Title

Teaching Large Language Models to Use Tools at Scale

Author

Patil, Shishir Girishkumar

Publication Date

2024

Peer reviewed|Thesis/dissertation

Teaching Large Language Models to Use Tools at Scale

By

Shishir Girishkumar Patil

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Joseph Edgar Gonzalez, Co-chair
Professor Prabal Dutta, Co-chair
Professor Ion Stoica
Professor Christopher Ré

Spring 2024

Teaching Large Language Models to Use Tools at Scale

Abstract

Teaching Large Language Models to Use Tools at Scale

by

Shishir Girishkumar Patil

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph Edgar Gonzalez, Co-chair

Professor Prabal Dutta, Co-chair

Large language models (LLMs) have shown impressive advancements in many complex tasks such as mathematical reasoning and program synthesis. Despite this progress, the ability of LLMs to effectively utilize tools, services, and applications remains limited. In order to address this gap, we first introduce Gorilla LLM, a finetuning recipe that enhances the ability of LLMs to use tools by invoking APIs. Gorilla also introduces abstract syntax tree (AST)-based metrics to evaluate API hallucination in LLMs. Further, recognizing that evaluating LLMs can be challenging, we develop OpenFunctions, a pre-trained model that does not require retraining and instead relies on retrieval-augmented generation (RAG) to surface relevant APIs. This system allows LLMs to access an updated repository of functions and services, improving their utility without the overhead of constant model retraining.

Complementing function calling, RAFT (Retrieval Augmented Fine Tuning) provides a recipe for embedding new domain-specific knowledge into models. By training LLMs to discern and utilize only relevant information from a set of retrieved documents, RAFT improves accuracy and reliability in "open-book" settings across various in-domain datasets.

Finally, to enable the autonomous execution of LLM-generated commands—which can be prone to errors—the Gorilla Execution Engine (GoEx) is a novel runtime system that enforces execution under least privilege by dynamically interpreting user intentions and also incorporates "undo" and "damage confinement" abstractions to mitigate risks. GoEx supports post-facto validation, allowing users to verify the correctness of actions after they are executed and to revert any undesired effects. GoEx enables LLMs to operate autonomously, significantly reducing the potential risks associated with their autonomous actions.

We believe that together, these developments—Gorilla, OpenFunctions, RAFT, and GoEx—are critical to unlocking the potential for LLM agents to interact with applications and services.

To my Parents.

Contents

List of Figures

List of Tables

Acknowledgments

This dissertation stands as a testament to the exceptional mentorship, advising, and guidance I have received throughout my education. It is an impossible task to adequately express my gratitude in words, for so significant has been your role in my life, but I will nonetheless attempt to do so.

I begin by expressing my deepest gratitude to Professors Joseph E. Gonzalez, Prabal Dutta, and Ion Stoica, who have been my advisors at the University of California at Berkeley. Joseph (Joey) Gonzalez has been an incredibly positive influence in my life. As I explored various interests during my PhD—from edge computing, to networks and security, to Large Language Models (LLMs)—he always met me with great enthusiasm. Joey has been always approachable throughout my PhD journey. A collaborator once remarked that my conversations with Joey resembled those with a friend rather than an advisor! Prabal and I first met during the visit days, and I remember walking out of his room knowing he would give me the freedom to pursue everything I wanted. I have had the pleasure of enjoying this for the past five years! In Ion, I found not only a great mentor but also an invaluable supporter. Ion taught me the importance of focus and the merit of consistently producing quality work. His emphasis on single-minded focus has yielded significant dividends, not least this dissertation. At Berkeley, I also have had the great pleasure of collaborating with Raluca Ada Popa, from whom I learned how to identify compelling academic problems; Sylvia Ratnasamy, who gave me the warmest welcome at Berkeley, with whom I taught Advanced Computer Networks, and collaborated on a fun congestion control project; Matei Zaharia, who enhanced our projects with his rigorous knowledge of the field, and also helped brainstorm the idea for this dissertation, and Kurt Keutzer, who convinced me that Berkeley was the best fit. I am also grateful to Chris Ré for serving on my committee, sharing his insights on tackling important problems, and mentoring me about my career during our long walks.

My research career blossomed at Berkeley, but the seeds were sown at Microsoft Research India (MSR). I had the honor and sheer joy of being mentored by Vivek Seshadri, Prateek Jain, and Harsha Vardhan Simhadri. Vivek was instrumental in getting me to Microsoft Research, and I am forever indebted for the trust he placed in me. It is my good fortune to call Vivek my first manager at a full-time job. As a cherry on top, I will always be his first report. Harsha pushed me to become a well-rounded researcher. Although he was an established researcher and I was still an undergraduate, he treated me as an equal. He insisted that I control my time, fostering the independent thinker I am today. At MSR, I approached Prateek both when things went well and when they did not. Working with Prateek, I had the opportunity to learn from his ability to discern signal from noise, identify concrete abstractions from nebulous problem statements, and systematically address them. On a more personal note, Prateek always prioritized my well-being and interests, even above his own. I remember a night in December when he sat in my hotel room in Redmond, helping me with my statement of purpose for graduate school, despite having a critical presentation to Microsoft's leadership scheduled the next morning. I would also like to thank Victor Bahl,

Ranveer Chandra, Ashish Kapoor, Venkat Padmanabhan, Sriram Rajamani, Ramachandran Ramjee, and Muthian Sivathanu, all of whom I continue to seek for advice.

My first foray into research was with Professor Manjunatha M Nayak (M. M. Nayak) at the Indian Institute for Science, Center for Nano Science and Engineering. Over more than a year and a half of collaboration, he encouraged me to aim higher, be fearless, and above all, taught me how enjoyable learning can be. I am also grateful to Namita Palecha, Ravishankar S, and Yendluri Gopala Rao at RV College of Engineering for their valuable advice and constant availability.

I would also like to acknowledge my fellow PhD collaborators at Berkeley. I have enjoyed my deepest collaboration with Tianjun Zhang. Tianjun is a delight to work with. Across multiple nights that turned into days, he always entertained me with his wit and unique take on English. With Tianjun, I never had to double-check—what was discussed was done. Jean-Luc Watson was my first collaborator and friend at Berkeley. He introduced me to campus life, shared the joy of skiing with me, and together we learned how to sail. Vivian Fang has impressively diverse skills, including designing logos for projects, coordinating ski trips, understanding LLMs, and of course, security. I am also thankful to my other collaborators, Tess Despres, Paras Jain, Xiuyu Li, Kevin Lin, Charles Packer, Alvin Tan, Sijun Tan, and Sarah Wooders. Additionally, I am grateful for the opportunity to mentor numerous undergraduates—Charlie Cheng-Jie Ji, Huanzhi Mao, Fanjia Yan, Noppapon Chalermchockcharoenkit, Roy Huang, Aaron Hao, Raman Varma, Pranav Ramesh, Sai Kolasani, Arth Bohra, Kai Wen Hao, Arth Bohra, Tim Barat, Riley Lyman, and many others who rank among Berkeley's finest.

PhD studies can be a solitary journey, but that was not my experience. I am thankful to the 3D Vision reading group that I ran alongside Charles Packer, Justin Wong, and Vivian Fang, which not only educated the lab about LLMs but also provided dinner. I relished every lunch spent debating with Sukrit Kalra, Jiwon Park, and Tianjun Zhang. Branden Ghena and Meghan Clark made Berkeley feel like home. Sameer Wagh taught me an invaluable skill—how to make sound decisions—while generously providing meals throughout the two years he was here. Qijin Huang has been part of the Gorilla journey, and in whose car many a memories were made. Additionally, I am grateful to Vivek Aithal, Romil Bhardwaj, Shiyi Cao, Audrey Cheng, Wei-Lin Chiang, David Chu, Emma Dauterman, Lisa Dunlap, Guangyu Feng, Laurenz Heppding, Paras Jain, Darya Kaviani, Noah Klugman, Woosuk Kwon, Shadaj Laddad, Dacheng Li, Zhuohan Li, Fangchen Liu, Lily Liu, Frank Luan, Michael Luo, Mae Milano, Simon Mo, Micah Murray, Pat Panuto, Nathan Pemberton, Suzie Petryk, Julien Piet, Conor Power, Laura Power, Daniel Rothchild, Peter Schafhalter, Shangyin Tan, Suhas B U, Stephanie Wang, Zhanghao Wu, Samyu Yagati, Zhongheng Yang, Wen Zhang, Lianmin Zheng, and Siyuan Zhuang for their support.

During my PhD, there were periods when I found myself spending more time in the South Bay than at Berkeley. Sundararajan Renganathan has been a steadfast friend throughout my PhD, sharing in both my highs and lows. Prakhar Agarwal and Sahil Rishi welcomed me into their home for several weeks at a time, mostly spent gaming. Kaushal Yagnik and Aesha Shah ensured I never felt homesick. I also deeply value the time spent with Kumar Ayush and

Shreya Singh, who deserve special mention for their generous hospitality. Shashank Bhushan, Haroun Habeeb, Ashwin Kanahre, Kopal Nihar, Prabhat, Nishant Rai, Abhishek Sinha, and Shweta Soni also provided significant support during this time. Interestingly, although both had graduated from Berkeley by the time we met, Rohan Bhavishi and Alvin Wan became close friends and co-conspirators.

This PhD thesis is a culmination of my time in school, and my parents have been the biggest champions through all of it. They have instilled in me the importance of working hard, striving for excellence, and always doing my best. They have always given me their unconditional support even at the expense of significant sacrifices in their own lives. To my mother *(amma)* and father *(appaji)*, I dedicate this dissertation.

Chapter 1

Introduction

1.1 Background and motivation

Large language models (LLMs) have rode an impressive wave of advances, with models now excelling in a variety of tasks, such as mathematical reasoning and program synthesis. However, despite these significant strides, one area where LLMs have struggled is in effectively utilizing external tools and services, which are crucial for expanding their applicability beyond just knowledge retrievers to decision making agents.

APIs present a unified modality to teach LLMs to use external tools and services. However, this is a challenging task even for today's state-of-the-art LLMs largely due to their unawareness of what APIs are available and how to use them in a frequently updated tool set. We develop Gorilla (Chapter 2), a finetuned LLaMA [139] 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

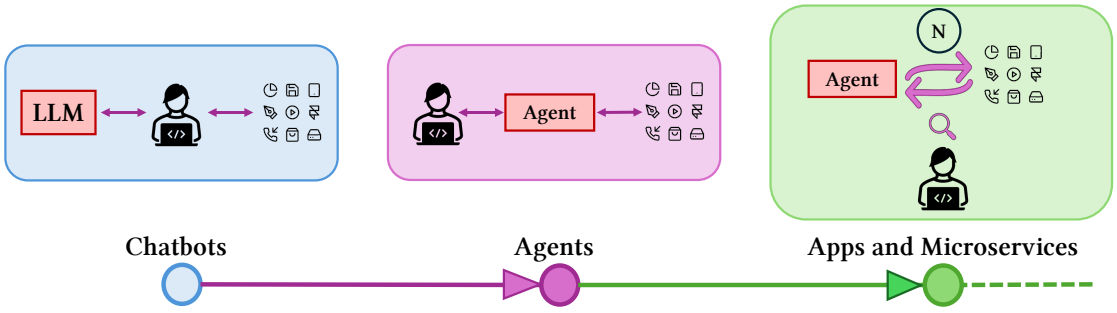

Figure 1.1: **Evolution of LLMs powered applications and services.** From chatbots, to decision-making agents that can interact with applications and services with human-supervision, to autonomous LLM-agents interacting with LLM-powered apps and services with minimal and punctuated human supervision.

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. Complementing Gorilla, we present OpenFunctions LLM, which bypasses the need for frequent retraining of LLMs by integrating retrieval-augemented generation (RAG) [74]. RAG surfaces the top few relevant APIs for a given user intent, and then OpenFunctions returns the right API call with the arguments filled in.

As an improvement over existing RAG systems, we introduce Retrieval Augmented Fine Tuning (RAFT) in Chapter 3. Pretraining LLMs on large corpora of textual data is now a standard paradigm. When using these LLMs for many downstream applications, including and especially for API calls, it is common to additionally bake in new knowledge (e.g., time-critical news, or private domain knowledge) into the pre-trained model either through RAG-based prompting, or fine-tuning. However, the optimal methodology for the model to gain such new knowledge remains an open question. RAFT is a training recipe that improves the model's ability to answer questions in an "open-book" in-domain setting. In RAFT, given a question, and a set of retrieved documents, we train the model to ignore documents that don't help in answering the question, which we call "distractor documents". RAFT accomplishes this by citing verbatim the right sequence from the relevant document that would help answer the question. This, coupled with RAFT's chain-of-thought-style response, helps improve the model's ability to reason. In domain-specific RAG, RAFT consistently improves the model's performance across PubMed, HotpotQA, and Gorilla datasets, presenting a post-training recipe to improve pre-trained LLMs to in-domain RAG.

Finally, in Chapter 4, we introduce the Gorilla Execution Engine (GoEx), a novel runtime that executes LLM generated API calls. GoEx enables LLM-powered applications to operate on user accounts while confining potential damage from unreliable LLMs. We argue that in many cases, "post-facto validation"—verifying the correctness of a proposed action after seeing the output—is much easier than the aforementioned "pre-facto validation" setting. The core concept behind enabling a post-facto validation system is the integration of an intuitive undo feature, and establishing a damage confinement for the LLM-generated actions as effective strategies to mitigate the associated risks. GoEx also draws inspiration from classical systems security design principles, such as enforcing least privilege, separating user credentials from the LLM provider, and treating the user as the ground truth for permissions. In order to map a given user query into a set of functions and permission-scopes—a seemingly intractable task—we counter-intuitively decide to use an LLM to do so. GoEx accesses credentials only through the runtime when executing LLM actions, and obtains user approval before granting permissions. GoEx is evaluated on a dataset of over 10,000 APIs across 250 services, demonstrating that it can enforce least privilege with only 0.5% of cases resulting in excessive permissions, even if users approve every permission request. The additional latency from GoEx ranges from 0.1% to 22%. GoEx enables autonomous and safer integration of

LLMs into user-facing applications across diverse domains.

Together, Gorilla, OpenFunctions, RAFT, and GoEx introduce techniques to teach LLMs the ability to use external tools and services through API calls. Recipes to train the LLM, coupled with the underlying systems to backstop unintended consequences, broaden the scope of LLM applications and ensure that these models can be deployed responsibly and effectively in diverse environments.

These projects have had significant impact academically and commercially. Gorilla introduced and pioneered the idea of teaching LLMs to use tools through APIs. Gorilla and OpenFunctions are being used in hundreds of enterprises, been downloaded tens of thousands of times, our hosted inference servers have served over $800,000+$ user requests. We will discuss some of our learnings from our deployment experience and conclude in Chapter 5.

Chapter 2

Gorilla

2.1 Introduction

The use of APIs and LLMs [1, 15, 16, 29, 100, 120] has changed what it means to program. Previously, building complex machine learning software and systems required extensive time and specialized skills. Now with tools like the HuggingFace API, an engineer can set up a deep learning pipeline with a few lines of code. Instead of searching through StackOverflow and documentation, developers can ask models like ChatGPT for solutions and receive immediate, actionable code with docstrings. However, using off-the-shelf LLM's to generate API calls remains unsolved because there are millions of available API's which are frequently updated.

We connect LLMs and massive APIs with Gorilla, a system which takes in an instruction like "build me a classifier for medical images" and outputs the corresponding imports and API calls, along with a step-by-step explanation of the pipeline. Gorilla uses 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. The model trained with Gorilla demonstrates good adaptation behavior to test-time API document changes such as API version and arguments.

With the development of API generation methods comes a question of how to evaluate, as many APIs have overlapping functionality with nuanced limitations and constraints. Thus, we construct APIBench ($\sim$ 1600 APIs) by scraping a large corpus of ML APIs and developing an evaluation framework that uses AST sub-tree matching to check functional correctness.

Using APIBench, we finetune Gorilla, a LLaMA-7B-based model with document retrieval, and show that it significantly outperforms both open-source and closed-source models like Claude and GPT-4 in terms of API functionality accuracy as well as a reduction in API argument hallucination errors (Fig. 2.1). Finally, we demonstrate Gorilla's ability to understand and reason about constraints.

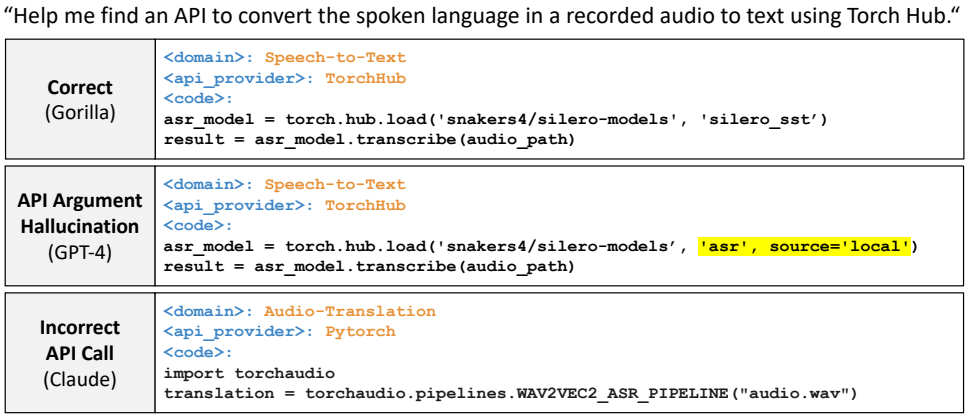

Figure 2.1: **Examples of API calls**. Example API calls generated by GPT-4 [1], Claude [8], and Gorilla for the given prompt. In this example, GPT-4 presents a model that doesn't exist, and Claude picks an incorrect library. In contrast, our Gorilla model can identify the task correctly and suggest a fully-qualified API call.

