arXiv:2305.15334v1 [cs.CL] 24 May 2023

Gorilla: Large Language Model Connected with Massive APIs

Shishir G. Patil[1*] **Tianjun Zhang**[1,*] **Xin Wang**[2] **Joseph E. Gonzalez**[1]
[1]UC Berkeley [2]Microsoft Research
sgp@berkeley.edu

Abstract

Large Language Models (LLMs) have seen an impressive wave of advances recently, with models now excelling in a variety of tasks, such as mathematical reasoning and program synthesis. However, their potential to effectively use tools via API calls remains unfulfilled. This is a challenging task even for today's state-of-the-art LLMs such as GPT-4, largely due to their inability to generate accurate input arguments and their tendency to hallucinate the wrong usage of an API call. We release Gorilla, a finetuned LLaMA-based model that surpasses the performance of GPT-4 on writing API calls. When combined with a document retriever, Gorilla demonstrates a strong capability to adapt to test-time document changes, enabling flexible user updates or version changes. It also substantially mitigates the issue of hallucination, commonly encountered when prompting LLMs directly. To evaluate the model's ability, we introduce APIBench, a comprehensive dataset consisting of HuggingFace, TorchHub, and TensorHub APIs. The successful integration of the retrieval system with Gorilla demonstrates the potential for LLMs to use tools more accurately, keep up with frequently updated documentation, and consequently increase the reliability and applicability of their outputs. Gorilla's code, model, data, and demo are available at `https://gorilla.cs.berkeley.edu`

1 Introduction

Recent advances in large language models (LLMs) [10, 5, 32, 6, 29, 30] have enabled significant new capabilities including natural dialogue, mathematical reasoning, and program synthesis. However, despite these advances, LLMs are still fundamentally limited by the information they can store in a fixed set of weights and the things they can compute using a static computation graph and limited context. Furthermore, as the world changes, LLMs require retraining to update their knowledge and reasoning capabilities.

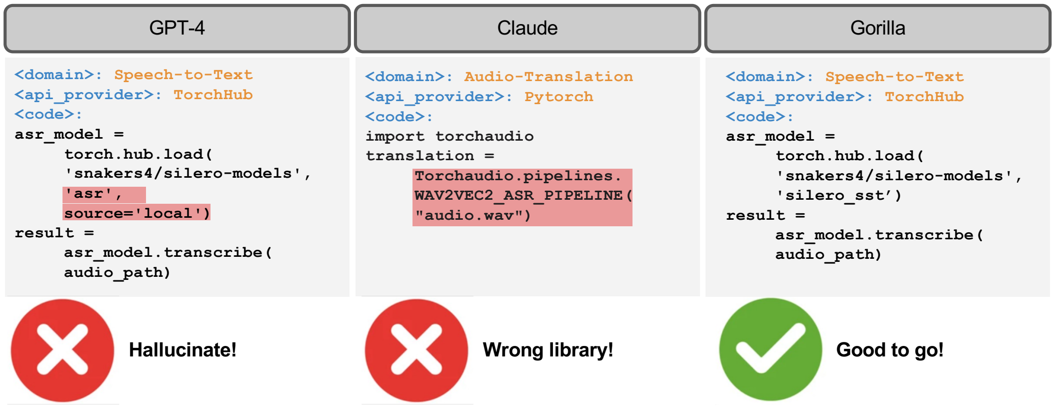
By empowering LLMs to use tools [33], we can grant access to vastly larger and changing knowledge bases and accomplish complex computational tasks. By providing access to search technologies and databases, [26, 39, 37] demonstrated that we can augment LLMs to address a significantly larger and more dynamic knowledge space. Similarly, by providing access to computational tools, [39, 2] demonstrated that LLMs can accomplish complex computational tasks. Consequently, leading LLM providers[29], have started to integrate plugins to allow LLMs to invoke external tools through APIs.

This transition from a small set of hand-coded tools, to the ability to invoke a vast space of changing cloud APIs could transform LLMs into the primary interface to computing infrastructure and the web. Tasks ranging from booking an entire vacation to hosting a conference, could become as simple as talking to an LLM that has access to the flight, car rental, hotel, catering, and entertainment web APIs. However, much of the prior work [35, 24] integrating tools into LLMs considered a small well documented set of APIs that can be easily injected into the prompt.

---

[*] Equal contribution.

Preprint. Under review.



```
<domain>: Speech-to-Text
<api_provider>: TorchHub
<code>:
asr_model =
     torch.hub.load(
     'snakers4/silero-models',
     'asr',
     source='local')
result =
     asr_model.transcribe(
     audio_path)
```

```
<domain>: Audio-Translation
<api_provider>: Pytorch
<code>:
import torchaudio
translation =
     Torchaudio.pipelines.
     WAV2VEC2_ASR_PIPELINE(
     "audio.wav")
```

```
<domain>: Speech-to-Text
<api_provider>: TorchHub
<code>:
asr_model =
     torch.hub.load(
     'snakers4/silero-models',
     'silero_sst’)
result =
     asr_model.transcribe(
     audio_path)
```

Prompt: Help me find an API to convert the spoken language in a recorded audio to text using Torch Hub.

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.

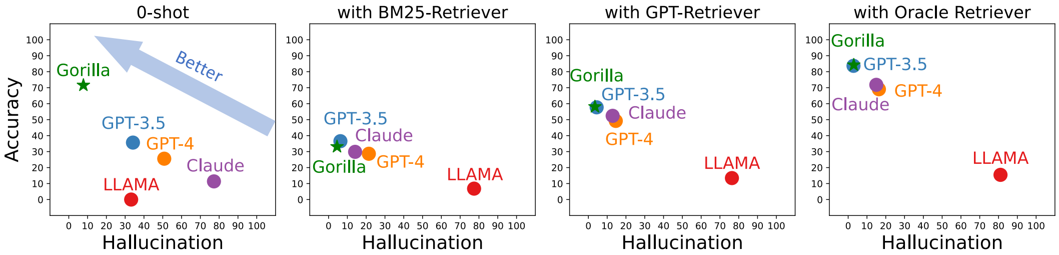


Figure 2: **Accuracy (vs) hallucination** in four settings, that is, *zero-shot* (i.e., without any retriever), and *with retrievers*. `BM25` and `GPT` are commonly used retrievers and the `oracle` retriever returns relevant documents at 100%, indicating an upper bound. Higher in the graph (higher accuracy)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

Supporting a web scale collection of potentially millions of changing APIs requires rethinking our approach to how we integrate tools. It is not longer possible to describe the full set of APIs in a single context. Many of the APIs will have overlapping functionality with nuanced limitations and constraints. Simply evaluating LLMs in this new setting requires new benchmarks.

