🦍 Gorilla: Large Language Model Connected with Massive APIs

Berkeley Function-Calling Leaderboard

Fanjia Yan* Huanzhi Mao* Charlie Cheng-Jie Ji* Ion Stoica Joseph E. Gonzalez Tianjun Zhang
Shishir G. Patil

'024-08-19 [Change Log]

, it is increasingly common to integrate Large Language Models (LLMs) to power many applications and software (e.g., ıa Index, AutoGPT, Voyager). Models like GPT, Gemini, Llama, Mistral etc, have demonstrated huge potential in this ıction calling (also called tool calling) capabilities.

ˈkeley Function-Calling Leaderboard (BFCL), the first comprehensive evaluation on the LLM's ability to call tools. We built this dataset from our learnings, to be representative of most users' function calling use-cases, for nts, as a part of enterprise workflows, etc. We consider function calls of various forms including parallel (one function nvocations of the function output) and multiple (multiple functions input, one function output), diverse languages JavaScript, etc. Further, we even execute these functions to execute the models, and we also evaluate the model's d picking any function when the right function is not available. And one more thing - the leaderboard now also includes for all the different models!

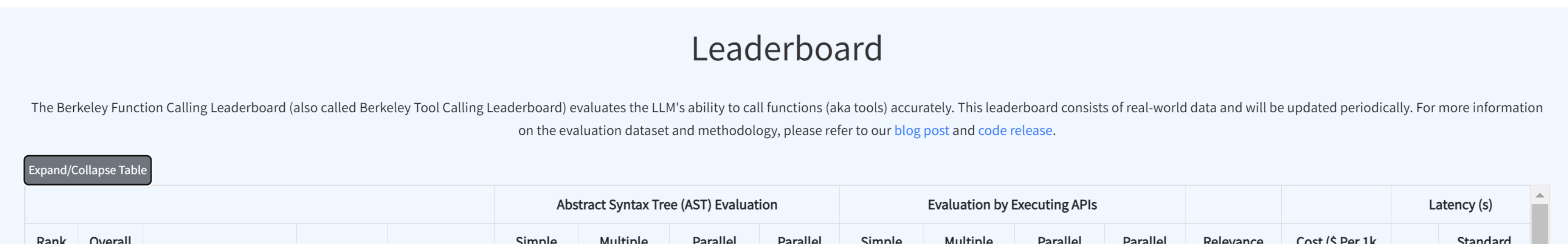
ʼ024, we released the BFCL V2 dataset, featuring enterprise-contributed data, tackling issues like bias and data ınd focuses on dynamic, real-world scenarios. Check out the *BFCL V2 · Live* Blog Post for more details.

ːrboard: Website
uation Dataset: HuggingFace Dataset 🤗
.ive: Blog Post
ıno: HuggingFace Space 🤗
ɔility: Github Code
tions-v2 (6.91B) on HuggingFace 🤗: gorilla-llm/gorilla-openfunctions-v2

Function Calling Leaderboard 🏆

ın-Calling Leaderboard (BFCL) aims to provide a thorough study of the function-calling capability of different LLMs. It ıestion-function-answer pairs with multiple languages (python, java, javascript, restAPI), diverse application domains and complex use cases (multiple function calls where the LLM needs to select one or more functions from multiple functions provided, and parallel function calls that the LLM needs to make multiple function calls together). We also investigate function relevance detection, to determine how the model will react when the provided function is not suitable to answer the user's question (in such case an "Error Message will be provided"). In more detail, **BFCL includes 100 Java, 50 JavaScript, 70 REST API, 100 SQL, and 1,680 Python on various simple, parallel, multiple, executable functions calling scenarios as well as function relevance detection**.

The leaderboard is shown below in the Figure, we can see that the latest checkpoint of GPT-4 (from OpenAI) leads the evaluation, with the open-source model (OpenFunctions-v2), Mistral-medium model (from Mistral AI) and Claude-2.1 (from Anthropic) following close behind. This blog post includes more information on the dataset, the evaluation methodology, some common failure patterns, and more!



Rank	Overall Acc	Model	Organization	License	Simple Function	Multiple Functions	Parallel Functions	Parallel Multiple	Simple Function	Multiple Functions	Parallel Functions	Parallel Multiple	Relevance Detection	Cost ($ Per 1K Function Calls)	Mean	Standard Deviation
1	79.35	Claude-3-Opus-20240229 (Prompt)	Anthropic	Proprietary	83.09	90.5	80	62.5	83.53	74	70	45	80.83	10.81	5.67	1.43
2	78.71	GPT-4-0125-Preview (Prompt)	OpenAI	Proprietary	83.64	90	86.5	64.5	82.94	76	68	42.5	69.17	5.22	2.1	1.4
3	78.35	Gorilla-OpenFunctions-v2 (FC)	Gorilla LLM	Apache 2.0	83.27	93	85.5	66	87.06	76	68	47.5	60.83	1.53	2.38	2.05
4	78.18	GPT-4-0125-Preview (FC)	OpenAI	Proprietary	78.91	91	88.5	66.5	70	68	70	47.5	81.67	4.76	4.37	5.31
5	77.71	GPT-4-1106-Preview (FC)	OpenAI	Proprietary	77.45	88	88	62	79.41	74	70	45	80.83	4.95	6.45	6.45
6	73.82	Claude-3-Sonnet-20240229 (Prompt)	Anthropic	Proprietary	79.45	85.5	85	65.5	75.29	74	70	45	53.33	2.13	2.38	0.66
7	72.24	Mistral-Medium-2312 (Prompt)	Mistral AI	Proprietary	74.73	73.5	79.5	53.5	67.65	68	64	27.5	88.33	1.75	3.41	4.65
8	70.71	Functionary-Small (FC)	MeetKai	MIT	72.36	86.5	76.5	55.5	60	72	66	45	74.17	1.94	3.02	3.2
9	69.18	Claude-instant-1.2 (Prompt)	Anthropic	Proprietary	76.18	84	78.5	46	76.47	66	58	45	54.17	0.95	1.75	1
		Claude-3-Opus-														