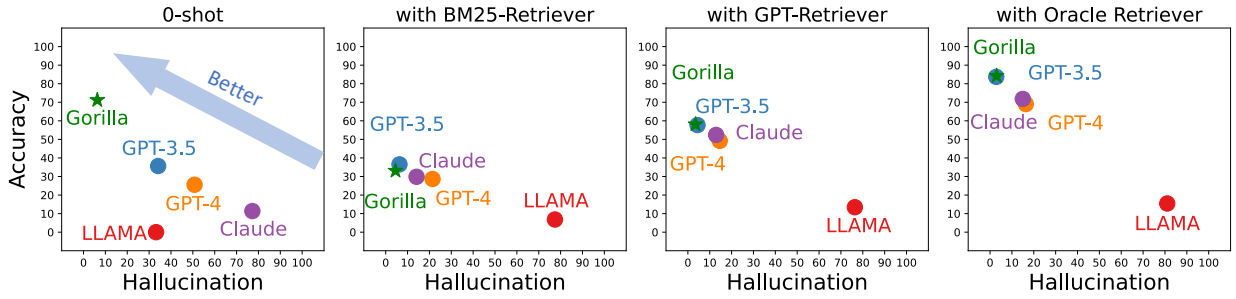

Figure 2.2: **Accuracy (vs) hallucination** in four settings, that is, *zero-shot* (i.e., without any retriever), and *with retrievers*. `BM25` and `GPT` are commonly used retrievers and the `oracle` retriever returns relevant documents at 100%, indicating an upper bound. Higher in the graph (higher accuracy) and to the left is better (lower hallucination). Across the entire dataset, our model, Gorilla, improves accuracy while reducing hallucination.

2.2 Related Work

By empowering LLMs to use tools [121], we can grant access to vastly larger and changing knowledge bases and accomplish complex computational tasks. By providing access to search technologies and databases, [96, 130, 138] demonstrated that we can augment LLMs to address a significantly larger and more dynamic knowledge space. Similarly, by providing access to computational tools, [4, 131, 133, 138, 173] demonstrated that LLMs can accomplish complex computational tasks. Consequently, leading LLM providers [1], have started to

integrate plugins to allow LLMs to invoke external tools through APIs.

Large Language Models Recent strides in the field of LLMs have renovated many downstream domains [29, 139, 167, 168], 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 [42, 149] and instruction fine-tuning [30, 56, 118, 148]. Recent open-sourced models like LLaMa [139], Alpaca [136], and Vicuna [28] 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 [66, 70, 96, 121]. Tools often incorporated include web-browsing [122], calculators [31, 138], translation systems [138], and Python interpreters [42]. While these efforts can be seen as preliminary explorations of marrying LLMs with tool usage, they generally focus on specific tools. Gorilla, 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.

Toolformer [121] and GPT-4 [1] highlight the importance of API calls, and have encouraged many works in employing API calls as tooling [79, 124]. Moreover, the application of API calls in robotics has been explored to some extent [2, 141]. 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, 35, 69, 78, 159]. Researchers have proposed an array of strategies to prompt LLMs to perform better in coding tasks, including in-context learning [23, 65, 149], task decomposition [64, 163], and self-debugging [25, 129]. Besides prompting, there have also been efforts to pretrain language models specifically for code generation [76, 97, 98].

DocPrompting [173] looked at choosing the right subset of code including API along with a retriever. Gorilla presents distinct advancements over DocPrompting. First, the way the data-sets are constructed are different, leading to intersting downstream artifacts. Gorilla focuses on model usages where we also collect detailed information about parameters, performance, efficiency, etc. This helps our trained model understand and respond to finer constraints for each API. Docprompting focuses on generic API calls but not on the details within an API call. Second, Gorilla introduces and uses the AST subtree-matching evaluation metric that helps measure hallucination which we find are more representative of code structure and API accuracy compared to traiditional NLP metrics. Finally, Gorilla focuses on instruction-tuning method and has "agency" to interact with users while DocPrompting

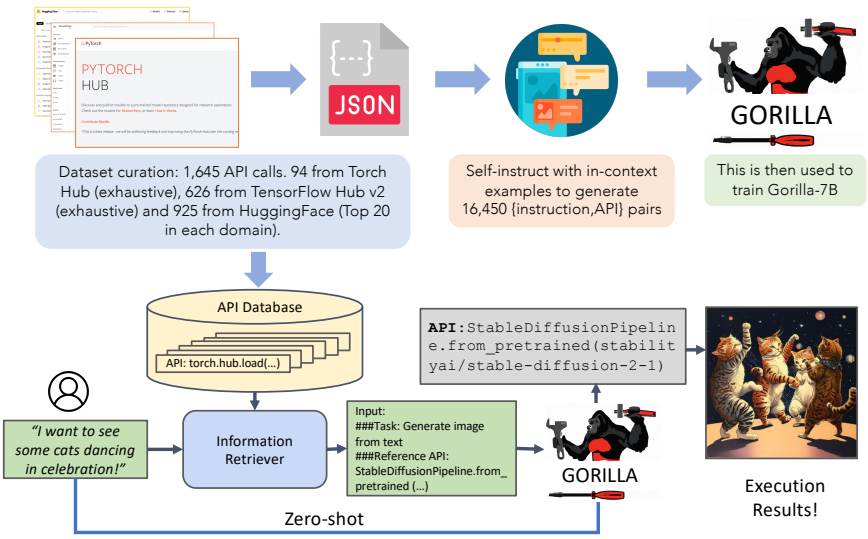

Figure 2.3: **Gorilla: A system for enabling LLMs to interact with APIs.** The upper half represents the training procedure as described in Sec 2.3. This is the most exhaustive API data-set for ML to the best of our knowledge. During inference (lower half), Gorilla supports two modes - with retrieval, 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.

focuses on building an NLP-to-Code generative model. On equal footing, we demonstrate that Gorilla performs better than DocPrompting in Appendix 2.5.

2.3 Methodology

We first 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 an information–retriever incorporated into the training and inference pipelines. Finally, we present our AST tree matching evaluation metric.

Dataset Collection

To collect the dataset, we recorded all online model cards for HuggingFace's "The Model Hub", PyTorch Hub, and TensorFlow Hub Models. Throughout the rest of this chapter, 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 arrive at 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. After filtering out model cards with little to no information, we are left with 626 models. Similar to TensorFlow Hub, we extract 95 models (exhaustive) 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, description}. We provide more information in 2.5. These fields were chosen to generalize beyond API calls within the ML domain, to other domains, including RESTful, SQL, and other potential API calls.

Instruction Generation Guided by the self-instruct paradigm [146], we employ GPT-4 to generate synthetic instruction data. We provide three in-context examples, along with reference API documentation, and task the model with generating real-world use cases that call upon the API. We specifically instruct the model to refrain from using any API names or hints when creating instructions. We constructed 6 examples (Instruction-API pairs) for each of the 3 model hubs. These 18 examples were the only hand-generated or modified data. For each of our 1,645 API datapoints, we generate 10 instruction-API pairs by sampling 3 of 6 corresponding instruction examples in each pair (illustrated in Figure 2.3).

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. 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. We also incorporate these instructions in our training dataset.

Gorilla

Our model Gorilla, is a retrieval-aware finetuned LLaMA-7B model, specifically for API calls. As shown in Figure 2.3, 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 datapoint 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. We would like to highlight that though we used the LLaMA model, our fine-tuning is robust to the pre-trained model (Appendinx 2.5).

Retriever-Aware training For training with retriever, the instruction-tuned dataset also appends to the user prompt, ‘‘`Use this API documentation for reference: <retrieved_API_doc_JSON>`’’. 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 (1) makes the LLM adapt to test-time changes in API documentation, (2) improves performance from in-context learning, and (3) 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 2.4.

Gorilla Inference During inference, the user provides the prompt in natural language (Figure 2.3). 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. Before being sent to Gorilla, the API documentation is concatenated to the user prompt along with the message *"Use this API documentation for reference."* 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 also implemented a system to execute these APIs, that is not a focus of this chapter.

Verifying APIs

Inductive program synthesis, where a program is synthesized to satisfy test cases, has found success in several avenues [12, 92]. 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 evaluation, 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.

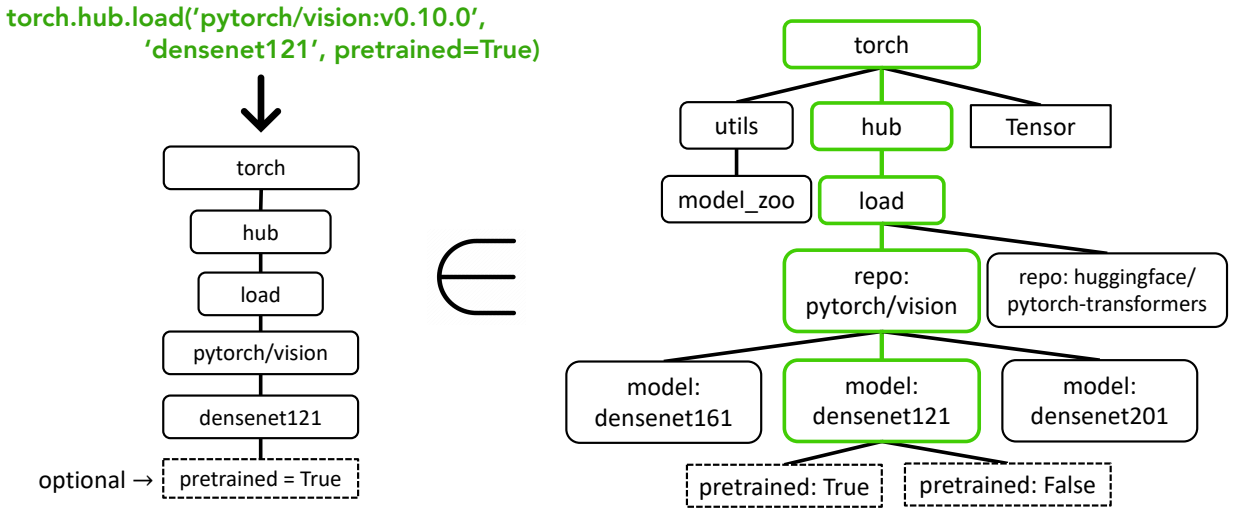

Figure 2.4: **AST Sub-Tree Matching to evaluate API calls.** On the left is an API call returned by Gorilla. We first build the associated API tree. We then compare this to our dataset, to see if the API dataset has a subtree match. In the above example, the matching subtree is highlighted in green, signifying that the API call is indeed correct. `Pretrained=True` is an optional argument.

Identifying and even defining hallucinations is 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. Figure 2.4 illustrates an example subtree check for 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`. We do not check for match along leaf node `pretrained=True` since that is an optional argument.

2.4 Evaluation

When evaluating Gorilla (finetuned on APIBench), we aim to answer the following questions:

1. How does Gorilla compare to other LLMs on API Bench? (2.4)

2. How well does Gorilla adapt to test-time changes in API documentation? (2.4)
3. How well can Gorilla handle questions with constraints? (2.4)

We show that Gorilla in the specific domain we evaluated on can outperform both open-source and close-source models. With our retriever-aware training techniques, our model can generalize to APIs that are outside of its training data. Finally, Gorilla also handles the constraints pretty well.

Baselines We primarily compare Gorilla with state-of-the-art language models in a zero-shot setting and with 3-shot in-context learning. The models under consideration include: GPT-4 by OpenAI with the `gpt-4-0314` checkpoint; GPT-3.5-turbo with the `gpt-3.5-turbo-0301` checkpoint, both of which are RLHF-tuned models specifically designed for conversation; Claude with the `claude-v1` checkpoint, a language model by Anthropic, renowned for its lengthy context capabilities; and LLaMA-7B, a state-of-the-art open-source LLM by Meta.

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 embedding model `text-embedding-ada-002-v2` from OpenAI where each embedding is 1,536 dimensional. 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 Section 2.3. 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.

AST Accuracy on API call

We first demonstrate the results for the AST accuracy for different models. We present the results in Table 2.1. 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, 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

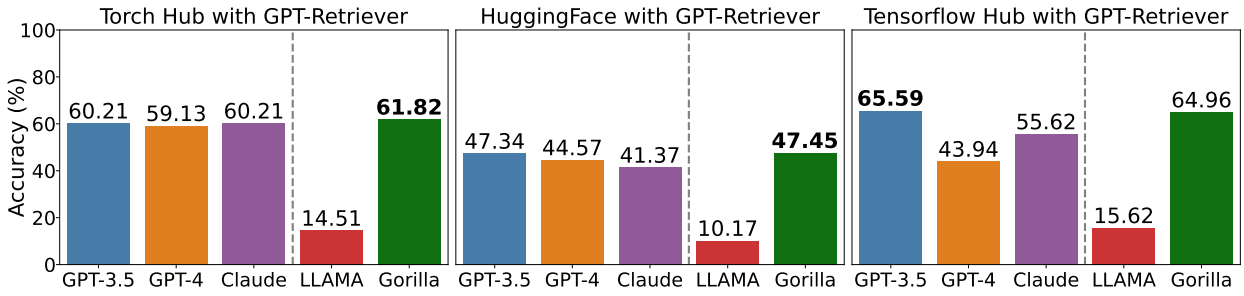

Figure 2.5: **Accuracy with GPT-retriever.** Methods to the left of the dotted line are closed source. Gorilla outperforms on Torch Hub and Hugging-Face while matching performance on Tensorflow Hub for all existing SoTA LLMs - closed source, and open source.

problem reduces to picking one of multiple choices. We also benchmark few-shot prompting for GPT in Tab. 2.7 and show that Gorilla still provides performance improvements.

Finetuning without Retrieval In Table 2.1 we show that lightly fine-tuned Gorilla is able to match, and often surpass performance in the 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 the retriever and putting a ground truth retriever during evaluation improves 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 sometimes misguide the model and result in more errors.

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 Table 2.3, 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 good retriever, finetuning an LLM to incorporate retrieved results is preferred than zero-shot finetuning.

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,

Table 2.1: **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

Table 2.2: **Gorilla 0-shot with GPT 3-shot in-context examples.**

	HF (Acc ↑)	**TH (Acc ↑)**	**TF (Acc ↑)**
GPT-3.5 (0s)	16.81	41.93	41.75
GPT-4 (0s)	19.80	54.30	18.20
GPT-3.5 (3i)	25.77	73.11	71.82
GPT-4 (3i)	26.32	**75.80**	77.37
Gorilla (0s)	**58.05**	**75.80**	**83.79**

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 2.5.

Table 2.3: **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

AST as a Hallucination Metric We manually executed Gorilla's API generations to evaluate how well AST works as an evaluation metric. Executing every code generated is impractical within academic setting—for example, executing the HuggingFace model needs the required library dependencies (e.g., transformers, sentencepiece, accelerate), correct coupling of software kernels (e.g., torch vision, torch, cuda, cudnn versions), and required hardware support (e.g., A100 40G gpus). Hence, to make it tractable, we sampled 100 random Gorilla generations from our evalualtion set. The accuracy using AST subtree matching is 78%, consistent with human evaluation of 78% accuracy in calling the right API. All the generations that AST flagged as incorrect, were the same ones that were manually flagged as incorrect. Additionally, Gorilla also generates supporting code to call the API which includes installing dependencies e.g., `pip install transformers[sentencepiece])`, environment variables, etc. When we manually attempted to execute the code, 72% of all codes generated were executed successfully. It's worth noting that the 6% discrepancy are not semantic errors, but errors that arose due to factors external to the API, and in the supporting code. We have included the full example to illustrate this further in 2.5. Considering the significant time and effort required for manual validation of each generation, the strong correlation between human evaluation and the AST evaluation further reinforces our belief in the efficiency of using AST as a robust offline metric.

	Accuracy
Gorilla AST metric (proposed)	0.78
Eval by Human	0.78
Code Executable (Eval by Human)	0.72

Table 2.4: **AST evaluation metric has strong correlation with accuracy.**

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

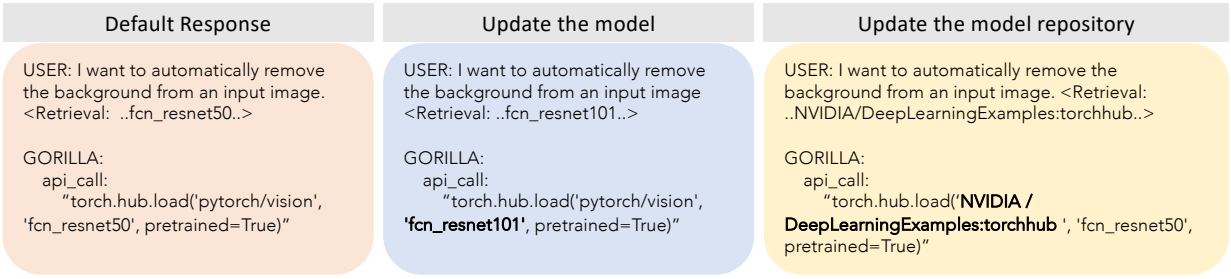

Figure 2.6: **Gorilla's retriever–aware training enables it to react to changes in the APIs.** The second column demonstrates changes in model upgrading FCN's ResNet–50 backbone to ResNet–101. The third column demonstrate changes in model registry from `pytorch/vision` to `NVIDIA/DeepLearningExamples:torchhub`.

outpaces the re-training 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 underlying API documentation undergoes modifications. This 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.

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

Table 2.5: **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**				

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%.

In Table 2.5, 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 2.1). 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, highlighting Gorilla's ability to navigate APIs while considering the trade-offs between different constraints.

2.5 Insights and In-depth

In this section, we present details and insights on the different training aspects of Gorilla LLM along with additional results.

Dataset Details

Gorilla's dataset is multi-faceted, comprising three distinct domains: Torch Hub, Tensor Hub, and HuggingFace. Each entry within this dataset is rich in detail, carrying critical pieces of information that further illuminate the nature of the data. Delving deeper into the specifics

of each domain, Torch Hub provides 95 APIs. The second domain, Tensor Hub, is more expansive with a total of 696 APIs. Finally, the most extensive of them all, HuggingFace, comprises 925 APIs. To enhance the value and utility of our dataset, we've undertaken an additional initiative. With each API, we have generated a set of 10 unique instructions. These instructions, carefully crafted and meticulously tailored, serve as a guide for both training and evaluation. This initiative ensures that every API is not just represented in our dataset, but is also comprehensively understood and effectively utilizable.

In essence, our dataset is more than just a collection of APIs across three domains. It is a comprehensive resource, carefully structured and enriched with added layers of guidance and evaluation parameters.