In this paper, we explore the use of self-instruct fine-tuning and retrieval to enable LLMs to accurately select from a large, overlapping, and changing set tools expressed using their APIs and API documentation. We construct, APIBench, a large corpus of APIs with complex and often overlapping functionality by scraping ML APIs (models) from public model hubs. We choose three major model hubs for dataset construction: TorchHub, TensorHub and HuggingFace. We exhaustively include every API call in TorchHub (94 API calls) and TensorHub (696 API calls); For HuggingFace, since the models come in a large number and lots of the models don't have a specification, we choose the most downloaded 20 models per task category (in a total of 925). We also generate 10 synthetic user question prompts per API using Self-Instruct [42]. Thus, each entry in the dataset becomes an instruction reference API pair. We adopt a common AST sub-tree matching technique to evaluate the functional correctness of the generated API. We first parse the generated code into an AST tree, then find a sub-tree whose root node is the API call that we care about (e.g., `torch.hub.load`) and use it to index our dataset. We check the functional correctness and hallucination problem for the LLMs, reporting the corresponding accuracy.

We then finetune Gorilla, a LLaMA-7B-based model with document retrieval using our dataset. We find that Gorilla significantly outperforms GPT-4 in terms of API functionality accuracy as well as reducing hallucination errors. We show an example output in Fig. 1. Further, our retrieval-aware training of Gorilla enables the model to adapt to changes in the API documentation. Finally, we demonstrate Gorilla's ability to understand and reason about constraints.

2 Related Work

Large Language Models Recent strides in the field of LLMs have renovated many downstream domains [10, 40, 48, 47], not only in traditional natural language processing tasks but also in program synthesis. Many of these advances are achieved by augmenting pre-trained LLMs by prompting [44, 14] and instruction fine-tuning [11, 31, 43, 15]. Recent open-sourced models like LLaMa [40], Alpaca [38], and Vicuna [9] have furthered the understanding of LLMs and facilitated their experimentation. While our approach, Gorilla, incorporates techniques akin to those mentioned, its primary emphasis is on enhancing the LLMs' ability to utilize millions of tools, as opposed to refining their conversational skills. Additionally, we pioneer the study of fine-tuning a base model by supplementing it with information retrieval - a first, to the best of our knowledge.

Tool Usage The discussion of tool usage within LLMs has seen an upsurge, with models like Toolformer taking the lead [33, 19, 21, 26]. Tools often incorporated include web-browsing [34], calculators [12, 39], translation systems [39], and Python interpreters [14]. While these efforts can be seen as preliminary explorations of marrying LLMs with tool usage, they generally focus on specific tools. Our paper, in contrast, aims to explore a vast array of tools (i.e., API calls) in an open-ended fashion, potentially covering a wide range of applications.

With the recent launch of Toolformer [33] and GPT-4 [29], the importance of API calls has been highlighted, encouraging many works in employing API calls as tooling [35, 24]. Moreover, the application of API calls in robotics has been explored to some extent [41, 1]. However, these works primarily aim at showcasing the potential of "prompting" LLMs rather than establishing a systematic method for evaluation and training (including fine-tuning). Our work, on the other hand, concentrates on systematic evaluation and building a pipeline for future use.

LLMs for Program Synthesis Harnessing LLMs for program synthesis has historically been a challenging task [23, 7, 45, 16, 13, 20]. Researchers have proposed an array of strategies to prompt LLMs to perform better in coding tasks, including in-context learning [44, 18, 7], task decomposition [17, 46], and self-debugging [8, 36]. Besides prompting, there have also been efforts to pretrain language models specifically for code generation [28, 22, 27].

However, these strategies focus on prompting large language models or pre-training them for general program synthesis. In our research, in contrast, we focus on a much restricted domain: the synthesis of linear programs using API calls. General program synthesis, not only is complex, but is also hard to verify and evaluate. API calls, on the other hand, function more like tool usage. This allows the LLM to significantly expand its capabilities without grappling with low-level implementation details.

3 Methodology

In this section, we describe APIBench, a comprehensive benchmark constructed from TorchHub, TensorHub, and HuggingFace API Model Cards. We begin by outlining the process of collecting the API dataset and how we generated instruction-answer pairs. We then introduce Gorilla, a novel training paradigm with a information–retriever incorporated into the training and inference pipelines. Finally, we present our AST tree matching evaluation metric.

3.1 Dataset Collection

To collect the dataset, we meticulously recorded all online model cards for HuggingFace's "The Model Hub", PyTorch Hub, and TensorFlow Hub Models. Throughout the rest of the paper, we call these HuggingFace, Torch Hub, and TensorFlow Hub respectively for brevity.

API Documentation The HuggingFace platform hosts and servers about 203,681 models. However, many of them have poor documentation, lack dependencies, have no information in their model card, etc. To filter these out, we pick the top 20 models from each domain. We consider 7 domains in multimodal data, 8 in CV, 12 in NLP, 5 in Audio, 2 in tabular data, and 2 in reinforcement learning. Post filtering, we got a total of 925 models from HuggingFace. TensorFlow Hub is versioned into v1 and v2. The latest version (v2) has 801 models in total, and we process all of them. Post filtering out models, whose mode cards had little to no information, we are left with 626 models. Similar to