FC = native support for function/tool calling.

nate of the cost per 1000 function calls, in USD. **Latency** is measured in seconds.

nted average of the four test categories under AST Evaluation. **Exec Summary** is the unweighted average of the four test categories under Exec Evaluation.

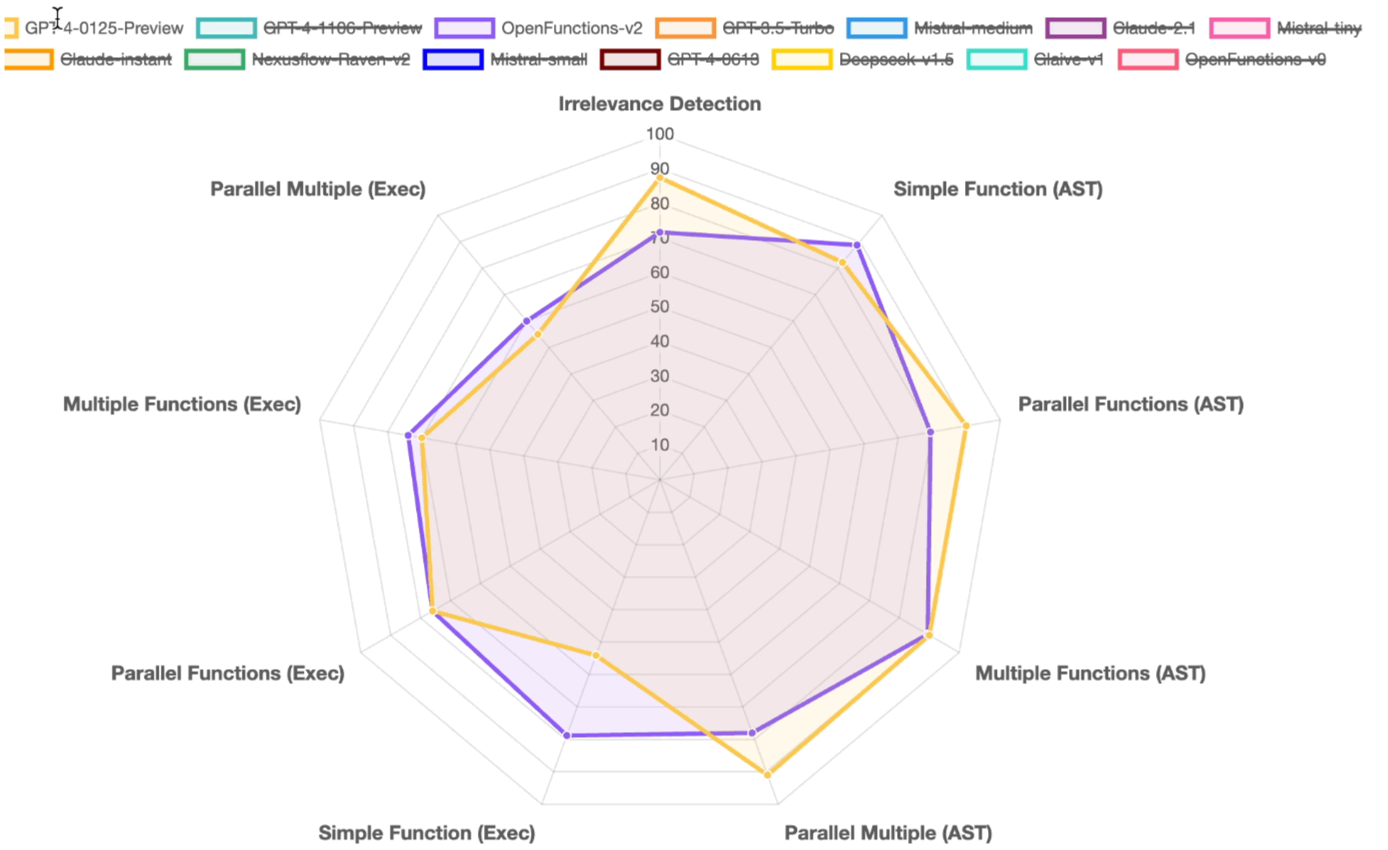
ort. If you would like to add your model or contribute test-cases, please contact us via discord.