Domain Classification The unique domain names encompassed within our dataset are illustrated in Figure 2.7. The dataset consists of three sources with a diverse range of domains: Torch Hub houses 6 domains, Tensor Hub accommodates a much broader selection with 57 domains, while HuggingFace incorporates 37 domains. To exemplify the structure and nature of our dataset, we invite you to refer to the domain names represented in Figure 2.8.

API Call Task In this task, we test the model's capability to generate a single line of code, either in a zero-shot fashion or by leveraging an API reference. Primarily designed for evaluation purposes, this task effectively gauges the model's proficiency in identifying and utilizing the appropriate API call.

API Provider Component This facet relates to the provision of the programming language. In this context, the API provider plays a vital role as it serves as a foundation upon which APIs are built and executed.

Explanation Element This component offers insights into the rationale behind the usage of a particular API, detailing how it aligns with the prescribed requirements. Furthermore, when certain constraints are imposed, this segment also incorporates those limitations. Thus, the explanation element serves a dual purpose, offering a deep understanding of API selection, as well as the constraints that might influence such a selection. This balanced approach ensures a comprehensive understanding of the API usage within the given context.

Code Example code for accomplishing the task. We de-prioritize this as we haven't tested the execution result of the code. We leave this for future works, but make this data available in-case others want to build on it.

Gorilla Details

We provide all the training details for Gorilla in this section. This includes how we divide up the training, evaluation dataset, training hyperparameters for Gorilla.

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

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)

Data For HuggingFace, we devise the entire dataset into 90% training and 10% evaluation. For Torch Hub and Tensor Hub, we devise the data in to 80% training and 20% testing.

Training We train Gorillafor 5 epochs with the 2e-5 learning rate with cosine decay. The details are provide in Tab. 2.6. We finetune it on 8xA100 with 40G memory each.

Table 2.6: **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

Performance Comparison

We provide a full comparison of each model's performance in this section. In Fig 2.10 and Fig. 2.11, 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.

Evaluation

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

$$\text{API\_name}(\text{API\_arg}_1, \text{API\_arg}_2, ..., \text{API\_arg}_k)$$

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.

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

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. 2.9 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. 2.1.

AST as a Hallucination Metric

We evaluated the generated results on 100 LLM generations (randomly chosen from our eval set). The accuracy using AST subtree matching is 78%, consistent with human evaluation with 78% accuracy in calling the right API. All the generations that AST flagged as incorrect, were the same ones that were manually also flagged as incorrect. Additionally, Gorilla generates supporting code to call the API which includes installing dependencies (e.g., `pip install transformers[sentencepiece]`), environment variables, etc. When we manually attempted to execute end-to-end code, 72% of all codes generated were executed successfully. It's worth noting that the 6% discrepancy were NOT semantic errors, but errors that arose due to factors external to the API in the supporting code - we have included an example to illustrate this further. Considering the significant time and effort required for manual validation of each generation, our data further reinforces our belief in the efficiency of using AST as a robust offline metric.

Here is a representative example, where we are able to load the correct model API. However, in the supporting code, after we have the output from the API, the `zip()` function tries to combine sentiments and scores together. However, since scores is a float, it's not iterable. `zip()` expects both its arguments to be iterable, resulting in an `'float' object`

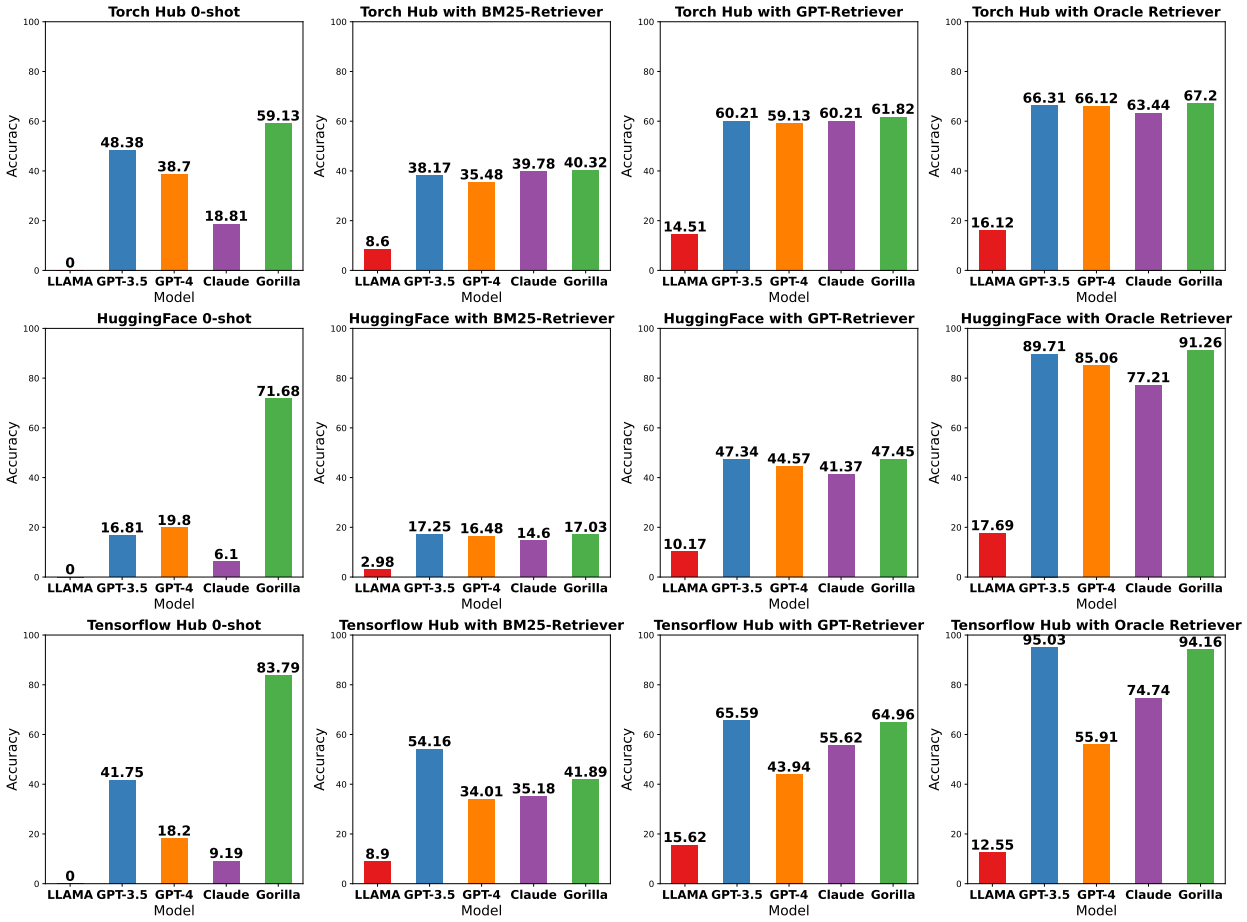

Figure 2.10: **Performance**: We plot each model's performance on different configurations. We see that Gorilla performs extremely well in the zero-shot setting. While even when the oracle answer is given, Gorilla is still the best.

`is not iterable` error.

Gorilla 0-shot (vs) GPT 3-shot

In Table 2.7, we compare the performance of 3-shot in-context learning. We find that 3-shot it-context learning boosts the performance of GPT-4, with even matching accuracy for Torch Hub, but Gorilla still performs better on average.

Gorilla (VS) DocPrompting

We evaluate Gorilla and DocPrompting [173] on the HuggingFace Dataset from Table 2.1. For a 7B model, when trained on the same number of epochs, with and the same learning

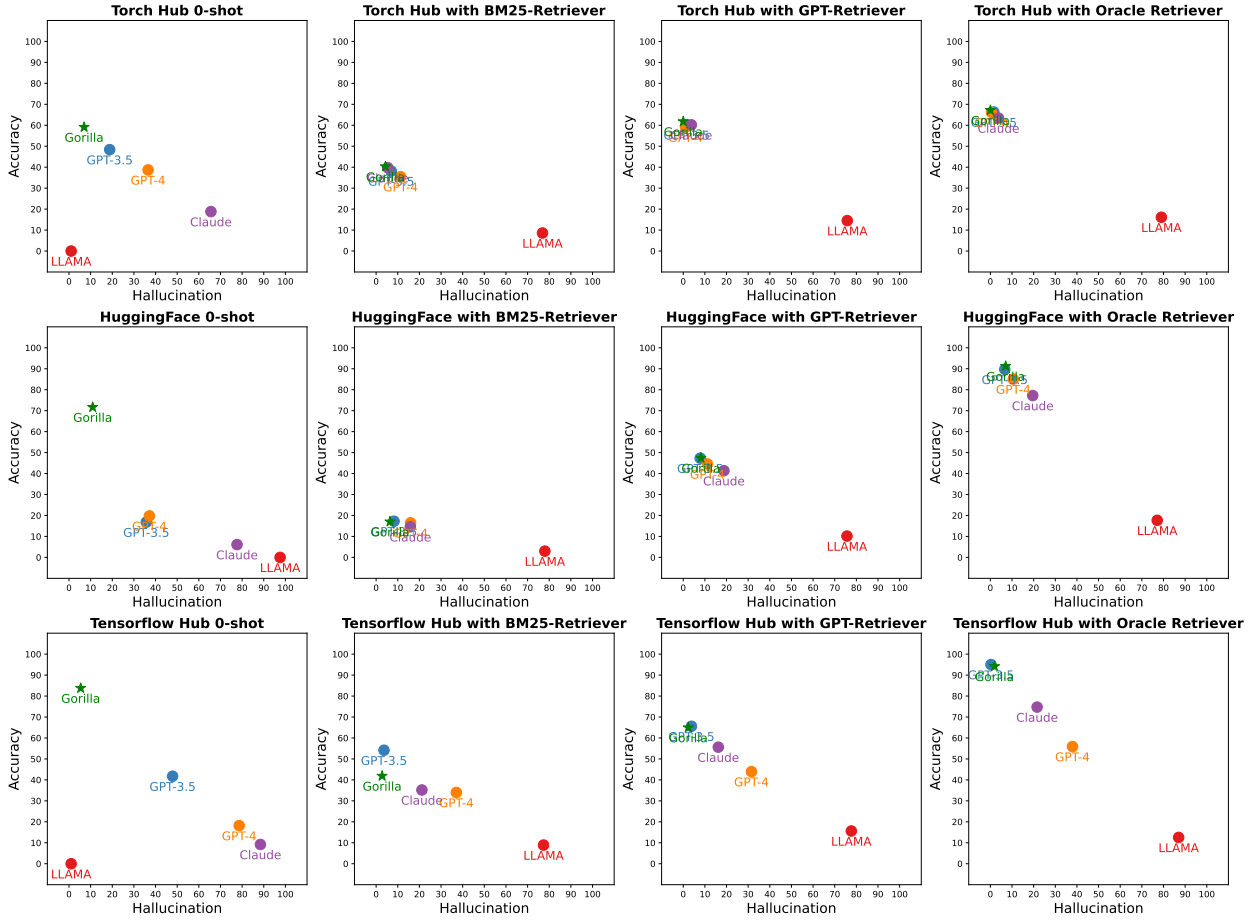

Figure 2.11: **Accuracy vs Hallucination**: We plot each model's performance on different configurations. We found that in the zero-shot setting, Gorilla has the most accuracy gain while maintaining good factual capability. When prompting with different retrievers, Gorilla is still capable to avoid the hallucination errors.

Table 2.7: **Evaluating Gorilla 0-shot with GPT 3-shot incontext examples.** Table Legend: *3-incont* → 3-incontext examples, *HF* → HuggingFace, *TH* → TorchHub, *TF* → TensorFlow Hub. *Acc* → Accuracy (higher is better). *Hall* → Hallucination (lower is better).

	HF (Acc ↑)	HF (Hall ↓)	TH (Acc ↑)	TH (Hall ↓)	TF (Acc ↑)	TF (Hall ↓)
GPT-3.5 (0-shot)	16.81	35.73	41.93	10.75	41.75	47.88
GPT-4 (0-shot)	19.80	37.16	54.30	34.40	18.20	78.65
GPT-3.5 (3 incont)	25.77	32.30	73.11	72.58	71.82	11.09
GPT-4 (3 incont)	26.32	35.84	**75.80**	**13.44**	77.37	11.97
Gorilla (0-shot)	**58.05**	**28.32**	**75.80**	16.12	**83.79**	**5.40**

```
from transformers import pipeline

def load_model():
  classifier = pipeline('sentiment-analysis',
    model='nlptown/bert-base-multilingual-uncased-sentiment')
  return classifier

def process_data(comments, classifier):
  response = classifier(comments)
  sentiments = response[0]['label'].split()
  scores = response[0]['score']
  result = [{'sentiment': sentiment, 'score': score}
    for sentiment, score in zip(sentiments, scores)]
  return result

comments = "These␣comments␣are␣about␣our␣news␣website."
# Load the model
classifier = load_model()
# Process the data
response = process_data(comments, classifier)
print(response)
```

Figure 2.12: **How well can we execute Gorilla generated code?** In this example, the API call by Gorilla model is accurate and bug-free, but the supporting `zip()` code has a bug.

Table 2.8: **Evaluating Gorilla (vs) DocPrompting.** Gorilla improves accuracy, while lowering the hallucination.

Accuracy ↑		Hallucination ↓	
DocPrompting	Gorilla	DocPrompting	Gorilla
61.72	**71.68**	17.36	**10.95**

rate for both the models, Gorilla improves accuracy while reducing hallucination.

Sensitivity to pre-training

Gorilla's training recipe is robust to the pre-training strategies and recipes of the underlying model. From figure 2.13 we demonstrate that all the three models can converge to within a

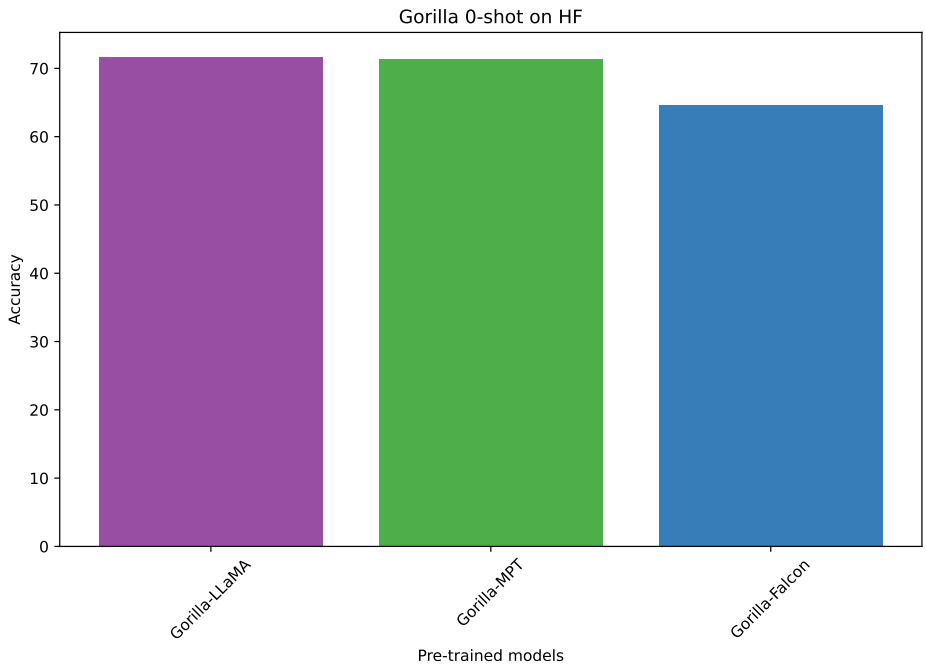

Figure 2.13: **Evaluating robustness.** For the same train-eval dataset, our fine-tuning recipe is robust to the underlying base model.

few percentage points in accuracy independent of the pre-trained base model.

2.6 OpenFunctions

OpenFunctions [58] is designed to extend LLMs to formulate executable APIs call given natural language instructions and API context. Unlike Gorilla which takes in a standard documentation of the API to return the right arguments, in open-functions, the LLM is provided with k different functions. To mean, k different function descriptions in JSON, or XML schema []. OpenFunctions LLM returns the right function, along with the required arguments filled in. While the fine-tuning of Gorilla means every user needs to be OpenFunctions is an LLM that is train using a curated set of API documentation, and Question-Answer pairs generated from the API documentations.

Training OpenFunctions

OpenFunctions is a series of models, in it's third iteration at the time this dissertation was written. Here, we will discuss the third and most latest iteration, OpenFunctions-v2.

OpenFunctions-v2 is a 6.91B parameter model trained further upon on the Deepseek-Coder-7B-Instruct-v1.5 6.91B model. To train the model, we collected in total of 65,283

```
query + [{'name': 'func1', 'description': 'order␣takeout'}] -> ans1
% Transformed to =>
query + [{'name': 'func2', 'description': 'order␣takeout'}] -> [ans2]
```

Figure 2.14: **Function Name Transformation:** From the original question-function-answer pairs, we augment this with a different function names to avoid the model memorizing the correlation between function names and the question (e.g., 'uber' API is used for transportation).

```
query + [{'name': 'func1', 'description': 'order␣takeout'}] -> ans1
                        % Transformed to =>
query + [{'name': 'func1', 'description': 'order␣takeout'},
         {'name': 'func2', 'description': 'get␣weather'}] -> [ans1]
```

Figure 2.15: **Multiple Functions Transformation:** Transform the original function with multiple function calls included in the training, so that the model can learn to choose which function call to use.

question-function-answer pairs from three different sources: Python packages (19,353), Java repositories (16,586), Javascript Repositories (4,245), public-API (6,009), and Command Line Tools (19,090) from various cloud providers.

After the initial data collection, we carry out four types of data augmentations to diversify our training dataset. First, we change the function names. This is critical to ensure the model does not "memorize" the API mapping. Second, we add random (randomly chosen, and random number of) functions to make our data-set compatible with parallel functions. This way we can generate multiple-function datasets from simple functions. Third, we adopt similar strategies of perturbing the prompt to generate scenarios of parallel-functions. We then extend it to also include multiple- and parallel- functions in the same data-points. Finally, we mix some portion of the dataset in which the functions provided during the input is not sufficient to the task. We flag these as "Relevance Detection" scenarios. As with most LLM training, we varied the extents of each data augmentation to train a robust model.