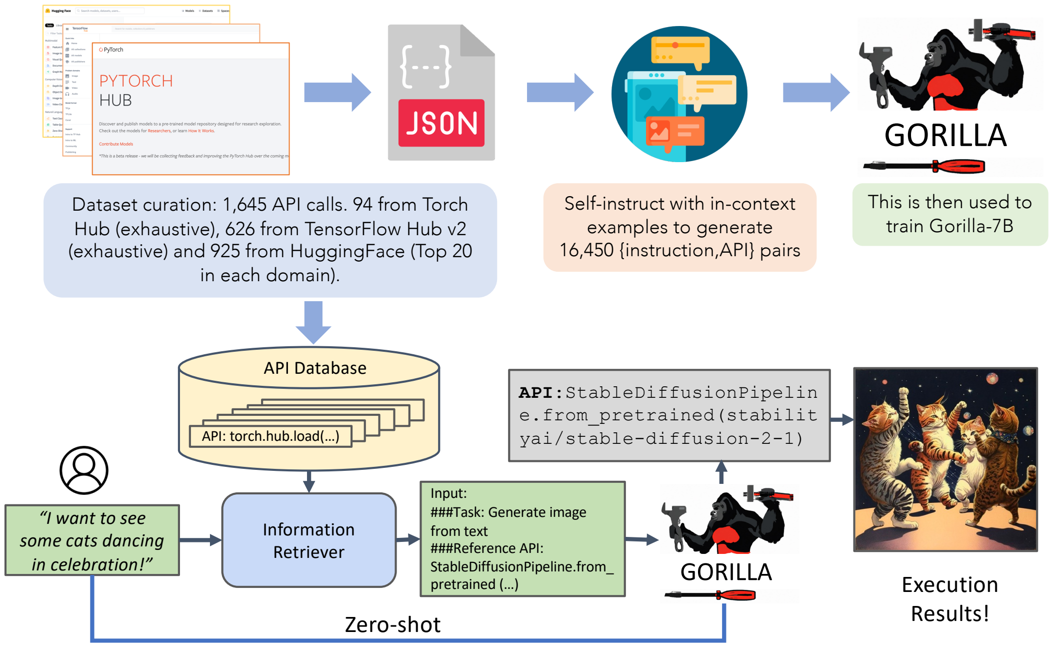

PYTORCH
HUB
JSON
GORILLA
Dataset curation: 1,645 API calls. 94 from Torch Hub (exhaustive), 626 from TensorFlow Hub v2 (exhaustive) and 925 from HuggingFace (Top 20 in each domain).
Self-instruct with in-context examples to generate 16,450 {instruction,API} pairs
This is then used to train Gorilla-7B
API Database
API: torch.hub.load(...)
"I want to see some cats dancing in celebration!"
Information Retriever
Input:
###Task: Generate image from text
###Reference API: StableDiffusionPipeline.from_pretrained (...)
API:StableDiffusionPipeline.from_pretrained(stabilityai/stable-diffusion-2-1)
GORILLA
Execution Results!
Zero-shot

Figure 3: **Gorilla: A system for enabling LLMs to interact with APIs.** The upper half represents the training procedure as described in Sec 3. This is the most exhaustive API data-set for ML to the best of our knowledge. During inference (lower half), Gorilla supports two modes - with retrieval, and zero-shot. In this example, it is able to suggest the right API call for generating the image from the user's natural language query.

TensorFlow Hub, we get 95 models from Torch Hub. We then converted the model cards for each of these 1,645 API calls into a json object with the following fields: {domain, framework, functionality, api_name, api_call, api_arguments, environment_requirements, example_code, performance, and description.}. We provide more information in the Appendix. These fields were chose to generalize beyond the API calls within ML domain, to other domains, includin RESTful API calls.

Instruction Generation Guided by the self-instruct paradigm [42], we employed GPT-4 to generate synthetic instruction data. We provided three in-context examples, along with a reference API documentation, and tasked the model with generating real-world use cases that call upon the API. We specifically instructed the model to refrain from using any API names or hints when creating instructions. We constructed six examples (Instruction-API pairs) for each of the three model hubs. These 18 points, were the only hand-generated or modified data. For each of our 1,645 API datapoints, we sample 3 of 6 corresponding instruction examples to generate a total of 10 instruction-api pairs as demonstrated in Figure 3. We would like to highlight that we only need to employ GPT-4 to generate the instructions and this can be swapped with open-source alternatives such as LLaMA, Alpaca, etc.

3.2 Gorilla

Our model Gorilla, is retrieve-aware finetuned LLaMA-7B model, specifically for API calls. As shown in Fig 3, we employ self-instruct to generate {instruction, API} pairs. To fine-tune LLaMA, we convert this to a user-agent chat-style conversation, where each data-point is a conversation with one round each for the user and the agent. We then perform standard instruction finetuning on the base LLaMA-7B model. For our experiments, we train Gorilla with and without the retriever.

API Call with Constraints API calls often come with inherent constraints. These constraints necessitate that the LLM not only comprehend the functionality of the API call but also categorize the calls according to different constraint parameters. This requirement introduces an additional layer of complexity to the process, demanding a more nuanced understanding from the LLM. Specifically, for machine learning API calls, two common sets of constraints are: parameter size and a lower bound on accuracy. Consider, for instance, the following prompt: ``‘‘Invoke an image classification model that uses less than 10M parameters, but maintains an ImageNet accuracy of at least 70%’’``. Such a prompt presents a substantial challenge for the LLM to accurately interpret and respond to. Not only must the LLM understand the user's functional description, but it also needs to reason about the various constraints embedded within the request. This challenge underlines the intricate demands placed on LLMs in real-world API calls. It is not sufficient for the model to merely comprehend the basic functionality of an API call; it must also be capable of navigating the complex landscape of constraints that accompany such calls. These observations necessitate the need to fine-tune an LLM for APIs.

Retriever-Aware training For training with retriever, the instruction-tuned dataset, also has an additional `"Use this API documentation for reference: <retrieved_API_doc_JSON>"` appended to the user prompt. Through this, we aim to teach the LLM to parse the second half of the question to answer the first half. We demonstrate that this a) makes the LLM adapt to test-time changes in API documentation, and b) improves performance from in-context learning, and finally c) show that it reduces hallucination error.

Surprisingly, we find that augmenting a LLM with retrieval, does not always lead to improved performance, and can at-times hurt performance. We share more insights along with details in Sec 4.