LLMs' performance on Berkeley Function-Calling Leaderboard (BFCL)

ınderstanding and visualization of the outcomes, we have introduced an interactive wagon wheel tool that allows users ıus models. This comparison is organized into nine distinct categories: function relevance detection, AST (Abstract ꜫ analysis, and execution function call verification across simple, multiple, and parallel multiple function scenarios. ıroach, it becomes evident that tests reveal unsatisfactory performance by the models. Specifically, in simple function ırietary and open-source models exhibit comparable performance. However, when it comes to handling multiple and calls, the GPT-series models demonstrate superior performance over their open-source counterparts.

Wagon Wheel

The following chart shows the comparison of the models based on a few metrics. You can select and deselect which models to compare.



Detailed analysis using Berkeley Function-Calling Leaderboard (BFCL) Wagon Chart

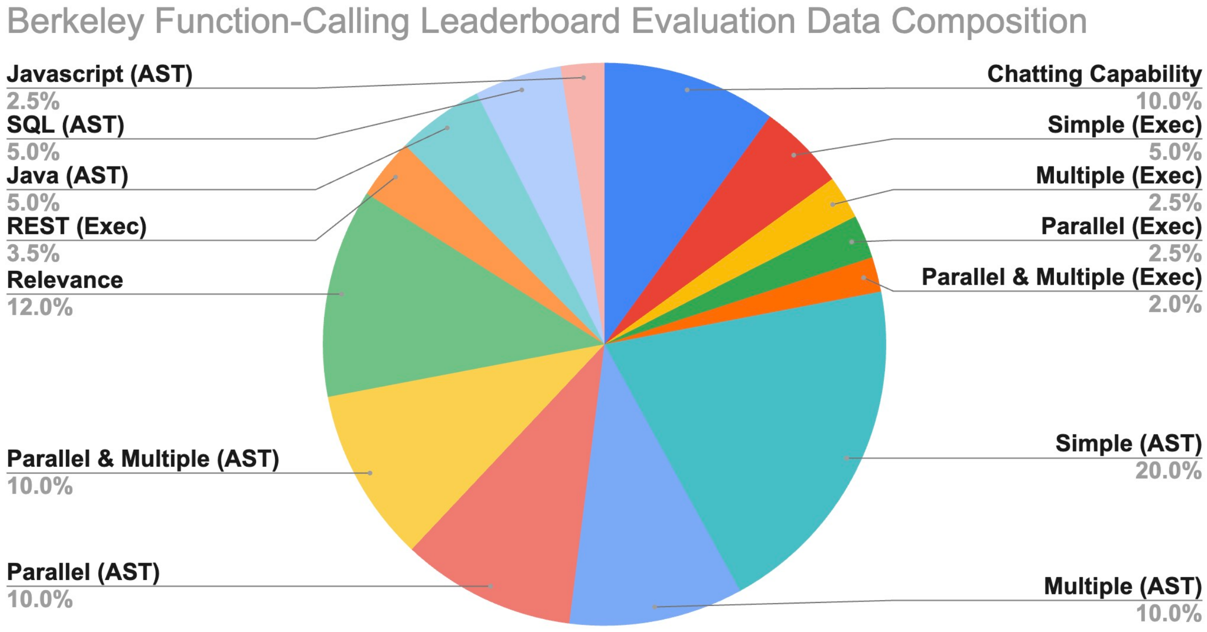
Dataset Composition

The Gorilla OpenFunctions evaluation dataset grows from its previous `OpenFunctions-v0` 's 100 data points to 2,000 data points! Beyond improvements in quality, the expanded dataset demonstrates diversity in:

- Domains of functions documentation
- Number of function documents and function call(s) pairs
- Data types of different programming languages

- Executability of real-world examples

Our evaluation JSON functions are scraped and generated from different sources of websites. We intentionally include domains like using functions related to `Mathematics-Algebra`, `Sports-Soccer`, `Finance-Mortgage`, etc. We include 40 sub-domains of functions within our generic evaluations. This allows us to understand the model performance not just in data-abundant domains like computing, and cloud, but also in niche domains like sports, and law.



[Berkeley Function-Calling Leaderboard](https://gorilla.cs.berkeley.edu/leaderboard.html) (BFCL) Data Composition

n Categories 📊

the majority of the evaluation into two categories:

mple Function, Multiple Function, Parallel Function, Parallel Multiple Function

›n: Chatting Capability, Function Relevance Detection, REST API, SQL, Java, Javascript

ation

n: Single function evaluation contains the simplest but most commonly seen format, where the user supplies a single ocument, with one and only one function call will be invoked.

›n: Multiple function category contains a user question that only invokes one function call out of 2 to 4 JSON function The model needs to be capable of selecting the best function to invoke according to user-provided context.

›n: Parallel function is defined as invoking multiple function calls in parallel with one user query. The model needs to / function calls need to be made and the question to model can be a single sentence or multiple sentence.

› Function: Parallel Multiple function is the combination of parallel function and multiple function. In other words, the d with multiple function documentation, and each of the corresponding function calls will be invoked zero or more

as both AST and its corresponding executable evaluations. In the executable evaluation data, we manually write ; drawing inspiration from free REST API endpoints (e.g. get weather) and functions (e.g. linear regression) that . The executable category is designed to understand whether the function call generation is able to be stably utilized in applications utilizing function calls in the real world.