Following the data augmentation process, we further refine the dataset by employing the Rouge score [80] for de-duplication, effectively eliminating redundant entries. Gorilla-OpenFunctions-v2 trained with this recipe set the new benchmark [161] as a state-of-the-art in function calling matching larger proprietary models in performance all the while being substantially low cost.

```
query1 + [{'name': 'func1', 'description': 'order␣takeout'}] -> ans1
                        Transformed to =>
query2 + [{'name': 'func1', 'description': 'order␣takeout'}] -> [ans1,
    ans2]
```

Figure 2.16: **Parallel Functions Transformation:** To handle a more complex case where multiple functions will be selected to answer the user's request, we change the original question to ask for multiple outputs.

```
query1 + [{'name': 'func1', 'description': 'order␣takeout'}] -> ans1
                        % Transformed to =>
query2 + [{'name': 'func1', 'description': 'order␣takeout'},
    {'name': 'func2', 'description': 'get␣weather'}] -> [ans1, ans2]
```

Figure 2.17: **Parallel Multiple Functions Transformation:** The combined of the above parallel, and multiple transforms.

```
query1 + [{'name': 'func1', 'description': 'order␣takeout'}] -> ans1
                        % Transformed to =>
query2 + [{'name': 'func1', 'description': 'order␣takeout'}] ->
    [Error, the function cannot solve the question.]
```

Figure 2.18: **Function Relevance Detection:** We also include some portion of the dataset in which the functions provided cannot solve the task. We call this 'Relevance Detection'.

2.7 Conclusion

In our study, we spotlight techniques designed to enhance the LLM's ability to accurately identify the appropriate API. Since APIs as a universal language enabling diverse systems to communicate effectively, their correct usage can boost the ability of LLMs to interact with tools. In this chapter, we proposed Gorilla, a novel pipeline for finetuning LLMs to call APIs. The finetuned model's performance surpasses prompting the state-of-the-art LLM 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.

Chapter 3

RAFT

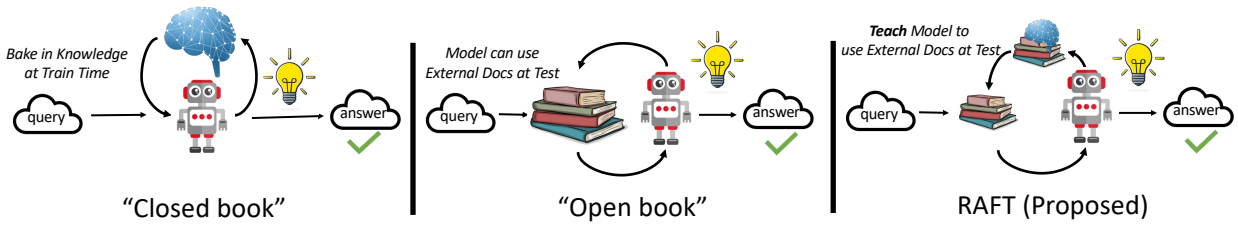

Figure 3.1: **How best to prepare for an exam?** (a) Fine-tuning based approaches implement "studying" by either directly "memorizing" the input documents or answering practice QA without referencing the documents. (b) Alternatively, in-context retrieval methods fail to leverage the learning opportunity afforded by the fixed domain and are equivalent to taking an open-book exam without studying. While these approaches leverage in-domain learning, they fail to prepare for open-book tests. In contrast, our approach (c) RAFT leverages fine-tuning with QA pairs while referencing the documents in a simulated imperfect retrieval setting — thereby effectively preparing for the open-book exam setting.

3.1 Introduction

Trained on vast quantities of public data, LLMs have achieved significant advances in a wide range of general knowledge reasoning tasks [15, 149].

However, LLMs are increasingly being employed in specialized domains to support tasks ranging from code completion for specific software frameworks to question answering on specific document collections (e.g., legal or medical documents). In these settings, general knowledge reasoning is less critical but instead, the primary goal is to maximize accuracy based on a given set of documents. Indeed, adapting LLMs to the specialized domains (e.g., recent news, enterprise private documents, or program resources constructed after the training cutoff) is essential to many emerging applications [70, 142] and is the focus of this work.

This chapter studies the following question: *How can we adapt pre-trained LLMs for Retrieval Augmented Generation (RAG) in specialized domains?*

When it comes to adapting LLMs to specialized domains, we consider the following two candidates: in-context learning through retrieval-augmented generation (RAG) and supervised fine-tuning. RAG-based methods allow the LLM to reference the documents when answering questions. However, these methods fail to leverage the learning opportunity afforded by the fixed domain setting and early access to the test documents. Alternatively, supervised fine-tuning offers the opportunity to learn more general patterns in the documents and better align to end tasks and user preferences [172]. However, existing fine-tuning based approaches either fail to leverage the documents at test time (don't incorporate RAG) or fail to account for the imperfections in the retrieval process during training.

We can draw an analogy to an open-book exam. Existing in-context retrieval methods are equivalent to taking an open-book exam without studying. Alternatively, existing fine-tuning based approaches implement "studying" by either directly "memorizing" [158] the input documents or answering practice questions [146] without referencing the documents. While these approaches leverage in-domain learning, they fail to prepare for the open-book nature of the test setting.

In this chapter, we study how to combine supervised fine-tuning (SFT) with RAG. We propose a novel adaptation strategy – Retrieval-Augmented Fine Tuning (RAFT). RAFT specifically addresses the challenge of fine-tuning LLMs to incorporate domain knowledge while also improving in-domain RAG performance. RAFT aims to not only enable models to learn domain specific knowledge through fine-tuning, but also to ensure robustness against inaccurate retrievals. This is achieved by training the models to understand the dynamics between the question posed (prompt), the domain specific documents retrieved, and the appropriate answer. Going back to our analogy, our approach is analogous to studying for an open-book exam by recognizing relevant, and irrelevant retrieved documents.

In RAFT, we train the model to answer the question (Q) from Document(s) (D*) to generate an answer (A*), where A* includes chain-of-thought [7, 149], and in the presence of distractor documents (D_k). We explain the methodology in detail in Section 3.3 and analyze the sensitivity to the number of distractor documents (k) at train- and test- time in Section 3.5. RAFT consistently outperforms Supervised-finetuning both with- and without- RAG across PubMed [34], HotpotQA [162], and HuggingFace Hub, Torch Hub, and Tensorflow Hub Gorilla datasets [107], presenting a novel, yet simple technique to improve pre-trained LLMs for in-domain RAG.

3.2 LLMs for Open-Book Exam

To understand our goal better, we expand on our analogy between training an LLM in the real-world setting of preparing for an exam.

Closed-Book Exam A closed book exam often refers to a scenario where the LLMs do not have access to any additional documents or references to answer the questions during the exam. For LLMs, this is equivalent to the scenario, for example, in which the LLM is used as a chatbot. In this scenario, the LLM draws from the knowledge baked in during pre-training and supervised finetuning to respond to the prompt.

Open Book Exam In contrast, we liken the open-book exam setting to the scenario in which the LLM can refer to external sources of information (e.g., a website or a book chapter). In such scenarios, typically, the LLM is paired with a retriever which retrieves 'k' documents (or specific segments of the document) which are appended to the prompt. It is only through these documents retrieved that the LLM gains access to "new knowledge". As a result, we argue that the LLM's performance in these settings, where it is trained as a general-purpose LLM is largely dependent on the quality of the retriever and how accurately the retriever can identify the most relevant piece of information.

Domain Specific Open-Book Exam In this chapter, we focused on a narrower but increasingly popular domain than the general open book exam, called the domain specific open book exam. In domain specific open book exams, we know apriori the domain in which the LLM will be tested – used for inference. The LLM can respond to the prompt using use any and all information from this specific domain, which it has been fine-tuned on. Examples of domain specific examples include enterprise documents, latest news, code repositories belonging to an organization, etc. In all these scenarios, the LLM will be used to respond to the questions, whose answers can be found within a collection of documents (a small practical domain). The retrieval technique itself has little to no impact on the mechanism (though it may impact the accuracy). This chapter mainly studies this, domain specific open-book setting and how to adapt a pretrained LLM to this specific domain, including how to make it more robust to a varying number of retrieved documents and distractors.

3.3 RAFT

In this section, we present RAFT, a novel way of training LLMs for domain specific open-book exams. We first introduce the classical technique of supervised fine-tuning, followed by the key takeaways from our experiments. Then, we introduce RAFT , a modified version of general instruction tuning. Lastly, we provide an overview of the experiments to expect in the later sections.

Supervised Finetuning

Consider the supervised fine-tuning (SFT) setting for a Question-Answer dataset. The formulation consists of the Dataset (D) from which a set of Question (Q) and corresponding answer (A) pairs are derived or already available. In the classical SFT setting, the model is

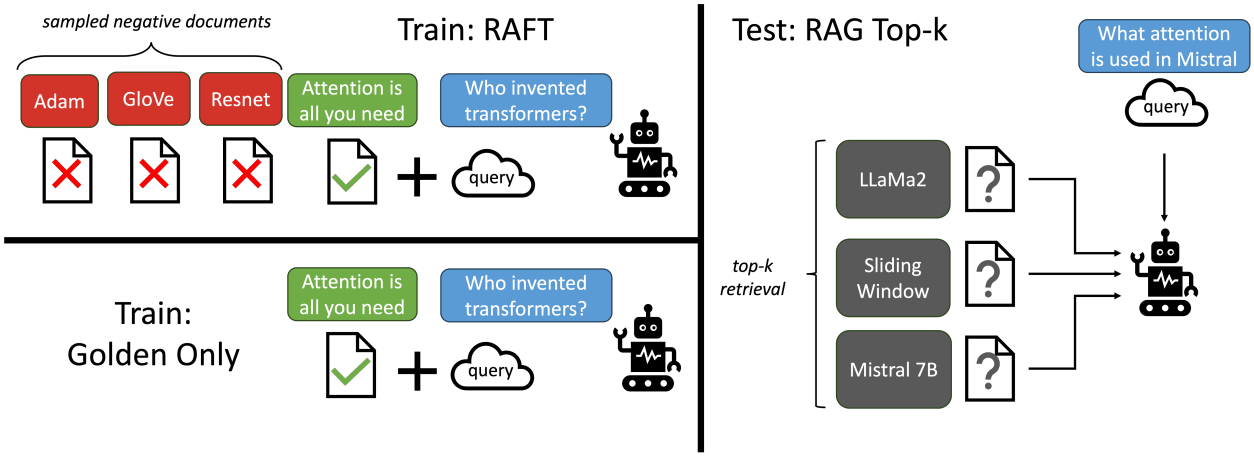

Figure 3.2: **Overview of our RAFT method.** The top-left figure depicts our approach of adapting LLMs to *reading* solution from a set of positive and negative documents in contrast to standard RAG setup where models are trained based on the retriever outputs, which is a mixture of both memorization and reading. At test time, all methods follow the standard RAG setting, provided with a top-k retrieved documents in the context.

trained to improve its ability to answer the questions based on its knowledge - obtained either during pre-training, or during the SFT training phase. The model so trained can also be used at test-time with the Retrieval Augmented Generation (RAG) setting, where additional documents can be introduced in the prompt to help the model answer the question. This can be represented as follows:

- Train: $\mathbf{Q} \rightarrow \mathbf{A}$
- 0-shot Inference: $\mathbf{Q} \rightarrow \mathbf{A}$
- RAG Inference: $\mathbf{Q} + \mathbf{D} \rightarrow \mathbf{A}$

RAFT

Retrieval Aware Fine-Tuning (RAFT), presents a novel recipe to prepare fine-tuning data to tailor the models for domain specific open-book settings, equivalent to in-domain RAG In RAFT, we prepare the training data such that each data point contains a question (Q), a set of documents (D_k), and a corresponding Chain-of-though style answer (A^*) generated from one of the document (D^*). We differentiate between two types of documents: 'oracle' documents ($D*$) i.e. the documents from which the answer to the question can be deduced, and 'distractor' documents (D_i) that do not contain answer-relevant information. As an implementation detail, the 'oracle' document doesn't need to be a single document, but can be more than one document, as is the case in HotpotQA [162]. Then, for P fraction of

the questions (q_i) in the dataset, we retain the oracle document (d_i^*) along with distractor documents (d_{k-1}). For $(1 - P)$ fraction of the questions (q_i) in the dataset, we include no oracle document and only include distractor documents (d_k). We then fine-tune the language model using the standard supervised training (SFT) technique, training it to generate answers from the provided documents and questions. Fig. 3.2 illustrates the high-level design principal for RAFT .

We demonstrate that our approach trains the model to perform better RAG on the set of documents it is trained on *i.e., in-domain*. By removing the oracle documents in some instances, we are compelling the model to memorize answers instead of deriving them from the context. The training data for RAFT is as follows, and an example of training data can be seen in Fig. 3.3:

- **P** % of data: $\mathbf{Q} + \mathbf{D}^* + \mathbf{D}_2 + \ldots + \mathbf{D}_k \rightarrow \mathbf{A}^*$
- $(1 - \mathbf{P})$ % of data: $\mathbf{Q} + \mathbf{D}_1 + \mathbf{D}_2 + \ldots + \mathbf{D}_k \rightarrow \mathbf{A}^*$

Subsequently, for the test scenario, the model is provided with the Q and top-k documents retrieved by the RAG pipeline. Note that RAFT is independent of the retriever used.

A key factor in enhancing training quality is the generation of a reasoning process, such as Chain-of-Thought, to explain the provided answers.RAFT approach is similar: we demonstrate that creating a full reasoning chain and in addition, clearly citing sources enhances the model's accuracy in answering questions. In Fig. 3.3, we illustrate this set-up. Generating the training data in this fashion, involves presenting the model with a question, context, and verified answers, and then requesting it to form a reasoning chain that appropriately references the original context.

For all the datasets in our experiments, we generate the answers using the technique described above. Note that the Gorilla APIBench dataset, already includes reasoning in the answers. We provide an example of the generation step in Fig. 3.3, the detailed reasoning answer includes a citation from the original context inside `##begin_quote##` and `##end_quote##` as well as the detailed explanation on how to reach the conclusion based on the citations. We demonstrate that adding detailed reasoning paragraphs helps boost the model's performance in our experiment section.

3.4 Evaluation

We design our experiments to study how well RAFT performs compared to various baselines. We find that the RAFT-7B model (a finetuned version of LlaMA-2) is better at reading and extracting information from in-domain documents, than domain specific finetuned model, and general-purpose model with RAG. As an ablation, we also demonstrate how important it is for the model to learn with Chain-of-Thought responses. In this section, we will first introduce all the datasets we used in the experiments, then all the baseline model/fine-tuning techniques that we benchmark against.

```
Question: The Oberoi family is part of a hotel company that has a head office in what
city?

context: [The Oberoi family is an Indian family that is famous for its involvement in
hotels, namely through The Oberoi Group]...[It is located in the city center of
Jakarta, near Mega Kuningan, adjacent to the sister JW Marriott Hotel. It is operated
by The Ritz-Carlton Hotel Company. The complex has two towers that comprise a hotel
and the Airlangga Apartment respectively]...[The Oberoi Group is a hotel company with
its head office in Delhi.]

Instruction: Given the question, context, and answer above, provide a logical
reasoning for that answer. Please use the format of: ##Reason: {reason} ##Answer:
{answer}.

--------------------------------------------------------------------------------

CoT Answer: ##Reason: The document ##begin_quote## The Oberoi family is an Indian
family that is famous for its involvement in hotels, namely through The Oberoi Group.
##end_quote## establishes that the Oberoi family is involved in the Oberoi group, and
the document ##begin_quote## The Oberoi Group is a hotel company with its head office
in Delhi. ##end_quote## establishes the head office of The Oberoi Group. Therefore,
the Oberoi family is part of a hotel company whose head office is in Delhi. ##Answer:
Delhi
```

Figure 3.3: **RAFT prompt** to help LLM evaluate its own generated reasoning and answers, contrasting them with the correct reasoning and answers. The LLM is prompted to identify errors in its reasoning and extract key insights for improvement. This figure specifically represents the ‘GenerateExplanation‘ step in the RAFT algorithm (3.3).

Datasets

In our experiments, we use the following datasets to evaluate our model and all baselines. We selected these datasets to represent both popular and diverse domains including Wikipedia, Coding/API documents, and question-answering on medical documents.

- Natural Questions (NQ) [68], Trivia QA [60] and HotpotQA [162] are the open-domain question-answers based on Wikipedia, mainly focused on common knowledge (e.g., movies, sports, etc).
- HuggingFace, Torch Hub, and TensorFlow Hub are from the APIBench [107] proposed in the Gorilla chapter. These benchmarks measure how to generate the correct, functional, and executable API calls based on the documentation.
- PubMed QA [59] is a question-answering dataset tailored only for biomedical-research question-answering. It mainly focuses on answering medical and biology questions

Table 3.1: **RAFT improves RAG performance forall specialized domains:** Across PubMed, HotpotQA, HuggingFace, Torch Hub, and Tensorflow Hub, we see that domain specific Finetuning improves significantly of the performance of the base model, but RAFT consistently outperforms the existing domain specific finetuning method with or without RAG. This suggests the need to train the model with context. We compare our model with LLaMA finetuning receipes, and provide GPT-3.5 for reference.

	PubMed	HotpotQA	HuggingFace	Torch Hub	TensorFlow Hub
GPT-3.5 + RAG	71.60	**41.5**	29.08	60.21	65.59
LLaMA2-7B	56.5	0.54	0.22	0	0
LLaMA2-7B + RAG	58.8	0.03	26.43	08.60	43.06
DSF	59.7	6.38	61.06	84.94	86.56
DSF + RAG	71.6	4.41	42.59	82.80	60.29
RAFT (LLaMA2-7B)	**73.30**	35.28	**74.00**	**84.95**	**86.86**

based on a given set of documents.

Note that the first category of dataset (NQ, Trivia QA, and HotpotQA) is a relatively general domain whereas the latter two domains are on very domain specific documents.