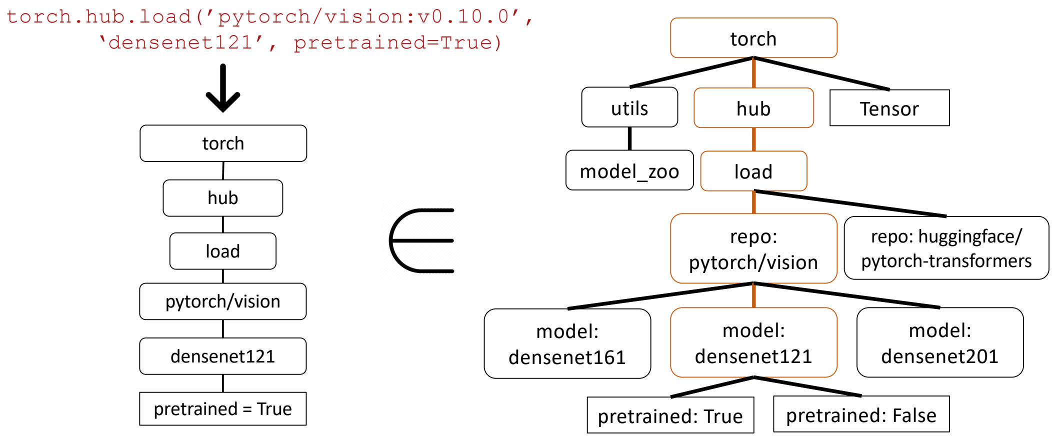
Gorilla Inference During Inference, the user provides the prompt in natural language (Fig: 3). This can be for a simple task (e.g, *"I would like to identify the objects in an image"*), or they can specify a vague goal, (.e.g, *"I am going to the zoo, and would like to track animals"*). Gorilla, similar to training, can be used for inference in two modes: zero-shot and with retrieval. In zero-shot, this prompt (with NO further prompt tuning) is fed to the Gorilla LLM model when then returns the API call that will help in accomplishing the task and/or goal. In retrieval mode, the retriever (either of BM25 or GPT-Index) first retrieves the most up-to-date API documentation stored in the API Database. This is then concatenated to the user prompt along with the message `Use this API documentation for reference:` before feeding it to Gorilla. The output of Gorilla is an API to be invoked. Besides the concatenation as described, we do *NO* further prompt tuning in our system. While we do have a system to execute these APIs, that is not a focus of this paper.

3.3 Verifying APIs

Inductive program synthesis, where a program is synthesized to satisfy test cases, has found success in several avenues [4, 25]. However, test cases fall short when evaluating API calls, as it is often hard to verify the semantic correctness of the code. For example, consider the task of classifying an image. There are over 40 different models that can be used for the task. Even if we were to narrow down to a single family of Densenet, there are four different configurations possible. Hence, there exist multiple correct answers and it is hard to tell if the API being used is functionally equivalent to the reference API by unit tests. Thus, to evaluate the performance of our model, we compare their functional equivalence using the dataset we collected. To trace which API in the dataset is the LLM calling, we adopt the AST tree-matching strategy. Since we only consider one API call in this paper, checking if the AST of the candidate API call is a sub-tree of the reference API call reveals which API is being used in the dataset.

Identifying and even defining hallucinations can be challenging. We use the AST matching process to directly identify the hallucinations. We define a hallucination as an API call that is not a sub-tree of any API in the database – invoking an entirely imagined tool. This form of hallucination is distinct from invoking an API incorrectly which we instead define as an error.

AST Sub-Tree Matching We perform AST sub-tree matching to identify which API in our dataset is the LLM calling. Since each API call can have many arguments, we need to match on each of these arguments. Further, since, Python allows for default arguments, for each API, we define which arguments to match in our database. For example, we check `repo_or_dir` and `model` arguments in our function call. In this way, we can easily check if the argument matches the reference API or not. Please refer to Fig. 4 for more details. In this example, Gorilla returns a torch API call. We first build the tree, and verify that it matches a subtree in our dataset along nodes `torch.hub.load`, `pytorch/vision`, and `densenet121`. But, we don't check for match along leaf node `pretrained = True` since that is an optional python argument.



```
torch.hub.load('pytorch/vision:v0.10.0',
      'densenet121', pretrained=True)
```

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.

4 Evaluation

We carried out an array of experiments on our collected dataset, benchmarking our model Gorilla with other models, and exploring how different retrieval methods may impact the performance of the model in making API calls. We then demonstrate that Gorilla can easily adapt to test-time changes in API documentation. In addition, we assess Gorilla's ability to reason about API calls under constraints. Lastly, we examined how integrating different retrieval methods during training influences the model's final performance.

Baselines Primarily, we compare Gorilla with state-of-the-art language models in a zero-shot setting. The models under consideration include: GPT-4 by OpenAI, we use the `gpt-4-0314` checkpoint; GPT-3.5-turbo with the `gpt-3.5-turbo-0301` checkpoint, both of which are RLHF-tuned model specifically designed for conversation; Claude with `claude-v1` checkpoint, a language model by Anthropic, renowned for its lengthy context capabilities; LLaMA-7B, a large language model by Meta and the finest open-source model to date.

Retrievers The term *Zero-shot* (abbreviated as 0-shot in tables) refers to scenarios where no retriever is used. The sole input to the model is the user's natural language prompt. For `BM25`, we consider each API as a separate document. During retrieval, we use the user's query to search the index and fetch the most relevant (top-1) API. This API is concatenated with the user's prompt to query the LLMs. Similarly, GPT-Index refers to the retrieval model `text-davinci-003` from OpenAI. Like BM25, each API call is indexed as an individual document, and the most relevant document, given a user query, is retrieved and appended to the user prompt. Lastly, we include an Oracle retriever, which serves two purposes: first, to identify the potential for performance improvement through more efficient retrievers, and second, to assist users who know which API to use but may need to help invoking it. In all cases, when a retriever is used, it is appended to the user's prompt as follows: `<user_prompt> Use this API documentation for reference: <retrieved_API_doc_JSON>`. The dataset for these evaluations is detailed in Sec 3. We emphasize that we have maintained a holdout test set on which we report our findings. The holdout test set was created by dividing the self-instruct dataset's instruction, API pairs into training and testing sets.