Non-Python Evaluation

While the previous categories consist of the majority of our evaluations, we include other specific categories, namely Chatting Capability, Function Relevance Detection, REST API, SQL, Java, and JavaScript, to evaluate model performance on diverse scenarios and support of multiple programming languages, and are resilient to irrelevant questions and function documentations.

Chatting Capability: In Chatting Capability, we design scenarios where no functions are passed in, and the users ask generic questions - this is similar to using the model as a general-purpose chatbot. We evaluate if the model is able to output chat messages and recognize that it does not need to invoke any functions. Note the difference with “Relevance” where the model is expected to also evaluate if any of the function inputs are relevant or not. We include this category for internal model evaluation and exclude the statistics from the live leaderboard. We currently are working on a better evaluation of chat ability and ensuring the chat is relevant and coherent with users' requests and open to suggestions and feedback from the community.

Function Relevance Detection: In function relevance detection, we design scenarios where none of the provided functions are relevant and supposed to be invoked. We expect the model's output to be no function call. This scenario provides insight into whether a model will hallucinate on its function and parameter to generate function code despite lacking the function information or instructions from the users to do so.

REST API: A majority of the real-world API calls are from REST API calls. Python mainly makes REST API calls through `requests.get()`, `requests.post()`, `requests.delete()`, etc that are included in the Python requests library. GET requests are the most common ones used in the real world. As a result, we include real-world GET requests to test the model's capabilities to generate executable REST API calls through complex function documentations, using `requests.get()` along with the API's hardcoded URL and description of the purpose of the function and its parameters. Our evaluation includes two variations. The first type requires passing the parameters inside the URL, called path parameters, for example, the `{Year}` and `{CountryCode}` in `GET /api/v3/PublicHolidays/{Year}/{CountryCode}`. The second type requires the model to put parameters as key/value pairs into the `params` and/or `headers` of `requests.get(.)`. For example, `params={'lang': 'fr'}` in the function call. The model is not given which type of REST API call it is going to make but needs to make a decision on how it's going to be invoked.

For REST API, we use an executable evaluation to check for the executable outputs' effective execution, response type and response JSON key consistencies. On the AST, we chose not to perform AST evaluation on REST mainly because of the immense number of
for complicated defined APIs that enumeration of all possible answers is exhaustive.

ation data includes our customized `sql.execute` functions that contain sql_keyword, table_name, columns, and
e four parameters provide the necessary information to construct a simple SQL query like `SELECT column_A from`
`column_C == D` Through this, we want to see if through function calling, SQL query can be reliably constructed and
an training a SQL-specific model. In our evaluation dataset, we restricted the scenarios and supported simple
ling `SELECT`, `INSERT INTO`, `UPDATE`, `DELETE`, and `CREATE`. We included 100 examples for SQL AST evaluation.
ST evaluation will not be shown in our leaderboard calculations. We use SQL evaluation to test the generalization
calling for programming languages that are not included in the training set for Gorilla OpenFunctions-v2. We opted to
formance from the AST evaluation in the BFCL due to the multiplicity of methods to construct SQL function calls
al outcomes. We're currently working on a better evaluation of SQL and are open to suggestions and feedback from
Therefore, SQL has been omitted from the current leaderboard to pave the way for a more comprehensive evaluation in
tions.

pt: Despite function calling formats being the same across most programming languages, each programming language
ecific types. For example, Java has the `HashMap` type. The goal of this test category is to understand how well the
nodel can be extended to not just Python type but all the language-specific typings. We included 100 examples for Java
and 70 examples for Javascript AST evaluation.

utlined above provide insight into the performance of different models across popular API call scenarios, offering
tives on the potential of function-calling models.

Evaluation Categories

a hierarchical categorization on our existing categories to have nine categories showcased in our Berkeley Function-
ard BFCL, which we group by on both evaluation method (AST or execution) and type of functions (simple, parallel,
multiple functions). Here, we display a table organizing counts of evaluation data points of each leaderboard category,
ed of more granular categories listed in the blog. Specifically, we categorize REST executable evaluation as `Simple`
`uation by Executing APIs)` because we considered cases where one REST API call is being called. For Java +
ation, we categorize these into `Simple Function (Abstract Syntax Tree (AST) Evaluation)` since our current
version of the evaluation set didn't include multiple, parallel, and parallel multiple cases of diverse programming languages.

The final counts of each of the nine categories shown in BFCL with the composition of more granular types are shown in the following table

Abstract Syntax Tree (AST) Evaluation 🌳				Evaluation by Executing APIs ⚙️				Relevance Detection
Simple Function	**Multiple Functions**	**Parallel Functions**	**Parallel Multiple**	**Simple Function**	**Multiple Functions**	**Parallel Functions**	**Parallel Multiple**	
Py: 400 Java: 100 JS: 50	Py: 200	Py: 200	Py: 200	Py: 100 REST: 70	Py: 50	Py: 50	Py: 40	Py: 240

