: [6->1:659.366397992367s; 6->2:688.021687951692s; 6->3:783.3409483022139s; 6->4:858.3622116616962s; 6->5:806.2468977196921s; 6->
Start from 7: [7->1:764.2517532106173s; 7->2:792.9070431699423s; 7->3:888.2263035204642s; 7->4:963.2475668799465s; 7->5:911.1322529379424s; 7
Start from 8: [8->1:103.50771766801346s; 8->2:132.1630076273385s; 8->3:227.48226797786032s; 8->4:302.5035313373426s; 8->5:250.38821739533856s
Start from 9: [9->1:733.6335215589177s; 9->2:762.2888115182427s; 9->3:857.6080718687646s; 9->4:932.6293352282469s; 9->5:880.514021286242 8s; 9
Start from 10: [10->1:622.504705640818s; 10->2:651.159995600143s; 10->3:746.4792559506649s; 10->4:821.5005193101472s; 10->5:769.385205368143 1
Start from 11: [11->1:580.9188254807657s; 11->2:609.5741154400907s; 11->3:704.8933757906126s; 11->4:779.9146391500949s; 11->5:727.79932520809
Start from 12: [12->1:702.4493610501335s; 12->2:731.1046510094585s; 12->3:826.4239113599804s; 12->4:901.4451747194627s; 12->5:849.32986077745
Start from 13: [13->1:1731.0961264846817s; 13->2:1759.7514164440067s; 13->3:1855.0706767945285s; 13->4:1930.091940154011s; 13->5:1877.9766262
Start from 14: [14->1:540.4490158781844s; 14->2:569.1043058375094s; 14->3:664.4235661880313s; 14->4:739.4448295475136s; 14->5:687.32951560550
Start from 15: [15->1:1668.1109570899469s; 15->2:1696.766247049272s; 15->3:1792.0855073997936s; 15->4:1867.10677075927 6s; 15->5:1814.99145681
Start from 16: [16->1:1874.9016067659625s; 16->2:1903.5568967252875s; 16->3:1998.8761570758093s; 16->4:2073.8974204352917s; 16->5:2021.782106
Start from 17: [17->1:1811.6697328358482s; 17->2:1840.3250227951733s; 17->3:1935.644283145695s; 17->4:2010.6655465051774s; 17->5:1958.5502325
Start from 18: [18->1:1461.9954174279162s; 18->2:1490.6507073872413s; 18->3:1585.969967737763s; 18->4:1660.9912310972454s; 18->5:1608.8759171
Start from 19: [19->1:1620.5001771946663s; 19->2:1649.1554671539914s; 19->3:1744.474727504513s; 19->4:1819.4959908639955s; 19->5:1767.3806769
```

+ Top ten most important stops:

```
Enter your query (enter exit to Exit): what are top 10 most important stops?
top_ten_stops(number=10)
100%|                                              | 4397/4397 [00:00<00:00, 216319.92it/s]
100%|                                              | 4397/4397 [00:24<00:00, 182.52it/s]
100%|                                              | 4397/4397 [00:04<00:00, 1050.96it/s]
100%|                                              | 10/10 [00:00<00:00, 21.09it/s]
Top 10 important stops saved successfully.
```