Baselines We consider the following baselines for our experiments:

- LlaMA2-7B-chat model with 0-shot prompting: this is the commonly used instruction-finetuned model for QA tasks, where we provide clearly written instructions, but no reference documentation.
- LlaMA2-7B-chat model with RAG (Llama2 + RAG): similar to the previous setting, except here we include reference documents. This is a popular technique when dealing with domain specific QA tasks.
- Domain specific Finetuning with 0-shot prompting (DSF): Performing standard supervised finetuning, without documents in context. We find that it mostly useful to align the answering style of the model as well as get familiar with the domain context.
- Domain specific Finetuning with RAG (DSF + RAG): Equip a domain specific finetuned model with external knowledge using RAG. So, for the "knowledge" the model does not know, it can still refer to the context.

Results

Using the above datasets and baselines, we evaluate our model RAFT and demonstrate the effectiveness of RAFT in Tab. 3.1. We see that RAFT consistently and significantly outperforms the baselines. Compared with the base Llama-2 instruction-tuned model, RAFT with RAG does much better in terms of extracting information as well as being robust towards distractors. The gain can be as big as 35.25% on Hotpot QA and 76.35% on Torch Hub evaluation. Compared with DSF on the specific dataset, our model does better at relying on the provided context to solve the problem. RAFT does much better on tasks like HotpotQA and HuggingFace datasets (30.87% on HotpotQA and 31.41% on HuggingFace). Note that for PubMed QA, since it is a binary yes/no question, we don't observe significant gains when we compare our model with DSF + RAG. Even compared with a much larger and better model GPT-3.5, RAFT demonstrates significant advantages.

Overall, the LLaMA-7B model, both with and without the RAG, performs poorly due to its answering style not aligning with the ground truth. By applying domain specific tuning, we significantly enhance its performance. This process enables the model to learn and adopt the appropriate style of answering. However, introducing RAG to a domain-specifically fine-tuned (DSF) model doesn't invariably lead to better outcomes. This might indicate that the model lacks training in context processing and extracting useful information from it. By incorporating our method, RAFT , we train the model not only to match its answering style with that required but also to improve its document processing capabilities. Consequently, our approach outperforms all others.

Effect of CoT

We also conduct an analysis to evaluate the effectiveness of the Chain-of-Thought approach in enhancing the model's performance. As indicated in Table 3.2, simply providing the answer to a question may not always be adequate. This approach can lead to a rapid decrease in loss, resulting in the training process to diverge. Incorporating a reasoning chain that not only guides the model to the answer but also enriches the model's understanding can improve the overall accuracy. In our experiments, integrating the Chain-of-Thought significantly enhances training robustness. We employ GPT-4-1106 to generate our Chain-of-Thought prompts and include an example of the prompt we used in Figure 3.3.

Qualitative Analysis

To illustrate the potential advantages of RAFT over the domain-specifically fine-tuned (DSF) approach, we present a comparative example in Figure 3.4. This example qualitatively demonstrates a scenario where the DSF model becomes confused by a question asking for the identity of a screenwriter. Instead of providing the correct name, it mistakenly cites one of the films written by the screenwriter. In contrast, the RAFT model accurately answers the question. This discrepancy suggests that training a model solely with question-answer pairs

Table 3.2: **Ablation on Chain-of-Thought:** The numbers of RAFT and RAFT without CoT. Results on various datasets show that adding CoT can significantly improve the performance of the finetuned model. With a gain of 9.66% and 14.93% on the Hotpot QA and HuggingFace datasets respectively.

	PubMed	HotpotQA	HuggingFace	Torch Hub	TensorFlow Hub
RAFT w.o CoT	68.30	25.62	59.07	**86.56**	83.21
RAFT	**73.30**	**35.28**	**74.00**	84.95	**86.86**

may impair its ability to derive relevant context from provided documents. The comparison underscores the importance of incorporating both standard instructional tuning and context comprehension into the training dataset to preserve and enhance the model's ability to process text effectively.

Should we train the LLM always with the oracle context for RAG?

In our exploration of whether large language models (LLMs) should always be trained with the oracle context for Retrieval-Augmented Generation (RAG), we address a key question: what proportion ($P\%$) of the training data should include oracle documents? Intuitively, one might assume that for effective training in reading and extracting information from context (e.g., RAG tasks), the oracle document should always be included during training (P = 100%). However, our findings challenge this assumption: incorporating a portion of the training data without the oracle document in the context (P = 80%) appears to enhance the model's performance on RAG tasks.

Fig. 3.5 investigates the hyperparameter P%, which represents the percentage of training instances that should include oracle documents. Our analysis reveals that the optimal proportion varies across datasets, with figures ranging from 40%, 60%, and 100%. This indicates that training your LLM without the correct corresponding context at times can be beneficial for the downstream task of answering questions related to the documents. In our training setup, we include four distractor documents alongside the oracle document, and at test time, we maintain this format by providing the oracle document with four distractors. Our findings suggest that, for domain specific RAG tasks, including a certain percentage of training data without the oracle documents in the context proves to be advantageous.

HotPot QA

Question: What screenwriter with credits for "Evolution"[0/1879]e a film starring Nicolas Cage and Téa Leoni?
Documents: ... David Weissman is a screenwriter and director. His film credits include "The Family Man" (2000), "Evolution" (2001), and "When in Rome" (2010).
...
The Family Man is a 2000 American romantic comedy-drama film directed by Brett Ratner, written by David Diamond and David Weissman, and starring Nicolas Cage and Téa Leoni. ...

RAFT: ##Reason: The screenwriter with credits for the film "Evolution," starring Nicolas Cage and Téa Leoni, is David Weissman. This information is provided in the reference documents which mention David Weissman as a screenwriter with film credits including "The Family Man" (2000), "Evolution" (2001), and "When in Rome" (2010). Therefore, the screenwriter for "Evolution" is David Weissman. ##Answer: David Weissman ✓
DSF: The Family Man ✗

Figure 3.4: **Comparison of RAFT and DSF:** We prompt RAFT and DSF fine-tuned models on the HotpotQA dataset. We can see that the DSF model extracts the wrong information from the context. For the question, who is the screenwriter, it responds with a film name. RAFT manages to get the result correctly .

3.5 RAFT Generalizes to Top-K RAG

After demonstrating the performance of RAFT on various benchmarks, we now study another important problem: How does the number of distractor documents in RAFT affect the model's performance when augmented with top-k retriever augmented generation (RAG) result during the evaluation? Previous research has highlighted the vulnerability of LLMs to irrelevant text (see studies [86, 125, 151]). This issue is particularly critical for LLMs + RAG since top-k RAG is frequently employed at test time to ensure high recall. Such a scenario necessitates the model to have the ability to discern and disregard irrelevant content, focusing solely on pertinent information.

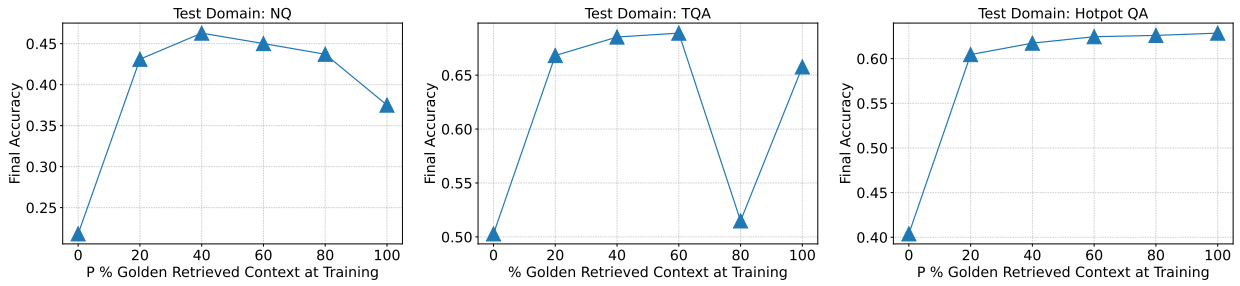

Figure 3.5: **How many golden documents to involve?** We study the hyperparameter $P\%$ which indicates what fraction of the training data contains the oracle document(s) in its context. Results on NQ, TQA and HotpotQA suggest that mixing a fraction of data that does not have the oracle document in its context is helpful for in-domain RAG.

Making Model Robust to top-K RAG

To tackle the challenge of enhancing large language models' (LLMs) ability to sift through irrelevant text within the retrieval pipeline, our analysis revealed that training solely with oracle (highly relevant) documents can inadvertently diminish the model's ability to discern and disregard irrelevant information. To address this, our algorithm, RAFT , adopts a strategy that integrates oracle documents with a mix of irrelevant ones. This methodology prompts us to investigate the ideal fraction of negative (irrelevant) documents to incorporate throughout the training process and to assess how well this training approach adapts to different volumes of documents encountered by the Retrieval-Augmented Generation (RAG) during the test phase. Our aim is to refine the balance between relevant and irrelevant information to strengthen the model's efficiency in identifying and utilizing pertinent content. Notice that Sec 3.4 looked at what P% of training data should include distractors, while in this section, we study test-time scenarios.

Training with negative documents. To enhance the robustness of LLMs against irrelevant text in retrieved documents, we adopted a finetuning approach that incorporates both golden (highly relevant) documents and distractor (irrelevant) documents. The model was trained with varying numbers of distractor documents, but consistently evaluated using the top-k documents obtained from the retriever - not to be confused with p.

Our findings, detailed in Fig. 3.6, reveal that finetuning with only the oracle document frequently results in inferior performance compared to configurations that include a greater number of distractor documents. As we can see in the figure, the better performance for Natural Questions is training with $D^* + 3D$ and it is $D^* + 1D$ documents with Hotpot QA. This insight has been particularly beneficial for our algorithm, RAFT . In our experiments, we typically employ a training setup consisting of 1 oracle document alongside 4 distractor documents. This approach strikes a balance, ensuring the model is not overwhelmed by distractors while still gaining the ability to effectively discern and prioritize relevant information.

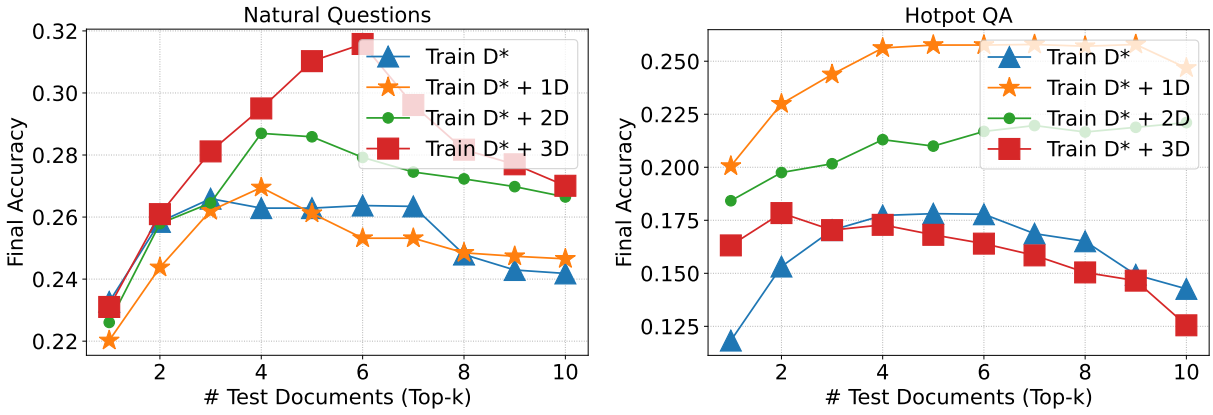

Figure 3.6: **Test-Time Documents Varying:** We study how robust RAFT is to varying numbers of test-time documents that a retriever might provide. In NQ, we find that training with 4 documents leads to the best performance, but training with 2 documents is optimal for HotpotQA. However, across both datasets, training with all datasets consisting of *oracle* documents hurts performance.

Generalization to a variable number of test-time documents. We extended our research to examine the impact of different quantities of test-time documents on the model's performance. Specifically, our experiments focused on assessing how models, trained with varying numbers of distractor documents, respond to changes in the number of documents presented at test time.

The results, illustrated in Fig. 3.6, confirm that the inclusion of distractor documents during training indeed makes the model more resilient to fluctuations in the number of documents encountered during testing. This ability to maintain consistent performance despite variations in test-time document numbers further validates the robustness of our approach, RAFT . This finding underscores the importance of a well-calibrated training environment to prepare the model for a range of scenarios it may encounter in real-world applications.

3.6 Related Works

Retrieval-augmented language models. RAG enhances language models by integrating a retrieval module that sources relevant information from external knowledge bases, significantly improving performance across various NLP tasks, including language modeling [10, 13, 48, 63, 81, 127, 128, 144, 160] and open-domain question answering [57, 74]. This integration follows a "retrieve-and-read" paradigm where the retrieval module provides additional context from external sources, which the LM then uses to generate the final output. The retrieval process involves using the input as a query to fetch documents, which the LM

incorporates for final predictions. For instance, Atlas [57] fine-tunes T5 models with the retriever, treating documents as latent variables, while RETRO [13] modifies the decoder-only architecture to include retrieved texts and conducts pre-training from scratch. kNN-LM [63] interpolates between the LM's next token distribution and distributions computed from retrieved tokens at inference. [114, 128] assume black-box access to an LM and combine it with either off-the-shelf or fine-tuned retriever.

Memorization. A key question around large neural language models is whether they truly "understand" text [37, 110] or simply rely on surface pattern memorization [20, 135]. [19, 20, 37] develop methodologies to quantify the extent of memorization in neural models. [15, 90, 110] further explored how memorization impacts the models' generalization capabilities. Recently, a seminal work by [21, 126] demonstrated the ability of language models to memorize and regurgitate training data, raising significant privacy concerns [61, 103].

Finetuning of LLMs. Recent years have seen rapid progress in developing large-scale language models (LLMs) [1, 6, 15, 139, 139, 154]. To adapt these foundation models to downstream tasks, fine-tuning [30, 58, 81, 94, 95, 118, 172] has become a prevalent approach. Traditional supervised fine-tuning may be limited by the cost and compute required for adapting LLMs. Addressing these challenges, research in the realm of parameter-efficient fine-tuning [51], such as Prompt Tuning [73], Prefix-Tuning [77], P-Tuning [87] and Low-Rank based fine-tuning [52], has gained traction. These methods enable LLMs to acquire domain-specific knowledge and adapt to specialized tasks such as question answering, summarization, and dialogue generation. Another branch of finetuning is through RLHF [85, 102, 113, 169], which adopts RL to align LLM's preference with human.

Finetuning for RAG. More recently, several papers have been exploring the idea of finetuning a pretrained LLM to be better at RAG tasks [82, 89, 144, 160]. These works focus on constructing a combination of finetuning dataset for RAG and train a model to perform well on these tasks. In particular, in their settings, at test time, the domain or documents can be different than the training time; whereas our work studies a slightly opposite scenario where we only care about testing the LLM on the same set of documents.

3.7 Conclusion

RAFT is a training strategy designed to enhance the model's performance in answering questions within a specific domain, in "open-book" settings, and demonstrates a fine-tuning recipe for LLMs for question-answering tasks based on a selected collection of documents. Our design decisions include training the model alongside distractor documents, organizing the dataset so a portion lacks oracle documents in their context, and formulating answers in a chain-of-thought manner with direct quotations from the relevant text. Our evaluations on PubMed, HotpotQA, and Gorilla API Bench show RAFT's promising potential. Looking

forward, we anticipate that in-domain RAG will continue to gain interest within both industry and academia. Unlike general RAG, our work addresses practical scenarios where LLMs are tasked with answering questions using domain-specific knowledge. Our findings suggest that smaller, fine-tuned models are capable of performing comparably well in domain-specific question-answering tasks, in contrast to their generic LLM counterparts.

Chapter 4

GoEx

4.1 Introduction

A world where LLMs can be utilized to be far more useful than they are today—one where they can manage schedules, automate tasks, and interact with a wide range of services—is on the horizon. Such LLM-powered applications have the opportunity to provide higher utility to the user but require LLMs to have access to user accounts across multiple services, necessitating the handling of sensitive user data (e.g., user credentials). Existing systems from major LLM providers avoid storing any sensitive data at the expense of functionality. For example, GPT plugins [101] requires a user to copy-paste the LLM output (e.g., a grocery list) into the target application (e.g, Instacart), which is not only cumbersome for the user but limits how autonomous the LLM can be.

The fundamental challenge in enabling LLMs to operate autonomously on user accounts is that the LLM is untrusted, yet it needs to run in a trusted or authorized user account. This presents a significant security risk, as the LLM may execute tasks that do not align with user intentions due to their inherent unreliability [115, 170] and the ambiguous nature of human language [55, 157]. Further, in non-autonomous settings, humans verify the correctness and appropriateness of the LLM-generated outputs (e.g., code, functions, or actions) before putting them into real-world execution. This poses significant challenges as code comprehension is well known to be notoriously difficult.

We argue that in many cases, "post-facto validation"—verifying the correctness of a proposed action after seeing the output—is much easier than the aforementioned "pre-facto validation" setting [106]. The core concept behind enabling a post-facto validation system is the integration of an intuitive *undo* feature, and establishing a *damage confinement* for the LLM-generated actions as effective strategies to mitigate the associated risks. Using this, a human can now either revert the effect of an LLM-generated output or be confident that the potential risk is bounded. We believe this is critical to unlock the potential for LLM agents to interact with applications and services with limited (post-facto) human involvement. Our open-source runtime for executing LLM actions, Gorilla Execution Engine (GoEx), solves

this by relying on existing primitives for API invocations in databases (`transactions`), file-systems (`git`), and RESTful calls (by maintaining a pair of actions and undo-action pairs). This is discussed in Section 4.3.

With *undo* semantics addressed, the question then lef to address is one of damage confinement: *How can we limit the damage that an untrusted LLM can cause when operating on behalf of users' accounts?*

Consider the following example: Alice wants to ask her LLM-powered digital assistant to check her email for any upcoming deadlines. The LLM needs access to Alice's email account to read her emails, and the straightforward approach is to give the LLM direct access to Alice's email credentials. However, this approach has several drawbacks:

1. **Unreliable and untrustworthy LLM execution.** LLMs are inherently unpredictable and may execute tasks that do not align with user intentions [164]. For example, Alice's LLM, though tasked with reading her email, may also mistakenly send a sensitive email to the wrong recipient (e.g., her manager).

2. **Centralization of user credentials.** The LLM provider becomes a central point of attack, as it has access to all user credentials across all applications.