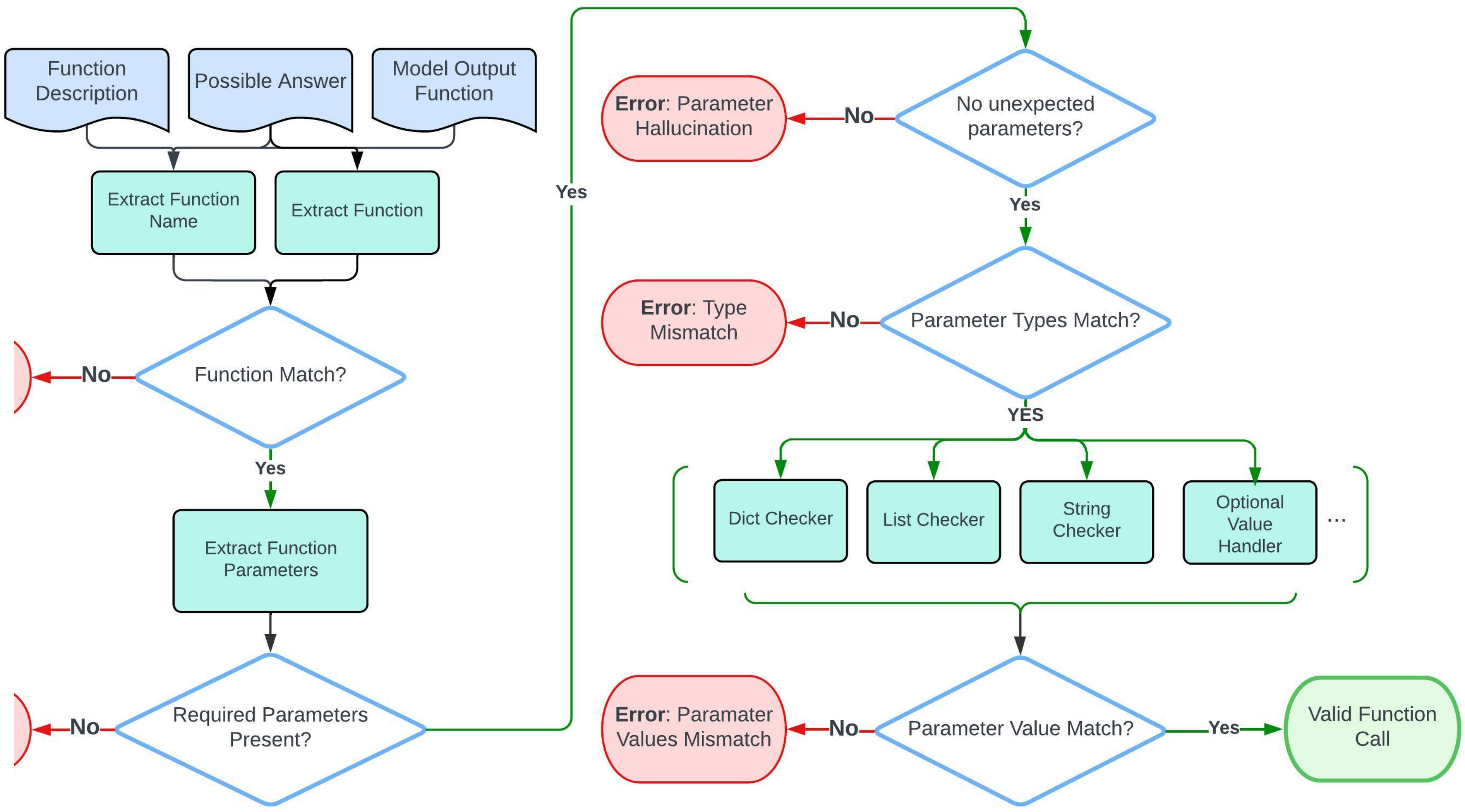
Evaluation Metrics📈

We use two popular methods to evaluate the accuracy of the model-generated answers: AST evaluation and Executable evaluation. Ideally one should use execution evaluation, but when we evaluate the answers, not all the results are easily executable (e.g., Java functions). So we use the AST as a complement to the execution evaluation.

- Abstract Syntax Tree (AST) Evaluation
- Executable Function Evaluation

Abstract Syntax Tree (AST) Evaluation 🌳

For **simple function evaluations**, the evaluation process focuses on comparing a *single model output function* against its *function doc* and *possible answers*. Here is a flow chart that shows the step-by-step evaluation process.



aluating function calls. The function description, and possible answer are used to evaluate the model's output function.

tion Through AST

rocess starts by parsing the function call using the AST tree.

```
ngle_area(base=10, height=5)
```

```
alue=List(elts=[
ll(
    func=Name(id='calculate_triangle_area', ctx=Load()),
    args=[],
    keywords=[
        keyword(arg='base', value=Constant(value=10)),
        keyword(arg='height', value=Constant(value=5))
    ]
)
    ], ctx=Load()))
], type_ignores=[])
```

Function Matching

The procedure first extracts the function name and verifies that it is consistent with the one in possible answer.

- Here, note that the function name may contain a `.`. Given that certain models (e.g. OpenAI series) may not support the dots `.` in their input, we substitute dots with underscores `_` in function names when inferencing the models to generate function outputs. This substitution is repeated during the result evaluation phase.

Required Parameters Matching

Then, it extracts the arguments from the AST and check if each parameter can be found and exact matched in possible answers.