```
Top 10 important stops in the graph:
1. StopID: 1239; Important: 2604239; Lat: 10.845187; Lng: 106.613522; Name: Bến xe An Sương; Code: HHM 058; StopType: Trụ dừng; Zone: Huyện
2. StopID: 1115; Important: 2594237; Lat: 10.845803; Lng: 106.613517; Name: Bến xe An Sương; Code: Q12 122; StopType: Trụ dừng; Zone: Quận
3. StopID: 1393; Important: 2590541; Lat: 10.853595; Lng: 106.608152; Name: Ngã tư Trung Chánh; Code: HHM 056; StopType: Nhà chờ; Zone: Huy
4. StopID: 1152; Important: 2536389; Lat: 10.85466; Lng: 106.607863; Name: Trung tâm Văn hóa Quận 12; Code: Q12 127; StopType: Nhà chờ; Zon
5. StopID: 510; Important: 2243745; Lat: 10.79472; Lng: 106.655198; Name: Bệnh viện Quận Tân Bình; Code: QTB 034; StopType: Nhà chờ; Zone:
6. StopID: 271; Important: 2225966; Lat: 10.822452; Lng: 106.630254; Name: Khu Công Nghiệp Tân Bình; Code: QTB 079; StopType: Nhà chờ; Zone
7. StopID: 174; Important: 2079854; Lat: 10.822315; Lng: 106.629825; Name: Trạm Dệt Thành Công; Code: QTP 010; StopType: Nhà chờ; Zone: Quậ
8. StopID: 272; Important: 2074526; Lat: 10.826957; Lng: 106.625828; Name: Cầu Tham Lương - Siêu thị Thiên Hòa; Code: Q12 128; StopType: Nh
9. StopID: 1234; Important: 2053891; Lat: 10.860602; Lng: 106.603324; Name: Ngã 3 Củ Cải; Code: HHM 053; StopType: Nhà chờ; Zone: Huyện Hóc
10. StopID: 1235; Important: 2053887; Lat: 10.866919; Lng: 106.598904; Name: Trường học Tân Xuân; Code: HHM 051; StopType: Trụ dừng; Zone:
```

- Other queries:

```
Enter your choice: 2
Enter your query: give me all the routes that have RouteNo = 04
route_var_query(RouteNo="04")
JSON file created successfully.
```


```
output > {} outputJSONRoutes.json > ...
;", "_RouteVarShortName": "Bến Xe An Sương", "_RouteNo": "04", "_StartStop": "Bến xe buýt Sài Gòn", "_EndStop": "Bến xe An Sương", "_Distance"
}", "_RouteVarShortName": "Bến Thành", "_RouteNo": "04", "_StartStop": "Bến xe An Sương", "_EndStop": "Bến xe buýt Sài Gòn", "_Distance": 154
```


```
Enter your choice: 2
Enter your query: Give me all routes have RouteVarShortName = Bến Thành
route_var_query(RouteVarShortName="Bến Thành")
JSON file created successfully.
```


```
output > {} outputJSONRoutes.json > ...
{"_RouteId": 3, "_RouteVarId": 6, "_RouteVarName": "Lượt về: Thạnh Lộc - Bến Thành", "_RouteVarShortName": "Bến Thành", "_RouteNo": "03", "_St
{"_RouteId": 1, "_RouteVarId": 2, "_RouteVarName": "Lượt về: BX Chợ Lớn - Bến Thành", "_RouteVarShortName": "Bến Thành", "_RouteNo": "01", "_S
{"_RouteId": 4, "_RouteVarId": 8, "_RouteVarName": "Lượt về: An Sương - Công viên 23/9", "_RouteVarShortName": "Bến Thành", "_RouteNo": "04",
{"_RouteId": 40, "_RouteVarId": 80, "_RouteVarName": "Lượt về: Hiệp Thành - Bến Thành", "_RouteVarShortName": "Bến Thành", "_RouteNo": "18", "
{"_RouteId": 41, "_RouteVarId": 82, "_RouteVarName": "Lượt về: Đại học Quốc gia - Bến xe buýt Sài Gòn", "_RouteVarShortName": "Bến Thành", "_R
{"_RouteId": 42, "_RouteVarId": 84, "_RouteVarName": "Lượt về: Bình Khánh - Bến xe buýt Sài Gòn", "_RouteVarShortName": "Bến Thành", "_RouteNo
{"_RouteId": 57, "_RouteVarId": 114, "_RouteVarName": "Lượt về: Bến xe buýt Thới An - Bến xe buýt Sài Gòn", "_RouteVarShortName": "Bến Thành",
{"_RouteId": 102, "_RouteVarId": 204, "_RouteVarName": "Lượt về: Bến xe An Sương - Bến xe buýt Sài Gòn", "_RouteVarShortName": "Bến Thành", "_
{"_RouteId": 126, "_RouteVarId": 2, "_RouteVarName": "Lượt về: Cầu Long Kiến - Bến xe buýt Sài Gòn", "_RouteVarShortName": "Bến Thành", "_Rout
{"_RouteId": 128, "_RouteVarId": 256, "_RouteVarName": "Lượt về: Chợ Long Phước - Bến Thành", "_RouteVarShortName": "Bến Thành", "_RouteNo": "
{"_RouteId": 133, "_RouteVarId": 2, "_RouteVarName": "Lượt về: Bến xe Miền Đông mới - Bến xe buýt Sài Gòn", "_RouteVarShortName": "Bến Thành",
```