4.1 AST Accuracy on API call

We first demonstrate the results for the AST accuracy for different models. We present the results in Tab. 1. We test each model for different retriever settings defined above. We report the overall accuracy, the error by hallucination and the error by selecting wrong API call. Note that for TorchHub and TensorHub, we evaluate all the models using AST tree accuracy score. However, for HuggingFace,

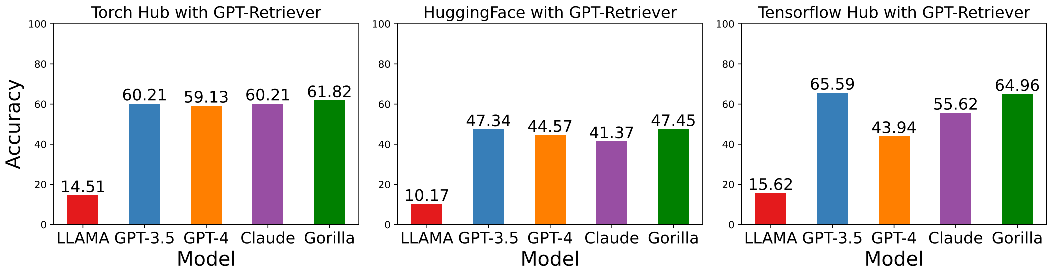


Figure 5: **Accuracy with GPT-retriever.** Gorilla outperforms on Torch Hub and Hugging-Face while matching performance on Tensorflow Hub for all existing SoTA LLMs - closed source, and open source.

since the dataset is not exhaustive, for all the models except Gorilla, we only check if they can provide the correct domain names. So this problem reduces to picking one of the multiple choices.

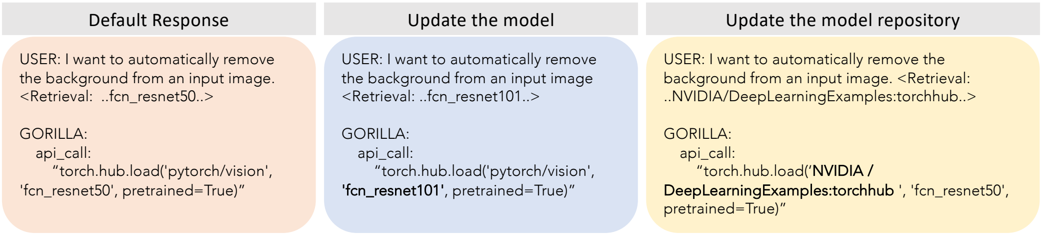
Finetuning without Retrieval In Tab. 1 we show that lightly fine-tuned Gorilla gets the state-of-the-art performance zero-shot over all the models, 20.43% better than GPT-4 and 10.75% better than ChatGPT. When compared to other open-source models LLAMA, the improvement is as big as 83%. his suggests quantitatively, that finetuning is better than retrieval, at-least in our scope.

In addition, we found that finetuning without retriever and putting ground truth retriever in evaluation time rarely helps the performance: 0.88% worse in TensorHub and 0.97% better in HuggingFace. If we put BM25 or GPT-Index as retriever, results will be significantly dropped: 21.50% in Torch Hub and 47.57% in HuggingFace. The result illustrates that adding a non-optimal retriever at test time will sometime misguide the model and result in more errors. We will discuss an interesting ablation on how finetuning with the retriever will help the performance in the next paragraph.

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

Finetuning with Retrieval We now discuss an interesting experiment on how finetuning language with retriever incorporated is helping the performance. The settings for this experiment are finetuning the base LLAMA with the prompt (instruction generated), reference API document (from golden-truth oracle), and the example output generated by GPT-4. In Tab. 2, we can see that incorporating ground truth retriever in the finetuning pipeline achieves significantly better results 12.37% better than training without retriever in Torch Hub and 23.46% better in HuggingFace. However, we found that at evaluation time, current retrievers still have a big gap between the ground truth retriever: using GPT-Index at evaluation results in 29.20% accuracy degradation, and using BM25 results in a 52.27% accuracy degradation. Nevertheless, we can still conclude that with a better retriever, finetuning



USER: I want to automatically remove the background from an input image. <Retrieval: ..fcn_resnet50..>

GORILLA:
api_call:
"torch.hub.load('pytorch/vision', 'fcn_resnet50', pretrained=True)"

USER: I want to automatically remove the background from an input image <Retrieval: ..fcn_resnet101..>

GORILLA:
api_call:
"torch.hub.load('pytorch/vision', **'fcn_resnet101'**, pretrained=True)"

USER: I want to automatically remove the background from an input image. <Retrieval: ..NVIDIA/DeepLearningExamples:torchhub..>

GORILLA:
api_call:
"torch.hub.load('**NVIDIA / DeepLearningExamples:torchhub** ', 'fcn_resnet50', pretrained=True)"

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`

with retriever is still a better method to adopt while in another scenario, when a good retriever is not available, zero-shot finetuning might be the preferred choice.

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

Hallucination with LLM One phenomenon we observe is that zero-shot prompting with LLMs (GPT-4/GPT-3.5) to call APIs results in dire hallucination errors. These errors, while diverse, commonly manifest in erroneous behavior such as the model invoking the "AutoModel.from_pretrained(dir_name)" command with arbitrary GitHub repository names. Surprisingly, we also found that in TorchHub, HuggingFace and TensorFlow Hub, GPT-3.5 has less hallucination errors than GPT-4. This finding is also consistent for the settings when various retrieving methods are provided: 0-shot, BM25, GPT-Index and the oracle. This might suggest that RLHF plays a central role in turning the model to be truthful. Additional examples and discussion are in Appendix.

4.2 Test-Time Documentation Change

The rapidly evolving nature of API documentation presents a significant challenge for the application of LLMs in this field. These documents are often updated at a frequency that outpaces the retraining or fine-tuning schedule of LLMs, making these models particularly brittle to changes in the information they are designed to process. This mismatch in update frequency can lead to a decline in the utility and reliability of LLMs over time.

However, with the introduction of Gorilla's retriever-aware training, we can readily adapt to changes in API documentation. This novel approach allows the model to remain updated and relevant, even as the API documentation it relies on undergoes modifications. This is a pivotal advancement in the field, as it ensures that the LLM maintains its efficacy and accuracy over time, providing reliable outputs irrespective of changes in the underlying documentation.

For instance, consider the scenario illustrated in Figure 6, where the training of Gorilla has allowed it to react effectively to changes in APIs. This includes alterations such as upgrading the FCN's ResNet-50 backbone to ResNet-101, as demonstrated in the second column of the figure. This capability ensures that the LLM remains relevant and accurate even as the underlying models and systems undergo upgrades and improvements. Furthermore, the third column in Figure 6 shows how Gorilla adapts to changes in the model registry from `pytorch/vision` to `NVIDIA/DeepLearningExamples:torchhub`. This reflects the model's ability to adjust to shifts in API sources, which is vital as organizations may change their preferred model registries over time.