- The evaluation process ensures all **required parameters**, as identified by the "required" attribute in the function documentation, are present in the model output.
- It also ensures that only **parameters exist in the function doc** are used, flagging the model hallucination outputs.

Parameter Type & Value Matching

The evaluation process is strict on **typing**. Here are the acceptable answers for each data type:

- For `bool` :
  - The procedure checks the direct matching of boolean values, and doesn't allow leniency on the string versions of boolean values.
- For `integer` , `float` :
  - For **Python tests only**, we allow the use of `int` values for Python parameters expecting `float` values to accommodate
    ython auto-conversion feature from `int` to `float` .

on-Python languages (Java and JavaScript), if the function documentation specifies `float` parameter, then it should
`float` in the model output (such as `5.0`); an `int` (such as `5`) will not be correct.
lying `float` value for `int` parameter is not allowed in any language.

`Tuple` :
r matters and the elements **must match exactly**. For example, `[1,2,3]` is not equal to `[2,3,1]` . So for questions
e the order in the list doesn't matter, permutations of the possible answer are used to accommodate this situation.
that the **type match** extends *recursively* for nested data structures (`List` or `Tuple`), where both the outer type and
ner types of elements must match the specified requirements.

g :
evaluation process is *case-insensitive*.
rings will be standardized before checking. This applies to both the model output and the possible answers.
All white space is removed.
A subset of punctuations `,./-_*^` are removed to make the evaluation more robust and accurate.
ible date `["20th June", "2023-06-20", "06/20/2023", "Jun.20, 2023"]`
ible Location `["New York City", "NYC"]`
ible Anything `["Manchester United", "Man United", "Man U", "MUFC"]`

evaluation focuses on the *key presence* and the *accuracy of associated values* as per the possible answers.
ring within dictionaries is not considered due to they are inherently *unordered*.
`of dictionaries` :
e the ordering of dictionaries is considered (since it's a `List`), the order of key-value pairs within each dictionary is not.
ptional Parameters in Function Calls:
arameters that are truly **optional** (in other words, the function doc didn't list that parameter as required and the
ble answer contains the empty string `""`), the model can choose to use the *default value* for that parameter, or not
de value. Both are considered correct.
arameters that aren't listed as *required* in the function doc, and the possible answer didn't contain the empty string `""` , then it is not truly *optional*. The prompt must imply that we should use some value different from the default for that parameter. So in this case, the model needs to explicitly provide the correct value; using default value or not providing value would be marked as wrong.

- When the parameter value is a **variable** mentioned in the prompt, the procedure special handles. For example, if the type of parameter `param1` is type `integer` , the user could provide `param1=value1` (where `value1` is a concrete value of type `integer`), or `param1=variable1` (where `variable1` is a variable that holds an `integer` value). So the evaluation takes into account both scenarios when checking the type.
- For the **Java** and **JavaScript** test categories, all the parameter values (in the input) are in the string format. The evaluation procedure will call the `java_type_converter` or the `js_type_converter` helper method to convert the string version of a **Java**/**JS** type into their corresponding Python format, since we perform checking with Python data types. For example, a `HashMap` in **Java** will be converted to a dictionary in Python. During this process, the converter will also perform the type checking for those parameters; e.g. if the parameter should be a long type in **Java**, the converter will check to make sure that the string input does have an `"L"` at the end (because otherwise, it wouldn't be a valid Java long).

Here are some examples of possible answers:

```
{"calculate_triangle_area": {"base": [10], "height": [5], "unit": ["units", "unit"]}}
```

```
{"predict_house_price": {"bedrooms": [3], "bathrooms": [2], "area": [1800], "location": ["San Francisco", "San Francisco, CA"]}}
```

Multiple/Parallel/Parallel-Multiple Functions AST Evaluation

The **multiple, parallel, or parallel-multiple function AST evaluation process** extends the idea in the simple function evaluation to support multiple model outputs and possible answers.

- The evaluation process first associates each possible answer with its function doc. Then it iterates over the model outputs and calls the **simple function evaluation** on each function (which takes in one *model output*, *one possible answer*, and one *function doc*).
  - The order of model outputs relative to possible answers is not required. A model output can match with any possible answer.
- The evaluation employs an *all-or-nothing* approach to evaluation. Failure to find a match across all model outputs for any given ıswer results in a *failed evaluation*.

· Function Evaluation ⚙️

le test category, we execute the generated API call to check for response correctness. The evaluation process differs ıd REST tests due to their distinct characteristics:

unction (Non-REST) Evaluation:

ınvolves running the specified function and examining its output.
criteria (either of the following must be met, depending on the executable function example):
t match: The output must exactly match the expected result.
·time match: A looser form of exact match that only applies to numerical execution result, where the execution result be within a certain percentage threshold (20%) from the expected result to accommodate the live updates of API ınses.
:tural match: The output must match the expected data type. For example, both `[1, 2, 3]` and `[1.5, 2.4, 6]` are ɔted if the expected type is a `list`. In addition, the following types have some special requirements:
For `List`, the length must match the expected length. The type of each element is not checked.
For `Dict`, the keys must match the keys present in the expected output. This means no extra keys nor missing keys. The type of each value is not checked.

unction (REST) Evaluation:

s involve executing API calls and assessing:
:tive execution: Assessing the success of API call executions.
ıonse type accuracy: Ensuring the API response matches the expected structure (e.g., `list of JSON` objects).
ʌ key consistency: Checking for consistency in *JSON key sets* between generated and expected responses.
th `REST` responses were initially gathered and stored in JSON format for comparison.
ʌariable nature of `REST` responses (e.g., changing weather data), the executable evaluation focuses on *structural* and *real-time execution success* rather than static values.
rticular, the executable evaluation verifies that the response type matches the ground truth (e.g., expecting a list of ɹSON objects) and checks for consistency in the number of elements and JSON key sets.