Given the nascent state of the field, there is little work on enabling LLMs to operate autonomously on behalf of users' accounts and to the best of our knowledge, existing systems do not address these challenges. For example, SecGPT [157] focuses on isolating LLM-powered applications from each other to secure the overall system, but assumes that there is a *trusted* LLM that acts as the planner interpreting whether the users' requests require invoking an app or an LLM. Given the unreliability of LLMs, the planner could end up sending an email to the Alice's manager instead of reading an email. Industry offerings such as Rabbit's R1 [117] require the user to provide credentials to their server. Although existing software runtimes like web browsers and mobile operating systems run untrusted code (e.g., browser extensions, mobile apps), these systems are not directly applicable because LLM-powered systems involve an untrusted and unreliable LLM executing general-purpose tasks, whereas historically, browser plugins have been designed for specific functionalities. There is also a rich line of work centered on improving trustworthiness and reliability of an LLM, which while shows promise, does not provide rigorous or fully reliable guarantees [153]. Despite the rapid progress in improving LLMs, there remains a non-trivial risk that these models might still execute tasks in ways that misalign with user intentions or produce harmful outputs [170].

Instead, we draw inspiration from classical systems security design principles, such as enforcing least privilege [32], separating the credentials from the LLM provider [91], and confining the LLM against the source of ground truth (i.e., human approval) [84]. In this chapter, we propose GoEx, a *runtime* for enabling autonomous LLM-powered applications while confining the damage that an untrusted LLM can cause. Central to GoEx is our novel architecture (illustrated in Fig. 4.1), which incorporates these security principles to ensure that the LLM-powered application operates strictly within the boundaries set by the user.

First, we separate user credentials from the LLM, ensuring that the LLM provider never sees user credentials. Instead, we store user credentials in a "secure vault" that is only accessible to the runtime. We then sandbox the LLM by only allowing it to access the user's accounts *through the runtime*, which enforces the permissions granted by the user. The runtime expects the LLM to produce actions (e.g., API calls) and the scope of permissions needed to execute those actions. We enforce permissions at the granularity granted by the application provider (e.g., OAuth 2.0 [50] scopes). The runtime ensures that the LLM operates strictly within the boundaries set by the user.

We treat the user as the source of ground truth, and we take inspiration from mobile operating systems [38, 39, 40, 41, 150], which have well-studied paradigms for implementing user permissions. Our system asks the user for permission before the LLM executes an action. We map the permissions needed for an action to human-readable descriptions. In order to map API calls to their respective scopes, we leverage self-consistency [145] in order to improve robustness and a-priori knowledge of scopes the user has pre-approved. The user can then grant or deny the permission based on the description, has the option to grant permissions permanently or for a single use, and can revoke permissions at any time.

We illustrate our approach with the previously mentioned example: consider an LLM-powered personal assistant that reads a user's email and instead sends a sensitive email to the user's manager. The LLM generates an action to send an email to the user's manager, and the runtime maps this action to the permissions needed to send an email. The runtime then asks the user for permission to send emails on behalf of the user. The user can grant or deny the permission based on the description provided by the runtime, and since the user's intention was to have the LLM read an email, they can prevent the sensitive email from being sent to the user's manager.

In summary, we make the following contributions:

- We propose a novel architecture for enabling autonomous LLM-powered applications while confining the damage that an untrusted LLM can cause (Section 4.3).
- We enforce the principle of least privilege by separating user credentials from the LLM (Section 4.4) and confining the LLM against the source of ground truth, i.e., human approval (Section 4.6).
- We implement GoEx, a runtime that maps LLM-generated actions to permissions (Section 4.5), and enforces the permissions granted by the user.

We evaluate our approach on 10,204 APIs across 595 scopes from 250 popular services including BigQuery [47], Blogger, DNS, Google Chat, Google IAM, Gmail, Redis, Slack, Spanner [33], Uber, Youtube, etc. We find that our techniques for mapping user intent to permission scopes and APIs leads to excessive permissions to be granted in only 0.5% of the scenarios even if the user approves every scope request.

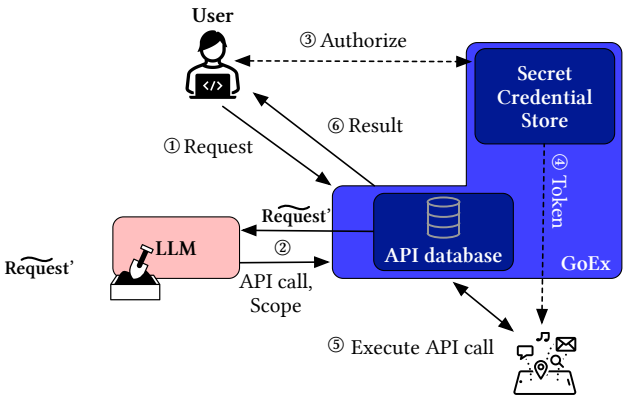

Figure 4.1: **GoEx workflow.** GoEx "sandboxes" the untrusted LLM by only allowing it to access the user's accounts through the runtime, which enforces the permissions granted by the user. Depending on the authorizations the user has granted, GoEx may or may not ask for further permissions. Based on the user's response to the permissions request, GoEx executes the API call (dashed lines).

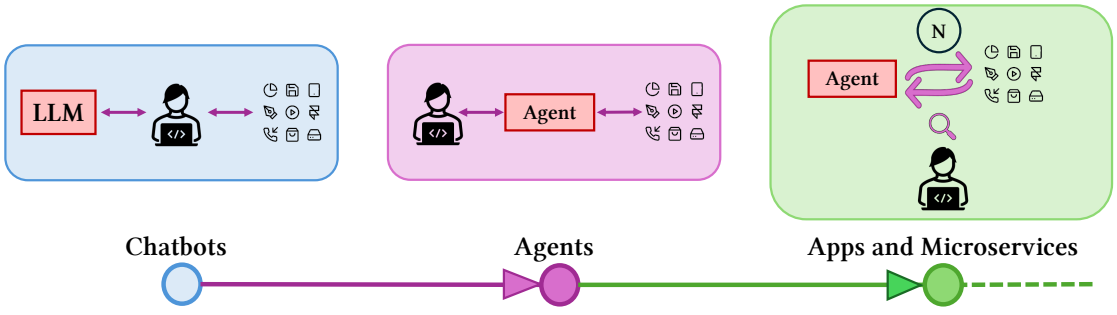

Figure 4.2: **Evolution of LLMs powered applications and services.** From chatbots, to decision-making agents that can interact with applications and services with human-supervision, to autonomous LLM-agents interacting with LLM-powered apps and services with minimal and punctuated human supervision.

4.2 Background and motivation

In this section we start-off by presenting an overview of the evolution of LLM-powered systems. We then present challenges from such a system to motivate the need for a least-privilege runtime for LLMs.

Evolution of LLM-powered systems

The evolution of LLM-powered applications is illustrated in Fig. 4.2. LLMs have advanced from simple chat bots to become actionable, decision-making agents that can engage with applications and services under human oversight, and are progressing towards a future where autonomous agents interact with other LLM-powered apps and services with only sporadic and limited human supervision.

Passive, read-only applications

LLMs have transformed the landscape of human-computer interaction. With early adoption as chat bots, these models were designed to mimic human conversation, allowing users to interact with computers in natural language. This era of LLMs focused on understanding and generating answers, serving as a bridge for humans to interact with vast amounts of web data in a more intuitive way. Early implementations were primarily used in the *read-only* model for information retrieval where the LLM did not make any state-ful changes [17], laying the foundation for more sophisticated applications.

Actionable LLMs

Increasing trustworthiness [143], wide-spread know-how of fine-tuning models, and novel techniques [107, 121] have expanded the role of LLMs from passive providers of information to active agents capable of executing simple tasks. These agents, powered by LLMs, can interact with applications, services, and APIs to perform actions on behalf of the user. This shift represents a significant leap in the capabilities of LLMs, enabling them to contribute actively to workflows and processes across various domains.

As LLMs become more integrated into daily workflows and systems, they are expected to play a significant role in enhancing functionality and adaptability across various domains [14, 26, 27, 54, 72, 99, 104, 105, 155, 171]. However, this evolution also brings challenges, which we elaborate on in the next subsection.

Challenges with integrating LLMs

Unreliability

LLMs, especially closed-source and continuously-updated models from third parties (e.g., ChatGPT [100], Claude [8], and others), introduce new challenges in ensuring the reliability

and correctness of the overall system. The integration of LLMs into software systems challenges traditional paradigms of unit testing and integration testing. While closed-source and continuously pre-trained LLMs from third-parties pose a new challenge of the model constantly changing, running in-house LLMs is no panacea either. Given the dynamic, often unpredictable nature of LLM outputs, establishing a fixed suite of tests that accurately predict and verify all potential behaviors becomes increasingly difficult, if not impossible.

Protecting sensitive data

In order for LLMs to interact with a user's accounts across multiple applications, the LLM must be able to reason about having access to credentials, thereby, granting access to the user's accounts. When an LLM is hosted by an (untrusted) external service, it is desirable to not directly pass any credentials or sensitive data to the LLM while still preserving functionality of the LLM-powered system. As mentioned in Section 4.2, LLMs can also generate (untrusted) code and are susceptible to hallucinations [115, 170]. This can result in running potentially malicious code or inadvertently performing actions that were not intended by the user.

Case study: Checking emails

We now revisit the use case mentioned in Section 4.1 where Alice is using an LLM to check her email. Alice asks her LLM-powered digital assistant to check her email for any upcoming deadlines. The LLM needs access to Alice's email account to read her emails and produce a summary. As LLMs can be unreliable (Section 4.2), the LLM may mistakenly send a sensitive email to the wrong recipient (e.g., Alice's manager) while reading her emails.

4.3 System Design

In this section we first present the threat model (Section 4.3) for our runtime, and then describe our system (Section 4.3).

Threat model

We consider an LLM-powered system that executes the outputs of the LLM—which may call external tools or services—on behalf of the user. We assume that the LLM is not malicious, but it is unreliable and error-prone, since LLMs can exhibit unreliable and unpredictable behavior [5], and natural language can be ambiguous [55, 157]. The LLM may produce outputs that are not aligned with the user's intentions, and may make mistakes. We consider the LLM to be untrusted, as it may execute tasks that are not intended by the user. We consider the user to be the source of truth; that is, we consider permissions authorized by the user to be within the "safe" scope of operation for the LLM.

In the presence of an unreliable LLM as previously described, we desire the following security guarantees:

1. **Decentralized user credentials.** User credentials should not be centralized anywhere in the system. There cannot be a single point of failure in which all users' secrets can be compromised.

2. **Confidentiality of user credentials.** The LLM provider should not learn user credentials and secrets through prompts at inference time.

3. **Safe execution of LLM output.** The LLM's output should be safely executed. We define *safe* execution of LLM output as execution that is either (1) in alignment with the user's intentions; (2) can be undone; or (3) operates strictly within the boundaries permitted by the user.

We assume that the service providers implement their APIs correctly and enforces permissions within their services. We also trust each service provider with the documentation of permissions and function calls within their API.

Reversibility and Undo Semantics

When possible, actions executed by an LLM should give users the right to *undo* an action. This approach may require maintaining multiple versions of the system state, leading to high costs in terms of memory and computational resources. Furthermore, the feasibility of implementing undoing an action is often dependent on the level of access granted to the system. For instance, in file systems or databases, where root access is available, undoing actions is possible. However, in scenarios where such privileged access is not granted, such as in email clients like Gmail, the ability to undo an action may be limited or require alternative approaches. One potential solution is for the runtime to make a local copy before deleting, which introduces additional state to the runtime but enables *undo* for email deletion.

To account for the resource costs associated with maintaining multiple system states, we adopt the notion of a *commit*, also called a "watermark" in streaming data-flow systems [3, 18]. By grouping together sets of actions based on their associativity, commutativity, and distributive properties, it may be possible to define checkpoints at which the system state can be saved or rolled back. This approach would enable selective undoing of actions within a defined scope, rather than maintaining the ability to undo every individual action.

System overview

GoEx's architecture (Fig. 4.1) consists of three main components: the *Secure Intelligence Vault*, the *Least-Privilege Mapper*, and the *Execution Runtime*. The Secure Intelligence Vault stores user credentials and secrets, ensuring that they are never centralized. The Least-Privilege Mapper maps LLM-generated actions to permissions, ensuring that the LLM

operates within the boundaries set by the user. The Execution Runtime executes the LLM-generated actions, enforcing the permissions granted by the user. The *Execution Runtime* is responsible for presenting undo semantics, and actually executing all the API calls. We describe this in Section 4.7.

We illustrate the workflow of the system in Fig. 4.1, and explain with the aforementioned example (Section 4.2): ① Alice asks GoEx to check her email for any upcoming deadlines. ② GoEx then sends this request to the LLM which generates an action to read Alice's emails and the permission scope(s) needed to execute the action. In this case, assume the LLM generates an action to read Alice's emails and the permission scope needed is the read-only permission. ③ The GoEx runtime requests the read-only scope from the secret vault, and the vault asks Alice for permission to read her emails on her behalf. ④ Alice can grant or deny the permission based on the description provided by the runtime, and authorizes the read-only scope, providing the runtime with the corresponding token. ⑤ The GoEx runtime fills in the API call with the token and executes the action to read Alice's emails, and sends the output of that action to the LLM, which then reads the emails and produces a summary of upcoming deadlines. ⑥ GoEx returns the resulting summary to Alice.

4.4 Separating user credentials from the LLM

In order to ensure that user credentials are never centralized, we store them in a secure vault that is only accessible to the runtime. The LLM never has access to user credentials, and the runtime only accesses the vault when executing LLM-generated actions. This separation ensures that user credentials are never exposed to the LLM provider, even in the presence of a compromised LLM provider.

Authentication

GoEx provides a secure way to handle user secrets, whether using OAuth2 for token-based authentication or API keys for direct service access. GoEx acts as the secure intermediary to facilitate authenticated actions across various services. For OAuth2, GoEx sits between the user and services, facilitating the necessary relay to retrieve access tokens. These tokens allow users to delegate the GoEx system to perform actions on their behalf. For other services that authenticate accounts through API keys, GoEx provides an interface that allows users to insert and retrieve them.

Symbolic permissions

In order to hide the actual permissions from the LLM, we use symbolic permissions that are mapped to actual permissions at runtime. The LLM generates actions that are associated with symbolic permissions, and the runtime maps these symbolic permissions to actual

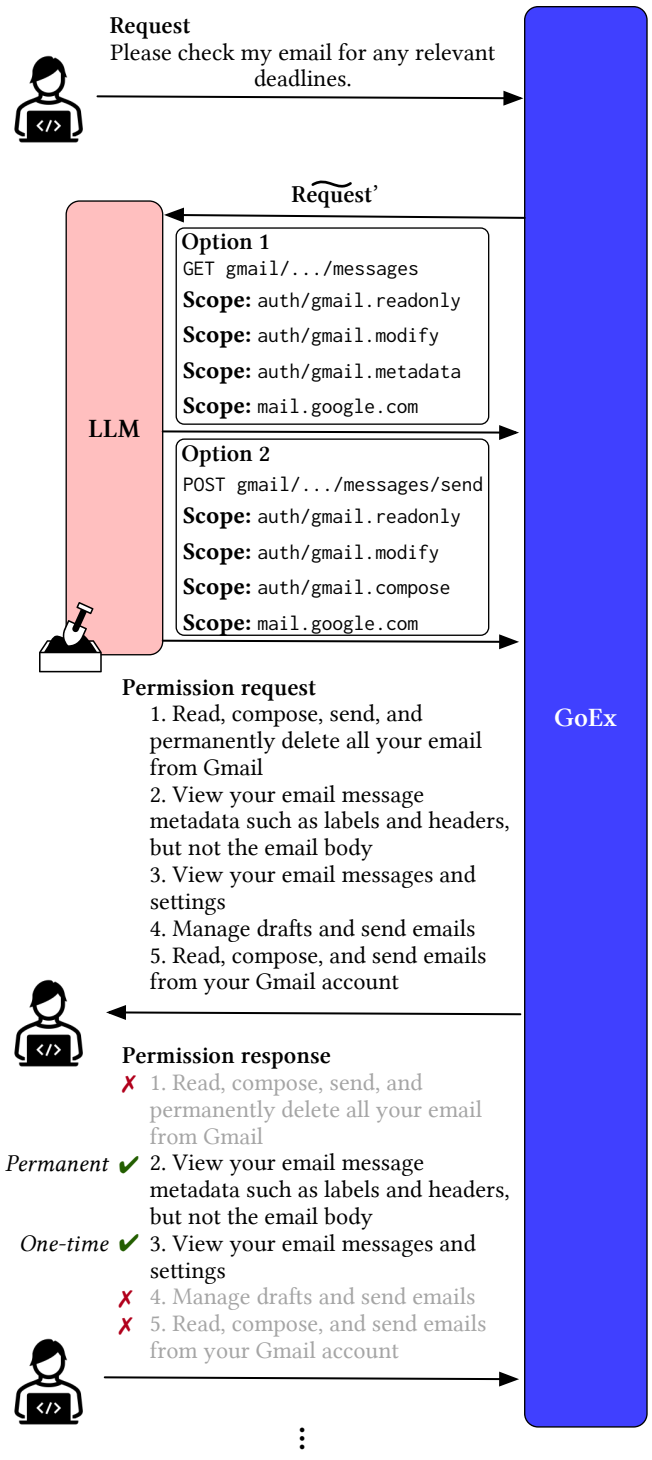

Figure 4.3: **Example of the email use case** (Section 4.2) and the permissions needed to execute the action. The least-privilege mapper (Section 4.5) maps the action to the permissions needed to execute the action. The runtime then requests the permissions from the user (Section 4.6), who, in this example, grants all read-only permissions but denies any write permissions.