```
Enter your query (enter exit to Exit): I want to go to stop 45 as fast as possible. I'm now on stop 4
shortest_path(start="4", end="45")
Find shortest path and save successfully.
GeoJSON file created successfully.
```


```
output > {} shortestPaths.json > ...
{
    "type": "FeatureCollection",
    "features": [
    {
        "type": "Feature",
        "properties": {},
        "geometry": {
          "coordinates": [[106.66041565,10.75296211],[106.66207886,10
          "type": "LineString"
        }
    }
            ]
}
```


VIII) Result of Llama 2 (just for comparison with Gorilla)

- Set up our PDF file first

```
PS C:\Users\USER\Desktop\Submit University\Solo Project\23125062_W11_01\AI_Model_LocalGPT_Method1\localGPT> python ingest.py
2024-04-27 17:01:27,783 - INFO - ingest.py:147 - Loading documents from C:\Users\USER\Desktop\Submit University\Solo Project\23125062_W11_01\AI_Model_LocalGPT_Method1\
localGPT/SOURCE_DOCUMENTS
Importing: TechnicalReportW07.pdf
C:\Users\USER\Desktop\Submit University\Solo Project\23125062_W11_01\AI_Model_LocalGPT_Method1\localGPT/SOURCE_DOCUMENTS\TechnicalReportW07.pdf loaded.

2024-04-27 17:01:42,693 - INFO - ingest.py:156 - Loaded 1 documents from C:\Users\USER\Desktop\Submit University\Solo Project\23125062_W11_01\AI_Model_LocalGPT_Method1
\localGPT/SOURCE_DOCUMENTS
2024-04-27 17:01:42,694 - INFO - ingest.py:157 - Split into 30 chunks of text
2024-04-27 17:01:43,779 - INFO - SentenceTransformer.py:66 - Load pretrained SentenceTransformer: hkunlp/instructor-large
load INSTRUCTOR_Transformer
max_seq_length  512
2024-04-27 17:01:47,917 - INFO - ingest.py:168 - Loaded embeddings from hkunlp/instructor-large
```

- Then type this**: python run_localGPT.py** and enter query.

***Notes:** In the context of this illustration, let's delve into my report discussing the application of Dijkstra's algorithm within the bus mapping framework to assess the compatibility of Llama 2 with handling queries. As previously highlighted, Llama 2 lacks fine-tuning for function calling, rendering it incapable of directly executing functions based on user prompts. Instead, interactions with Llama 2 are primarily document-based, limiting its ability to dynamically call functions, unlike the Gorilla model. This inherent limitation places Llama 2 at a disadvantage in comparison to the Gorilla model, which excels in function-calling tasks, thereby underscoring the pivotal role of model selection in optimizing system performance and user experience.

```
> Question:
What is the algorithm that Kiet used in the report to find shortest paths?

> Answer:
  Hello! I'm happy to help you with your question. In the provided context, Kiet used an algorithm called Dijkstra's algorithm to find the shortest paths in the graph.
Dijkstra's algorithm is a well-known algorithm for finding the shortest path in a graph, and it is often used in graph theory and computer science. The algorithm works
 by maintaining a priority queue of nodes, where the priority of each node is its distance from the starting node. The algorithm repeatedly extracts the node with the
minimum distance from the priority queue and updates the distances of its neighbors. This process continues until the destination node is reached or the priority queue
 is empty.
In more detail, the steps of Dijkstra's algorithm are as follows:
1. Initialize the distance of the starting node to 0, and the distance of all other nodes to infinity.
2. Add the starting node to the priority queue.
3. While the priority queue is not empty, extract the node with the minimum distance from the priority queue and update the distances of its neighbors.
4. If the extracted node is the destination node, return the shortest path from the starting node to the destination node. Otherwise, add all unvisited nodes that are
reachable from the extracted node to the priority queue.
5. Repeat step 3 until the priority queue is empty.