Acknowledgment: Given the potential for updates in `REST` API response structures by their developers, the evaluation methodology has an optional API sanity check to ensure that all APIs involved during the evaluation process are working as expected before running any executable category tests. Ground truth for the `REST` category are also reviewed and updated regularly to ensure the evaluation remains accurate and relevant.

Multiple/Parallel/Parallel-Multiple Executable Functions Evaluation

The **multiple, parallel, or parallel-multiple executable function evaluation process** extends the idea in the simple executable function evaluation.

- The evaluation process first executes each model-generated function call. Then, it iterates over the real-time executed outputs with the ground truth execution outputs, and calls the simple executable function evaluation procedure on each pair.
  - The order of model execution outputs relative to the ground truth execution outputs is not required.
- The evaluation employs an *all-or-nothing* approach. Failure to find a match across all model execution outputs for any given

ground truth execution output with its respective evaluation criteria results in a *failed evaluation*.

Cost & Latency

In our recent update, we have also paid close attention to the **cost** and **latency**.

- For models from service providers such as OpenAI, Mistral, Google, Anthropic, and etc:
  - **Latency**: we measure the latency by timing each request to the endpoint ignoring the function document preprocessing time.
  - **Cost**: we follow the formula to derive the cost per 1000 function callings.

$$1000 \times \left(\frac{1}{\#eval\ dataset\ entries} \sum_{d \in eval\ dataset} (\text{input token} \times \text{input token price} + \text{output token} \times \text{output token price}) \right)$$

- For models that we evaluate using local hosting. This includes Deepseek, Gemma, and etc.:
  - **Latency**: We calculated the number when serving the model with vLLM using 8 V100 GPUs. Since we batched and
ıated the model, we derive latency by dividing the total time by the number of evaluation dataset entries.
: Since the open source model does not have a price tag, we estimate the cost by:

$$\text{Cost} = \text{Latency per 1000 function call} * \text{8xV100 azure-pay-as-you-go-price per hour} \;/\; 3600$$

se the Azure ND40rs-v2 instance (8X V100 GPU) April 2024 pay-as-you-go pricing in the cost calculation. This is not o be precise as the price can change often. We will try our very best to keep this up-to-date on daily or at least weakly . Nonetheless this should give an idea of what the magnitude of costs should look like, and help understand the relative ing all things constant.

-v1, Nexusflow-Raven-v2, and Meetkai-Functionary, we use their endpoints while their services are free, so we did not for their models.

ce of insight will help individuals or enterprises to decide which model to incorporate based on the demand and budget.

'unction-call (tool-call) and when to prompt? 🖋

ɔdel cards below present insights and function-calling features supported by the different models we evaluate.

Basic Information				Function Calling Support			Data Type Support			
	Model Size	Organization	License	Function Calling Support	Parallel Functions	Multiple Functions	Java	Javascript	C++	Python
29	Unknown	Anthropic	Proprietary	✓	✓	✓	✗	✗	✗	✓
:	Unknown	OpenAI	Proprietary	✓	✓	✓	✗	✗	✗	✓
:	Unknown	OpenAI	Proprietary	✓	✓	✓	✗	✗	✗	✓
Gorilla OpenFunctions-v2	6.91B	Gorilla LLM	Apache 2.0	✓	✓	✓	✓	✓	✗	✓
Claude-3-Sonnet-20240229	Unknown	Anthropic	Proprietary	✓	✓	✓	✗	✗	✗	✓

From the model cards above, we highlight that our evaluation involves both function-calling and non-function-calling models. For function calling models, since they are specifically designed to generate function calls, we did not provide any system prompt but instead toggle the function calling mode on and put the function definitions where they should be. For non-function call models, we simply prompt them with system messages. We provide all the prompts we used to evaluate our propriety and open-source models.

1. For all the function calling models, we did not supply any system prompt but instead, toggle the function calling mode on and put the function definitions where they should be.

2. For chat model, we explicitly provide a **system message**:

```
SYSTEM_PROMPT_FOR_CHAT_MODEL = """
You are an expert in composing functions. You are given a question and a set of possible functions.
Based on the question, you will need to make one or more function/tool calls to achieve the purpose.
If none of the function can be used, point it out. If the given question lacks the parameters required by the
function, also point it out. You should only return the function call in tools call sections.
"""
```

```
USER_MESSAGE_FOR_CHAT_MODEL = "Questions:{user_prompt}\nHere is a list of functions in JSON format that you
can invoke:\n{functions}. Should you decide to return the function call(s), NO other text MUST be included."
```

Common Mistakes

With our benchmark BFCL, we are able to identify some common mistakes that LLMs make when generating function calls. These mistakes are interesting because they help us understand the limitations of the current models and provide insights into how to improve them.