In summary, Gorilla's ability to adapt to test-time changes in API documentation offers numerous benefits. It maintains its accuracy and relevance over time, adapts to the rapid pace of updates in

API documentation, and adjusts to modifications in underlying models and systems. This makes it a robust and reliable tool for API calls, significantly enhancing its practical utility.

4.3 API Call with Constraints

We now focus on the language model's capability of understanding constraints. For any given task, which API call to invoke is typically a tradeoff between a multitude of factors. In the case of RESTFul APIs, it could be the cost of each invocation ($), and the latency of response (ms), among others. Similarly, within the scope of ML APIs, it is desirable for Gorilla to respect constraints such as accuracy, number of learnable parameters in the model, the size on disk, peak memory consumption, FLOPS, etc. We present the underlying ablation study evaluating the ability of different models in zero-shot and with retrievers settings to respect a given accuracy constraint. This setting is best understood with an example. If the user were to ask for an Image classification model that achieves at least 80% top-1 accuracy on the Imagenet dataset, then while both are classification models hosted by Torch Hub, `ResNeXt-101 32x16d` with a top-1 accuracy of 84.2% would be the right model whose API to call and not, say, `MobileNetV2` which has a top-1 accuracy of 71.88%.

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**

For Table 3, we filtered a subset of the Torch Hub dataset that had accuracy defined for at least one-dataset in its model card (65.26% of TorchHub dataset in Table 1). We notice that with constraints, understandably, the accuracy drops across all models, with and without a retriever. Gorilla is able to match performance with the best-performing model GPT-3.5 when using retrievals (BM25, GPT-Index) and has the highest accuracy in the Zero-shot case. This highlights Gorilla's ability to navigate APIs while considering the trade-offs between different constraints.

5 Conclusion

LLMs are swiftly gaining popularity across diverse domains. In our study, we spotlight techniques designed to enhance the LLM's ability to accurately identify the appropriate API for a specific task—a significant but often overlooked aspect in the advancement of this technology. Since APIs function as a universal language enabling diverse systems to communicate effectively, their correct usage can boost the ability of LLMs to interact with tools in the wider world. In this paper, we propose Gorilla, a new novel pipeline for finetuning LLMs to call APIs. The finetuned model's performance surpasses prompting the state-of-the-art LLM (GPT-4) in three massive datasets we collected. Gorilla generates reliable API calls to ML models without hallucination, demonstrates an impressive capability to adapt to test-time API usage changes, and can satisfy constraints while picking APIs.

6 Limitations & Social Impacts

With the goal of wanting to have a challenging dataset, we chose ML APIs, given their functional similarity. The potential downside to APIs that focus on the ML domain, is their propensity to produce biased predictions if trained on skewed data, potentially disadvantaging certain sub-groups. To counter this concern and foster a deeper understanding of these APIs, we are releasing our extensive dataset, consisting of over 11,000 instruction-API pairs. This resource will serve the wider community as a valuable tool for studying and benchmarking existing APIs, contributing to a more fair and optimized usage of machine learning.

7 Acknowledgement

This research is supported in part by gifts to UC Berkley Sky Computing Lab from Astronomer, Google, IBM, Intel, Lacework, Microsoft, Nexla, Samsung SDS, Uber, and VMware.

References

[1] Ahn, M., Brohan, A., Brown, N., Chebotar, Y., Cortes, O., David, B., Finn, C., Gopalakrishnan, K., Hausman, K., Herzog, A., et al. (2022). Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*.

[2] Andor, D., He, L., Lee, K., and Pitler, E. (2019). Giving bert a calculator: Finding operations and arguments with reading comprehension. *arXiv preprint arXiv:1909.00109*.

[3] Anthropic, h.-c. (2022). Claude.

[4] Bavishi, R., Lemieux, C., Fox, R., Sen, K., and Stoica, I. (2019). Autopandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27.

[5] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

[6] Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y. T., Li, Y., Lundberg, S., et al. (2023). Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*.

[7] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

[8] Chen, X., Lin, M., Schärli, N., and Zhou, D. (2023). Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

[9] Chiang, W.-L., Li, Z., Lin, Z., Sheng, Y., Wu, Z., Zhang, H., Zheng, L., Zhuang, S., Zhuang, Y., Gonzalez, J. E., Stoica, I., and Xing, E. P. (2023). Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality.

[10] Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. (2022). Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

[11] Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., Li, E., Wang, X., Dehghani, M., Brahma, S., et al. (2022). Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*.

[12] Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., et al. (2021). Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

[13] Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.-r., and Kohli, P. (2017). Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR.

[14] Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., and Neubig, G. (2022). Pal: Program-aided language models. *arXiv preprint arXiv:2211.10435*.

[15] Iyer, S., Lin, X. V., Pasunuru, R., Mihaylov, T., Simig, D., Yu, P., Shuster, K., Wang, T., Liu, Q., Koura, P. S., et al. (2022). Opt-iml: Scaling language model instruction meta learning through the lens of generalization. *arXiv preprint arXiv:2212.12017*.

Torch Hub domain names: Classification, Semantic Segmentation, Object Detection, Audio Separation, Video Classification, Text-to-Speech

Tensor Hub domain names: text-sequence-alignment, text-embedding, text-language-model, text-preprocessing, text-classification, text-generation, text-question-answering, text-retrieval-question-answering, text-segmentation, text-to-mel, image-classification, image-feature-vector, image-object-detection, image-segmentation, image-generator, image-pose-detection, image-rnn-agent, image-augmentation, image-classifier, image-style-transfer, image-aesthetic-quality, image-depth-estimation, image-super-resolution, image-deblurring, image-extrapolation, image-text-recognition, image-dehazing, image-deraining, image-enhancemenmt, image-classification-logits, image-frame-interpolation, image-text-detection, image-denoising, image-others, video-classification, video-feature-extraction, video-generation, video-audio-text, video-text, audio-embedding, audio-event-classification, audio-command-detection, audio-paralinguists-classification, audio-speech-to-text, audio-speech-synthesis, audio-synthesis, audio-pitch-extraction