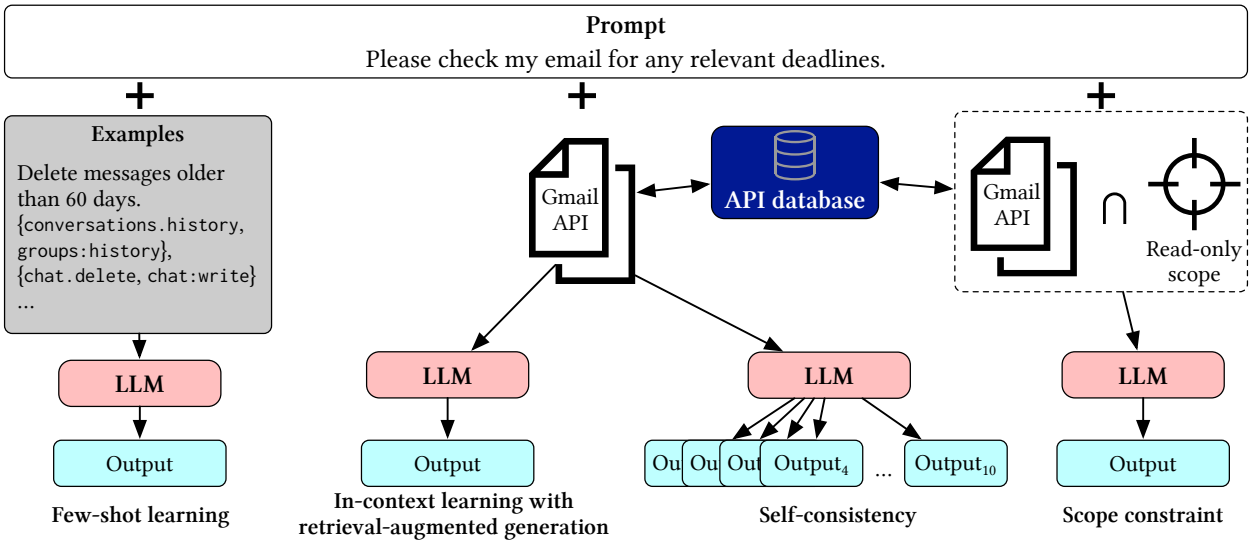

Figure 4.4: **Different techniques to map user intent to set of APIs and scopes.** RAG-based techniques of in-context learning and self-consistency perform much better than our baseline of few-shot learning. In RAG, the user prompt is augmented with relevant documents from a database. The scope constraint improves user-experience by lowering the number of times we request users for additional scope required to accomplish a task.

permissions based on the user's authorization. This ensures that the LLM does not have direct access to the user's permissions, and that the permissions are enforced by the runtime.

Storing secrets

User secrets and keys are stored locally on the user's device in a secret credential store (SCS), which maps `service_name` to `key` and `format`. When the user wishes to interact with specific service(s), the corresponding keys are requested from the SCS. The `format` specifies how the keys are stored (e.g., file or string). The role of the SCS is to selectively retrieve just the required keys for a given execution. For example, if a user wants to send an email invite to their friend for lunch, the agent only needs their OAuth2 token for their email provider's API, and not, for example, their key with scope associated with reading their bank accounts.

4.5 Least-Privilege Mapper

GoEx adopts the principle of least privilege to grant the run-time only those permissions which are essential for its intended function. However, mapping the user intent to a set of APIs and scopes is challenging, due to the unpredictability of LLM actions and the ambiguity of human language.

Challenges

Creating a least-privilege mapper requires overcoming the following challenges:

1. **Complexity of scope definitions.** The integration of LLMs into service operations involves navigating an extensive array of permissions. Each service, such as Google Workspace [43] or Microsoft Azure [93], defines numerous scopes (e.g., permissions or access levels) that can be granted to an application. These scopes range from reading basic user profile information to modifying sensitive data or configuring service settings. For example, a service like Google Drive may offer scopes from solely reading files to complete management over all of a user's files.

2. **Challenges in mapping actions → scopes.** Determining the appropriate scope for a particular task is further complicated by the fact that for many services, multiple {API, scope} pairs could potentially accomplish the same task.

3. **Hierarchy of scopes.** Moreover, the absence of a formal hierarchy or clear granularity in scope levels further complicates the enforcement of principle of least-privilege. For example, it is often *not* straightforward to determine whether an "owner" privilege is more or less restrictive than an "admin" privilege because these terms' semantics can vary significantly across different services.

Strawman: Static mapping of actions → scopes

One can overcome the above challenges to some extent by having a predefined, static set of {API, scope} mappings for a set of APIs. However, supporting a new set of APIs would require updating this mapping, which limits the generality of our runtime. We elaborate on the desired properties of our least-privilege mapper in the next subsection.

Desired properties

A. **Generality and flexibility.** A key requirement for the GoEx system is generality. The LLM should be capable of extending its functionality to APIs that it was not explicitly trained on. This requires a flexible and dynamic mapping system that can interpret the intended action from the natural language query and map it to the most appropriate and minimal scope required to perform that action securely.

B. **Scope reuse.** Further for agentic systems, it is ideal to re-use the same scopes that were granted before. For example if the user has already granted GoEx scope $\{scope_a, scope_b\}$ for a service, then, if there are two alternative API, scope pairs for accomplishing the user task say $\{API_1, scope_a\}$ and $\{API_2, scope_c\}$ than the GoEx system should ideally choose $\{API_1, scope_a\}$.

Dynamically mapping LLM-generated actions to permissions

To address the challenges and achieve the desired properties listed above in mapping user intent to a set of APIs and scopes, GoEx leverages recent advancements in LLMs. Specifically, we explore the following techniques: few-shot prompting [15, 112], retrieval-augmented generation (RAG) [75], and self-consistency [145]. We depict these techniques in Fig. 4.4. Few-shot prompting [15, 112] introduces several example pairs of requests and the corresponding {API, scope} pairs to guide the model in the formatting of the response. We first provide a brief overview of each technique and then explain in more detail their respective subsections.

In the few-shot setting, we do not provide the documentation for the API calls and instead rely on the model's extensive training to already cover the APIs of interest (4.5). Retrieval-augmented generation (RAG) [75] relies on the retrieved APIs' documentation for the likely APIs to help guide the model to invoke the API correctly. RAG relies on an retrieval process to identify APIs that might be relevant to the user's request based on their service. However, it is possible that the correct API may not be retrieved and that the retrieve APIs could be wrong or distracting (Section 4.5). We use few-shot learning and RAG as baselines to implementing the least-privilege mapper.

To improve robustness of the LLM's generations, we also explore the using self-consistency [145], which leverages the stochasticity of the generation process to sample multiple possible API and scope selections and then select the most likely from the set of k samples (Section 4.5). Further, the user could have approved certain scopes a-priori, or in a previous interaction. Ideally, GoEx would find APIs that can accomplish the user's task without requesting additional scopes (Section 4.5).

Baseline: Few-shot learning (no RAG)

In this approach we utilize 2 in-context examples to guide the LLM in generating API calls and the surrounding code to generate them. These examples serve as templates to follow, but do not include any relevant information about the service or the API. The examples chosen are fixed independent of user's intended action. Using 2 in-context examples, also known as few-shot learning, is a technique to ground the format of LLM generation [15]. This is the second most popular way to use the LLM, besides the most popular way to naively prompting the LLM with the intended goal a-la ChatGPT. If one were to naively prompt GPT, we find the performance to be quite poor since the output is not grounded. For example, the model can start a verbose monologue without giving any code or API as output. The technique we adopt and this method tests the LLM's ability to infer correct actions from a limited set of closely related precedents and it's pre-training dataset.

Baseline: In-context learning with RAG

In this technique, we enhance the LLM's context with a retrieval-augmented generation mechanism. The system includes all relevant APIs in the context (prompt) to provide a rich option-set, which the LLM can draw from. This improves the accuracy and relevance of the

API call generation by leveraging a broader base of knowledge. To construct such a system, we first consider the prompt and find the similarity of the prompt to a service. For example, for a user prompt of *"I would like to send a message to @Bob on Slack"*, GoEx will fetch all the APIs of the Slack service and append that to the user's prompt before feeding that into the LLM. The LLM can then pick the most relevant API for inference. The mapping of prompt to service is relatively straight-forward, and the techniques range from sub-string matches to using an AI based solution [107, 111]. Similarly, in GoEx we include all the APIs from a service, although potential optimizations include filtering the set of APIs based on semantic similarity. Both of these are out of scope from the view of this work.

GoEx self-consistency

Self-consistency [145] is a recently introduced state-of-the-art technique that improves the LLM's ability to reason in math. We demonstrate how self-consistency can be adopted in our domain of mapping scopes and APIs to user intent. Under the self-consistency approach, the input prompt to the LLM remains the same - the system generates k possible responses, and then finds the scopes with the largest agreement from all the k generations. We demonstrate how this technique can reduce errors and increase confidence in the selected API's scope by emphasizing consistency across multiple generated outcomes. k is a hyperparameter, and we demonstrate the sensitivity to different values of k with the best performance at $k = 10$.

GoEx introducing scope constraints

Often, in the course of the user interactions with GoEx the user could grant scopes for a period of time. In such scenarios, where GoEx already has access to certain scopes, we pre-select a scope based on the intent before generating the API call. Here, the LLM is tasked with generating an API call that fits within the predefined scope, focusing the generation process and we evaluate if the LLM could choose APIs that could be executed with requiring additional scopes.

4.6 Asking the user for permission

In GoEx, users are the ultimate arbitrators in determining which scopes and for what service can be delegated to the runtime. An integral part of this is to present users with appropriate information to make informed decision.

Human-readable descriptions of permissions

We make the observation that most services have a well-documented API, and the corresponding scopes are mapped to human-readable descriptions. We scrape the documentation of the service to generate human-readable descriptions of the permissions needed to execute

an action. We then ask the user for permission to execute the action, providing the user with the human-readable description of the permissions needed.

As shown in Fig. 4.3, the permission request sent to the user is a human-readable description from the Gmail API documentation [44]. For example, the scope `mail.google.com` is mapped to the description, *"Read, compose, send, and permanently delete all your email from Gmail."* Since Alice only wanted to read her emails, she can deny the permission request for this scope. Note that a least-privilege permission implies that there is some ordering on permissions. We assume that this ordering is determined by the user as the user can grant or deny permissions based on the human-readable descriptions provided by the runtime. That is, a permission that the user grants is considered to be lower privileged than a permission that the user denies.

Permission granting model

We take inspiration from mobile operating systems [9], which have well-studied paradigms for implementing user permissions. This permission granting model is also present in SecGPT [157], which uses this to grant permissions *between* different LLM applications interacting with each other. We use this permission granting model in the context of the user granting permissions to an LLM-powered application to act *on behalf* of the user.

One-time grant

A one-time permission grant allows the LLM to execute an action using that permission once, and the permission is revoked after the action is executed. This is useful for actions that the user does not want the LLM to execute again.

Session grant

A session permission grant allows the LLM to execute actions using that permission for the duration of the user's request. The permission is revoked after the session ends.

Permanent grant

A permanent permission grant allows the LLM to execute actions using that permission indefinitely. The permission is not revoked until the user explicitly revokes it. This is useful for actions that the user wants the LLM to execute repeatedly.

4.7 Execution Runtime

GoEx's execution runtime supports a range of "actions" including RESTful API requests (Section 4.7), databases operation (Section 4.7), and filesystem actions (Section 4.7). Each action type, while initiated from a unified GoEx interface, are handled uniquely as described below.

RESTful API calls

We first describe how GoEx handles RESTful API calls (illustrated in Fig. 4.5).

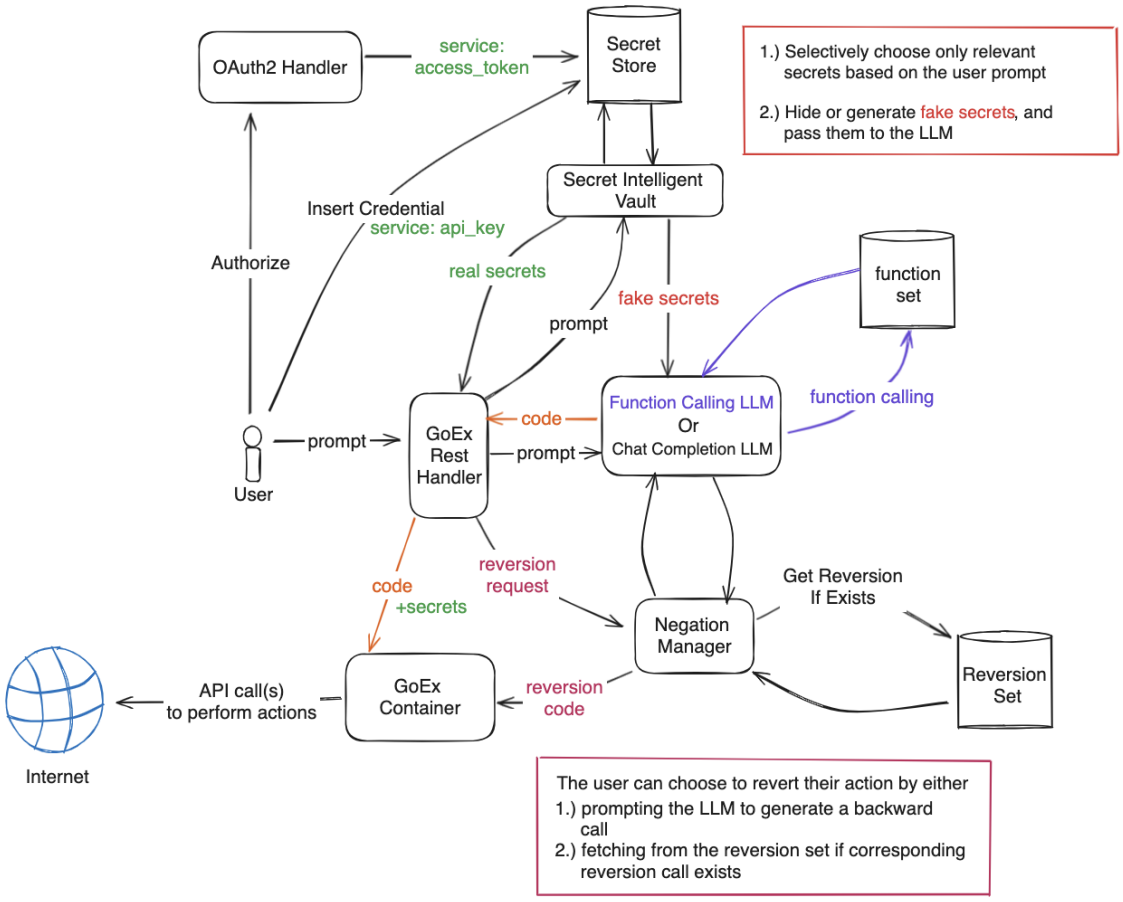

Figure 4.5: GoEx's runtime for executing RESTful API calls. Upon receiving the user's prompt, GoEx presents two alternatives. First, an LLM can be asked for an (`Action`, `Undo-Action`) pair. Second, the application developer can provide tuples of actions and their corresponding undo-actions (function calls) from which the LLM can pick amongst.

Authentication. GoEx provides a secure way to handle user secrets, whether using OAuth2 for token-based authentication or API keys for direct service access. GoEx acts as the secure intermediary to facilitate authenticated actions across various services. For OAuth2, GoEx

sits between the user and services, facilitating the necessary relay to retrieve access tokens. These tokens allow users to delegate the GoEx system to perform actions on their behalf. For other services that authenticate accounts through API keys, GoEx provides an interface that allows users to insert and retrieve them.

Storing secrets. User secrets and keys are stored locally on the user's device in a Secret Intelligent Vault (SIV). SIV maps `service_name` to `key` and `format`. When user wishes to interact with specific service(s), the corresponding keys are requested from the SIV. The `format` specifies how the keys are store, that is, in a file, or as a string, etc. The role of the SIV is to selectively retrieve just the required keys for a given execution. For example, if a user wants to send an email invite to their friend for lunch, the agent only needs their OAuth2 token for their email provider, and not, for example, their bank account's API keys. The policy used for SIV is user-defined and highly flexible; it could be as simple as parsing through the user prompt to detect which service's keywords are present, or as complex as a fine-tuned prompt-to-service retrieval model.

Generating actions. The GoEx framework supports two techniques to generate the APIs. In the Chat Completion case, assuming the user prompt is, "send a Slack message to `gorilla@yahoo.com`," the user must initially authorize GoEx to use their access token through the Slack browser. After receiving the user prompt, GoEx requests the SIV for the necessary secrets from the Secret Store. Slack secrets (OAuth2) are inherently hidden because they are stored as a file, so GoEx passes the file path along with the prompt directly to the LLM. GoEx mounts the Slack secret file and passes the LLM-generated code to be executed in the GoEx container. If the user wishes to revert the execution, the reversion call will be retrieved from the reversion set if it exists; otherwise, the handler prompts the LLM to generate it. If the user chooses Function Calling, instead of asking the LLM to come up with a command to satisfy the user's prompt, GoEx asks it to select a function from a user-defined function set and populate the arguments. Secrets will be chosen from the SIV similarly, and execution occurs in the GoEx container. If the user wishes to revert, another function from the function set will be chosen by the LLM.

Generating undo actions. First, we check if the reverse call for the action API is in the database `Reversion Set` as shown in figure 4.5. GoEx presents the systems abstractions, while developers are free to define the policies for mapping. For some APIs it might be critical to check for exact match for all parameters of the API. On the other hand for some other APIs, just the API name might be sufficient to uniquely identify what the reverse API would be. For example it is *not* sufficient to say the reverse of `send_slack_message` is `delete_slack_message`, since number of messages to delete could be an argument.

To populate such a mapping, first, we instruct the LLM to generate a reverse API call whenever the user attempts to perform an action. We recognize that this gives no guarantees, but the philosophy is that we allow the LLM to be wrong at most once. Post each new API,

the table is then if the reversion worked or not making this information available for future invocations. For applications that need guarantee, developers can pre-populate this table and combined with function-calling mode of operation, the system can be forced to only use those API's that are 'guaranteed' by the developers to be reversible.

Damage confinement. Often reversibility cannot be guaranteed. For examples sending an email isn't really reversible. For such scenarios, GoEx presents abstraction to bound the worst case. Currently, the way blast-radius-containment is implemented is through coarse-grained access control, and exact string match. First, GoEx looks at the user's prompt to determine the end service that they are then authorized to use. For example, a prompt of *I would like to send a slack message* would only need credentials for slack, and not, say, their bank. GoEx currently does this, through a simple sub-string check of the prompt, while giving developers the flexibility to adopt any mapping they might choose.