1. GPTs' *function documents are difficult to format* and their *typings are restrictive* in real-world scenarios.


```
":
": "calculate_binomial_probability",
meters":
type": "object",
properties":
   "number_of_trials":
   {
       "type": "integer",
       "description": "The total number of trials."
   },
   "number_of_successes":
   {
       "type": "integer",
       "description": "The desired number of successful outcomes."
   },
   "probability_of_success":
   {
       "type": "float",
       The probability of a successful outcome on any given trial.",
       "default": 0.5
   }
   ...
required": ["number_of_trials", "number_of_successes"]
```

e, we need to manually convert float into number to make the function OpenAI compatible. In addition to that, `number` s convey less information compared to `float` s in terms of precision and type consistency.

In Gorilla OpenFunctions-v2, we improve the flexibility of the function documents by not restricting the typing of the parameters. In other words, the user can supply `Tuple` , `Float` , and even language-specific types like `Hashmap` and `Linked List` in Java!

2. GPT underperforms in scenarios where the *parameters are not immediately available* in the user question but instead require some implicit conversions. Here is an example:

```
"Function":
{
    "name": "finance.predict_future_value",
    ...
    "parameters":
    {
        "type": "object",
        "properties":
        {
            "present_value":
            {
                "type": "number",
                "description": "The present value of the investment."
            },

            "annual_interest_rate":
            {
                "type": "number",
                "description": "The annual interest rate of the investment."
            },

            "compounding_periods_per_year":
            {
                "type": "integer",
                "description": "The number of times that interest is compounded per year.",
            },
            "time_years":
            {
                "type": "integer",
                "description": "The investment horizon in years."
            }
            ...
required": ["present_value", "annual_interest_rate", "time_years"]
```

: Predict the future value of a $5000 investment with an annual interest rate of 5% in 3 years with monthly ing.

```
put:
": "finance.predict_future_value",
meters":

present_value": 5000,
annual_interest_rate": 5,
compounding_periods_per_year": 12,
time_years": 3

}]
```

```
Gorilla-openfunctions-v2 output:
[{
    "name": "finance.predict_future_value",
    "parameters":
    {
        "present_value": 5000,
        "annual_interest_rate": 0.05,
        "compounding_periods_per_year": 12,
        "time_years": 3
    }
}]
```

3. Chat Models tend to generate *malformed function calls* in which parameters can be extracted but not executable

Example: `mistral-medium` generates results like `solve\\_quadratic\\_equation(a=2, b=6, c=5)`. With gorilla-openfunctions-v2, we are able to directly output `solve_quadratic_equation(a=3, b=2, c=1)` which is executable upon receiving the result.

4. REST API *missing URLs*:

   A discrepancy arises due to the absence of the required URL in REST API requests made by GPT-4 for fetching weather data. While the GPT-4 output omits the necessary URL, the Gorilla Openfunctions-v2 model successfully includes the correct API endpoint, enabling it to successfully execute and retrieve the requested weather information for the specified coordinates and forecast period.

```
"User": "Can you fetch me the weather data for the coordinates
37.8651 N, 119.5383 W, including the hourly forecast for temperature,
wind speed, and precipitation for the next 10 days?"

"Function":
{
    "name": "requests.get",
```

Blog 8: Berkeley Function-Calling Leaderboard


```
meters":

type": "object",
properties":

   "url":
   {
       "type": "string",
       "description": "The API endpoint for fetching
       weather data from the Open-Meteo API for the
       given latitude and longitude, default
       https://api.open-meteo.com/v1/forecast"
   }
   ...
```

```
put:

": "requests.get",
meters": {
url": "Missing",
params":

   "latitude": "37.8651",
   "longitude": "-119.5383",
   "forecast_days": 10
,
```

```
Gorilla-Openfunctions-v2 output:
{
    "name": "requests.get",
    "parameters": {
        "url": "https://api.open-meteo.com/v1/forecast",
        "params":
        {
            "latitude": "37.8651",
            "longitude": "-119.5383",
            "forecast_days": 10
        },
    }
}
```

Conclusion

We provide a comprehensive and systematic evaluation of LLMs for function calling with Berkeley Function Calling Leaderboard. The studies here suggest that in terms of simple function calling (without complex planning and chained function calling), finetuning an open-source can be as effective as propriety models. Furthermore, we provide Gorilla Open Functions v2, an open-source model that can help users with building AI applications with function calling and interacting with JSON compatible output.

We hope you enjoyed this blog post. We would love to hear from you on [Discord](), [Twitter (#GorillaLLM)](), and [GitHub]().

Citation

If you would like to cite Gorilla:

@inproceedings{berkeley-function-calling-leaderboard,
title={Berkeley Function Calling Leaderboard},
author={Fanjia Yan and Huanzhi Mao and Charlie Cheng-Jie Ji and Tianjun Zhang and Shishir G. Patil and Ion Stoica and Joseph E. Gonzalez},
year={2024},
{\url{https://gorilla.cs.berkeley.edu/blogs/8_berkeley_function_calling_leaderboard.html}},

Blog 8: Berkeley Function-Calling Leaderboard