HuggingFace domain names: Multimodal Feature Extraction, Multimodal Text-to-Image, Multimodal Image-to-Text, Multimodal Text-to-Video, Multimodal Visual Question Answering, Multimodal Document Question Answer, Multimodal Graph Machine Learning, Computer Vision Depth Estimation, Computer Vision Image Classification, Computer Vision Object Detection, Computer Vision Image Segmentation, Computer Vision Image-to-Image, Computer Vision Unconditional Image Generation, Computer Vision Video Classification, Computer Vision Zero-Shor Image Classification, Natural Language Processing Text Classification, Natural Language Processing Token Classification, Natural Language Processing Table Question Answering, Natural Language Processing Question Answering, Natural Language Processing Zero-Shot Classification, Natural Language Processing Translation, Natural Language Processing Summarization, Natural Language Processing Conversational, Natural Language Processing Text Generation, Natural Language Processing Fill-Mask, Natural Language Processing Text2Text Generation, Natural Language Processing Sentence Similarity, Audio Text-to-Speech, Audio Automatic Speech Recognition, Audio Audio-to-Audio, Audio Audio Classification, Audio Voice Activity Detection, Tabular Tabular Classification, Tabular Tabular Regression, Reinforcement Learning Reinforcement Learning, Reinforcement Learning Robotics

Figure 7: **Domain names**: Domain names with the three dataset. Tensor Hub is the smallest dataset while the other two hubs contain many more models.

Table 4: Hyperparameters for training Gorilla

Hyperparameter Name	Value
learning rate	2e-5
batch size	64
epochs	5
warmup ratio	0.03
weight decay	0
max seq length	2048

8.3 Performance Comparison

We provide a full comparison of each model's performance in this section. In Fig 10 and Fig. 11, the full set of comparisons is provided. We see that especially in zero-shot case, Gorilla surpasses the GPT-4 and GPT-3.5 by a large margin. The GPT-4 and GPT-3.5 gets around 40% accuracy in Torch Hub and Tensor Hub, which are two structured API calls. Compared to that, HuggingFace is a more flexible and diverse Hub, as a result, the performance on HuggingFace is not as competitive.

### User: Our customer is a zoo and we want to help them detect movement of different animals. Write a python program in 1 to 2 lines to call API in TensorFlowHub.
### Assistant:
<domain>: image-object-detection,
<api_call>: detector = hub.load('https://tfhub.dev/google/openimages_v4/ssd_mobilenet_v2/1'),
<api_provider>: TensorFlow Hub,
<explanation>: This API call loads the pre-trained Single Shot Detector (SSD) MobileNet V2 model from TensorFlow Hub, which has been trained on the Open Images v4 dataset. This model is used for object detection in images. Once loaded, you can pass an image (and an optional score threshold for detection confidence) to the detector to get back the detected objects along with their bounding boxes, class labels, and associated confidence scores.
<code>: import tensorflow as tf; import tensorflow_hub as hub; # Load the SSD MobileNet V2 model; detector = hub.load('https://tfhub.dev/google/openimages_v4/ssd_mobilenet_v2/1'); # Use the model to detect objects in an image (assuming 'image' is a TensorFlow tensor representing your image); result = detector(image, score_threshold=0.5)

### User: I am an engineer at Uber and I need to find a API that can classify pedestrians, cars, etc. from an image of the scene. Write a python program in 1 to 2 lines to call API in TorchHub.
<Reference API>: {domain: Object Detection, framework: PyTorch, functionality: Traffic Object Detection, Drivable Area Segmentation, Lane Detection, api_name: HybridNets, api_call: torch.hub.load(repo_or_dir='datvuthanh/hybridnets', model='hybridnets', pretrained=True), api_arguments: repo_or_dir, model, pretrained, python_environment_requirements: Python>=3.7, PyTorch>=1.10, example_code: 'import torch; model = torch.hub.load('datvuthanh/hybridnets', 'hybridnets', pretrained=True); img = torch.randn(1,3,640,384); features, regression, classification, anchors, segmentation = model(img), performance: dataset: [name: BDD100K, accuracy: Traffic Object Detection: Recall (%): 92.8, mAP@0.5 (%): 77.3, Drivable Area Segmentation: Drivable mIoU (%): 90.5, Lane Line Detection: Accuracy (%): 85.4, Lane Line IoU (%): 31.6], description: HybridNets is an end2end perception network for multi-tasks. Our work focused on traffic object detection, drivable area segmentation and lane detection. HybridNets can run real-time on embedded systems, and obtains SOTA Object Detection, Lane Detection on BDD100K Dataset.}
### Assistant:
<domain>: Object Detection, **<api_call>**: model = torch.hub.load('datvuthanh/hybridnets', 'hybridnets', pretrained=True),
<api_provider>: PyTorch,
<explanation>: Load the pretrained HybridNets model for object detection from PyTorch Hub to classify pedestrians, cars, etc. in an image of the scene,
<code>: import torch; model = torch.hub.load('datvuthanh/hybridnets', 'hybridnets', pretrained=True)

Figure 8: **Example of the Dataset**: Two examples of the dataset, the above one is zero-shot (without information retrievers) and the bottom one is with information retriever.

```
generate_video = pipeline("text-to-video", model="your_model_name")
```

```
vqa = pipeline("visual-question-answering", model="microsoft/clip-vqa-base",
tokenizer="microsoft/clip-vqa-base")
```

```
depth_estimator = pipeline("depth-estimation", model="intel-isl/MiDaS", tokenizer="intel-
isl/MiDaS")
```

Figure 9: **Hallucination Examples**: GPT-4 incurs serious hallucination errors in HuggingFace. We show a couple of examples in the figure.

8.3.1 Evaluation

For ease of evaluation, we manually cleaned up the dataset to make sure each API call domain only contains the valid call in the form of:

API_name(API_arg_1, API_arg_2, ..., API_arg_k)

Our framework allows the user to define any combination of the arguments to check. For Torch Hub, we check for the API name `torch.hub.load` with arguments `repo_or_dir` and `model`. For Tensor Hub, we check API name `hub.KerasLayer` and `hub.load` with argument `handle`. For HuggingFace, since there are many API function names, we don't list all of them here. One specific note is that we require the `pretrained_model_name_or_path` argument for all the calls except for `pipeline`. For `pipeline`, we don't require the `pretrained_model_name_or_path` argument since it automatically select a model for you once `task` is specified.