I hope this helps! Let me know if you have any other questions.
```

- However, if we ask Llama2 to find the shortest path, It cannot call our desired function but just can make logic based on our report.

````
> Question:
Find the shortest path from stop 4 to stop 7

> Answer:
  Sure, I'd be happy to help you with that! To find the shortest path from stop 4 to stop 7, we can use Dijkstra's algorithm for one stop. We will return the distance
(dist) and trace of the shortest path from stop 4 to stop 7.
Here is the implementation of Dijkstra's algorithm for one stop:
```
       5) Dijkstra's algorithm for one stop and then save it to files

Similar to the implementation of Dijkstra's algorithm for all pairs, but this time we will do with one stop and also handle the reading one into a JSON file and a norm
al text file. We will return the dist and trace of this stop.

After Dijkstra for one-stop, we can trace back the shortest path from that stop to any other stops easily through the dictionary trace that it returns.

The loop before is to find the corresponding path in the correct routeId and routeVarId from start_stop_id to end_stop_id.

Finally, our job is to save the paths to the JSON file and text file:

6) Counting the importance of a stop
```
Now, let's apply Dijkstra's algorithm for one stop to find the shortest path from stop 4 to stop 7. We will use the context provided earlier to determine the distances
 and routes between each pair of stops.
The distance from stop 4 to stop 7 is 3. The trace of the shortest path from stop 4 to stop 7 is [4, 5, 2].
Therefore, the shortest path from stop 4 to stop 7 is 4 -> 5 -> 2.
I hope this helps! Let me know if you have any questions or need further assistance.
````

IX) Conclusion

- Overall, I think we have made a significant success when we can apply the Large Language Model to our program for better usage. As I mentioned above, many LLMs are pre-trained and finetuned for function calling, however, from my point of view, I think Gorilla has done a really good job of connecting humans with LLMs through api_call.

- The utilization of Dijkstra's algorithm within the context of bus mapping, coupled with AI models, represents a significant advancement in query handling and function calling efficiency. By employing natural language processing techniques, we have facilitated seamless interaction between users and the system, enhancing accessibility and user experience.

- Why utilize Large Language Models (LLMs) in this context?" you might inquire. Consider this scenario: when developing an application for users to search for the shortest path between two stops, or to identify all stops with a specific ID, or even locate a route tailored to their needs, LLMs offer distinct advantages. They streamline these tasks, making them more accessible and efficient.

→ The Gorilla model, fine-tuned for function calls, has demonstrated impressive performance in deciphering user queries and executing corresponding functions accurately. Through meticulous definition of functions and integration of APIs, we have empowered the model to navigate complex datasets and provide precise responses, thereby optimizing system performance.

+ Furthermore, comparison with other Large Language Models (LLMs) underscores Gorilla's efficacy, particularly in zero-shot modelling and API navigation. Its versatility and adaptability make it a standout choice for handling diverse queries and datasets, outperforming alternatives by a significant margin.

+ The visualization of the operational framework elucidates the intricacies of function invocation and response handling, showcasing the seamless interaction between users and the model. By establishing a clear workflow and interface, we have fostered a user-centric approach that prioritizes efficiency and accuracy.

+ The user interface, adept at handling queries across various domains such as stops, routes, and paths, further exemplifies the versatility and utility of our system. By allowing users to specify their query areas, we have optimized performance and accuracy, enhancing the overall user experience.

→ In conclusion, the successful integration of Dijkstra's algorithm within a robust AI framework represents a significant milestone in transit mapping and query handling. Through meticulous implementation and fine-tuning, we have created a powerful tool capable of addressing diverse user needs and facilitating informed decision-making in public transit navigation.

X) Acknowledgement

- Information about the Gorilla model is supported by the paper written by:

+ Shishir G. Patil1∗

+ Tianjun Zhang1,∗

+ Xin Wang2

+ Joseph E. Gonzalez1

- From UC Berkeley – Microsoft Research