8.3.2 Hallucination

We found especially in HuggingFace, the GPT-4 model incurs serious hallucination problems. It would sometimes put a GitHub name that is not associated with the HuggingFace repository in to the domain of `pretrained_model_name_or_path`. Fig. 9 demonstrates some examples and we also observe that GPT-4 sometimes assumes the user have a local path to the model like `your_model_name`. This is greatly reduced by Gorilla as we see the hallucination error comparison in Tab. 1.

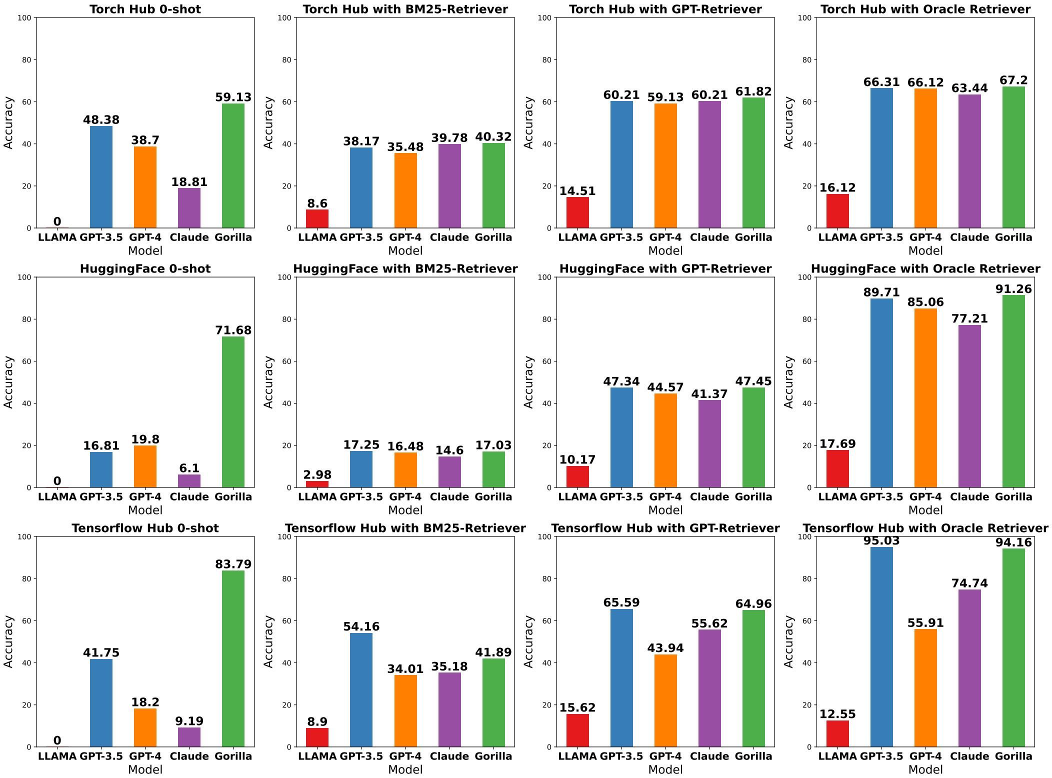


Figure 10: **Performance**: We plot each model's performance on different configurations. We see that Gorilla performs extremely well in the zero-shot setting. While even when the oracle answer is given, Gorilla is still the best.

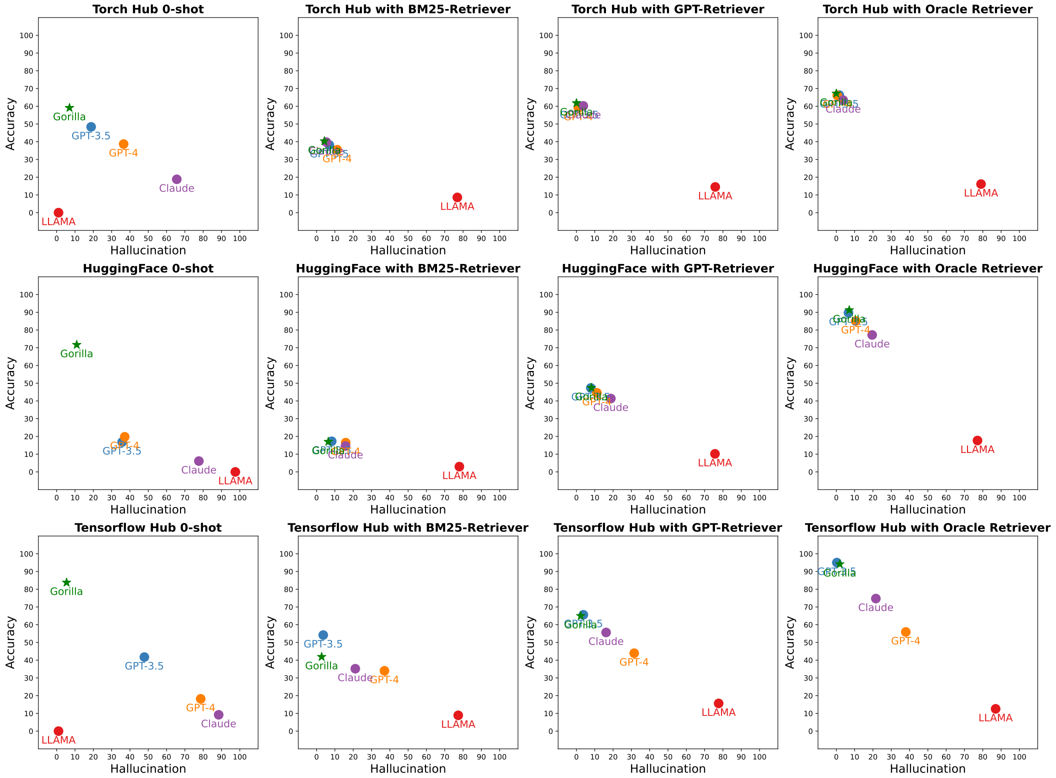


Figure 11: **Accuracy vs Hallucination**: We plot each model's performance on different configurations. We found that in the zero-shot setting, Gorilla has the most accuracy gain while maintaining good factual capability. When prompting with different retrievers, Gorilla is still capable to avoid the hallucination errors.