Execution. Once the API, and the set of credentials required are determined, the APIs are then executed in a Docker container for isolation.

Database Operations

GoEx leverages the mature transaction semantics offered by databases. This section describes the abstractions available, and the two default policies.

Abstractions

GoEx relies on the LLM to generate database operations, but there are two prerequisites needed to execute database operations: (1) knowledge of the current database state, and (2) knowledge on how to access the database. To provide these, `DBManager` class is used. This allows the database to readily minimally query for the database state (e.g. only the schema) to provide additional info to the LLM during prompting without leaking sensitive data. It also tracks the connection configuration to the database so that connections can be established without leaking credentials to the LLM as an untrusted third-party by asking the user to store the credentials locally, and after the LLM generates the operation, GoEx then executes the operation.

`DBManager` also assists the user store with storing a previous state. Here, the *commit* and *undo* actions are introduced where a *commit* means the user permanently saves the executed changes, and an *undo* reverses the aforementioned changes. Most modern databases also provide ACID guarantees [49], including NoSQL databases like DynamoDB and MongoDB, which we leverage to implement committing and undoing actions.

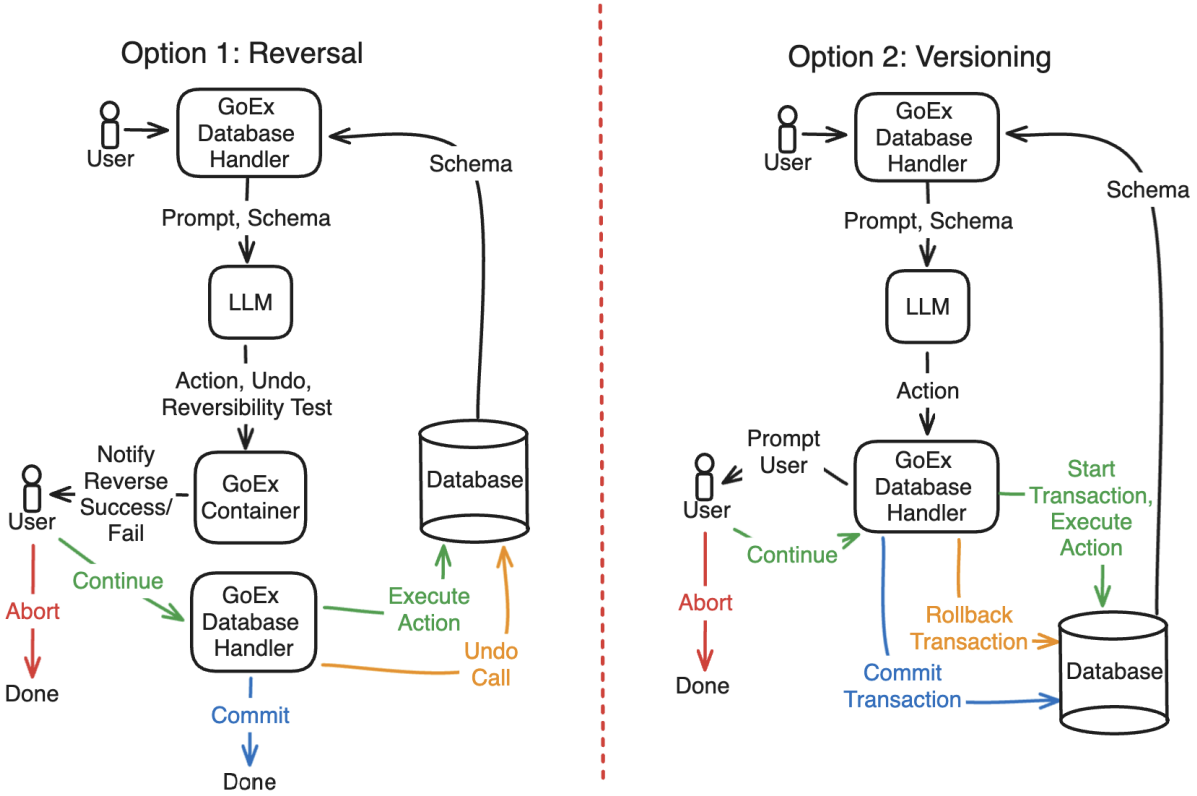

Figure 4.6: Runtime for executing actions on a database. We present two techniques to determine if a proposed action can be undone. On the left, for non-transactional databases like MongoDB, and for flexibility, we prompt the LLM to generate (`Action`, `Undo-Action`, `test-bed`) tuples, which we then evaluate in a isolated container to catch any false (`Action`, `Undo-Action`) pairs. On the right, we can provide a deterministic undo with guarantees by employing the transaction semantics of databases.

Policy

`DBManager` implements reversibility in two ways. The user chooses which one to use when they execute a prompt in GoEx.

1. **Option 1 (Reversal).** Makes use of a reverse database operation to perform the *undo*. It is done by prompting the LLM with the original operation (action call) along with the schema to generate the reversal operation (undo call). Committing requires no action, and undoing would just be performing the undo call after the action call is done. This option scales better as additional users can continue perform database actions without needing to wait for the previous user to finish their transaction at the cost of relying on the LLM to come up with an undo call, which may or may not have unexpected behaviors.

2. **Option 2 (Versioning).** Makes use of the traditional ACID transaction guarantees of the database and holds off on completing a transaction until the user specifies to do so, or rolls back to the previous state. Committing would involve committing the transaction, and undoing is synonymous to a rollback transaction. This branch is able to provide reversal guarantees that branch 1 cannot, at the expense of higher performance overhead.

Reversibility testing. Within Option 1, GoEx also performs a reversibility test to verify that the generated reversal operation indeed reverses the original operation. This requires a containerized environment to be separate from the original database to maintain the original database state. Since copying over the database into the container is very expensive, the approach is to ask the LLM to generate a bare-bones version of the database for reversibility testing, given the action, undo calls, and the database schema. The outcome of the test is sent back to the user for final confirmation before committing or undoing the operation. This method allows for efficient testing by decoupling the testing runtime from being scaled by the number of entries in the database.

File Systems

GoEx tries to present expressive abstractions to let LLM-powered systems to interact with file-systems using Git version control. To track the directory tree, on every GoEx filesystem-type execution, GoEx does an exhaustive, recursive walk of the directory and its sub directories and stores the directory structure as a formatted string.

Abstractions

Filesystems operation support in GoEx uses abstractions similar to what is used to support database operations. `FSManager`, is a filesystems manager that tracks (1) the directory tree structure with all filenames, and (2) the directory path that the user wishes to execute the filesystem's operations in. The tree structure, which is updated with executions, enables the LLM to generate operations that reflect the actual state of the user's filesystem.

Utilizing the relevant abstractions presented by journaling and log-structured filesystem for undo-semantics is left as future work, as the current GoEx system aims for compatibility.

Policy

The options are similar to the database case, where Option 1 is for reversals and Option 2 is for versioning. The largest differences are how `FSManager` carries out reversibility testing and that versioning is accomplished using Git.

Git. GoEx uses Git to perform versioning. Since Git is already a version-control system for files, it is a straightforward solution to use, but has several limitations. Git does not have the ability to version track outside of the directory that it was initialized in. GoEx limits

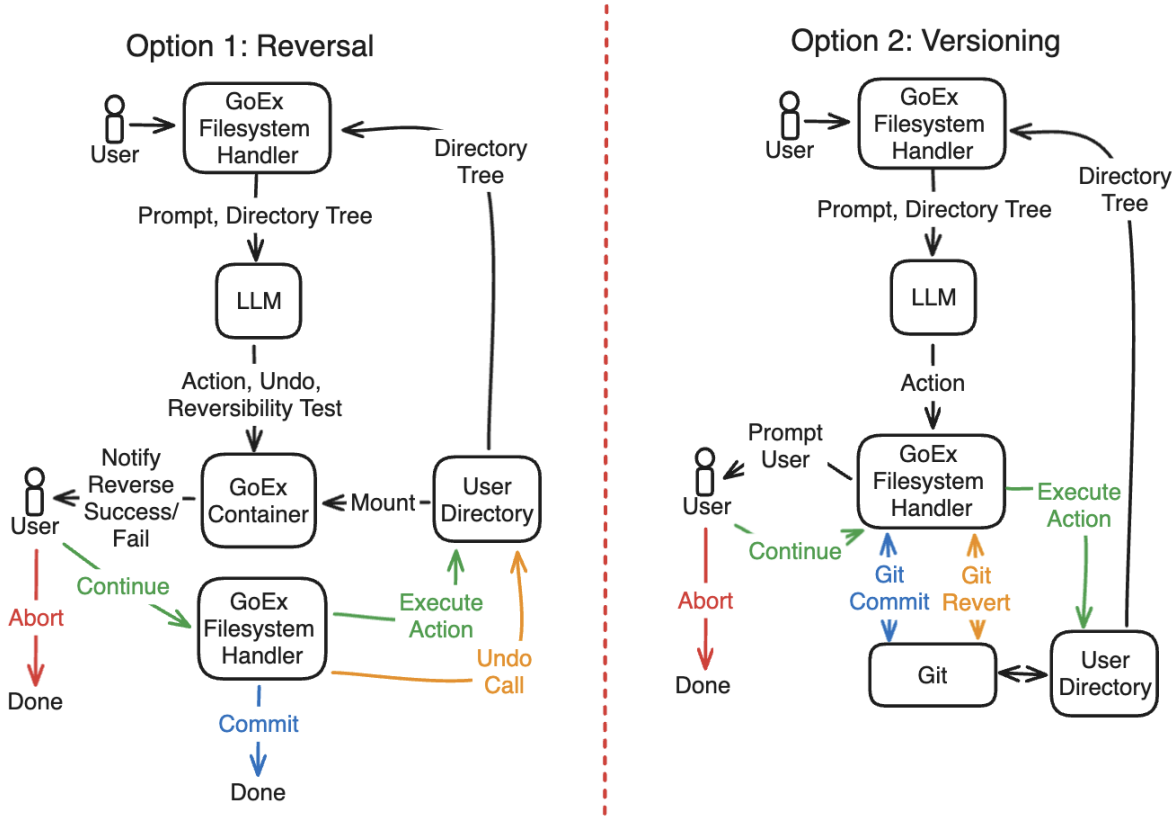

Figure 4.7: Runtime for executing actions on a filesystem. GoEx presents two abstractions. On the left, the LLM is prompted to come up with an (`Action`, `Undo-Action`, `test-bed`) which GoEx evaluates in a isolated container to catch any false (`Action`, `Undo-Action`) pairs. On the right presents deterministic guarantees by using versioning control system like Git or Git LFS.

the user execution scope to the specified path in `FSManager`—which is always inside of a Git repository—and its sub directories in accordance to our blast-radius confinement abstraction to prevent the LLM from performing arbitrary actions in undesired parts of the user's system. With larger directories, Git versioning can be expensive space-wise. GoEx leverages Git LFS for larger directories as an optimization. A threshold is defined for directory size that GoEx would then check whether or not to initialize Git LFS (200 MB by default).

Reversibility testing. Similar to supporting databases operations, the LLM generates the testing code using the action and undo calls, along with the directory tree. Inside the container, the specified path is mounted in read-only mode to again do blast radius containment. GoEx begins by duplicating the directory contents in the container, then run

the action and undo calls on the copied directory, and finally compare contents. Depending on the original operation, the content comparison can just be a check of filenames or an exhaustive file content comparison of all the files. We rely on the LLM to come up with the test-case. Unsurprisingly, here GoEx allows you to trade off guarantees for performance.

4.8 Evaluation

In this section, we ask the following questions to evaluate the effectiveness of the GoEx framework:

1. To what extent is GoEx able to protect against unintended scope access?
2. What is the performance regression when using this framework compared to a baseline scenario reflecting typical automation practices where the LLM is directly granted access to credentials?

Given the intended versatility of an LLM runtime, it is crucial to test GoEx's ability to consistently understand and manage permissions across a diverse range of applications. However, evaluating performance overhead and cost may focus on a single application, as these aspects are expected to be relatively uniform across different applications.

Services considered

To study how robust the system is to such a design, we considered 250 popular services including Slack, Uber, Google IAM, Google Chat, BigQuery [47], Redis, Blogger, DNS, Gmail, Youtube, Spanner [33], etc. These services were picked based on ease of access to their API documentation. For all the services, all the APIs were enumerated exhaustively. In total, they constituted 10,204 APIs and 595 scopes. We plot the number of APIs and scopes available for each service in Fig. 4.8. The set of APIs and scopes each service provider exposed is diverse, ranging from a single API with only one possible scope for Public Certificate Authority [45] to 778 APIs with tens of possible scopes for Google's compute service.

Generating the evaluation dataset

To evaluate the efficiency of our system, we first curate the APIs available and their corresponding scopes of all the services exhaustively. The data we collate for each API includes the URL endpoint, the method (`GET`, `PUT`, `POST`, `DELETE`, etc.), an example invocation, and the scopes—at least one of which is required to authenticate and successfully execute the API. Given this data, we then need to generate a dataset that maps potential human intents into an API call and scope pair. One example in our evaluation dataset is as follows: the user prompt is *Go through my slack messages with Bob and delete everything older than 60 days*. We then append all the APIs from slack as a part of the input prompt for the

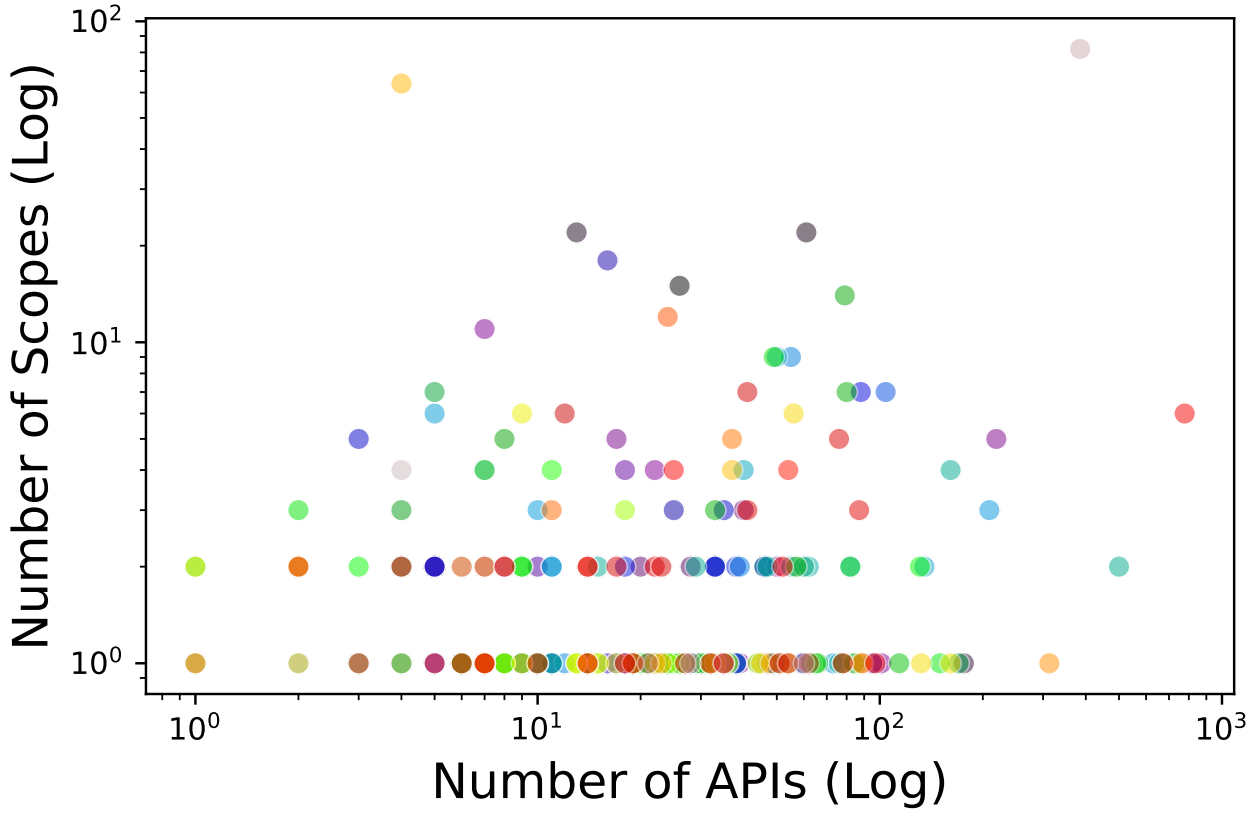

Figure 4.8: **Number of APIs available and scopes that can be chosen.** Each dot represents one of the 250 services, and its X-intercept corresponds to the number of APIs that service exposes, and the Y-intercept represents the unique number of scopes that service provides. In our dataset, services have between 1 to 778 APIs.

LLM to pick amongst. The expected output from the LLM are the `conversations.history` and `chat.delete` APIs with scopes `groups:history` and `chat:write` respectively. In this example, two APIs from the same service, Slack, are invoked. In all our experiments, we evaluate GoEx's ability to invoke at most three APIs from at most two services, which is a limitation of our resources to generate, and time to validate the evaluation dataset, and not a limitation of the underlying system GoEx.

To build this evaluation dataset we use the self-instruct paradigm [147]. First, we handcraft 6 examples of user prompts and their mapping to APIs and scopes that are needed to execute the respective API calls. Then for each evaluation data point we want to generate, we sample 3 of the 6 handcrafted examples randomly and include that in the prompt to guide the LLM on what needs to be done. This process is also called in-context learning [36]. We then present a general-purpose LLM, say GPT-4, with all the APIs of a service, say, slack, and ask the LLM to pick a (or a few) APIs and choose a potential user intent or prompt for which the chosen API and scopes would be the right answer. Notice that by doing so, we are using

the LLM to perform a very simple task of mapping the API to a question, as opposed to mapping the question (human prompt) to an API. This approach is a standard recipe for data generation [147]. The evaluation data generated in this fashion is not perfect. We then manually inspect each of the data points to ensure consistency between the question (human intent), and the corresponding API(s) and the scope(s). Self-instruct can be thought of as an aid for the humans for coming up with {prompt, API, scope} tuples. In total, our evaluation includes 2,790 unique data-points from 10,204 unique APIs across 250 services, with 595 possible scopes that can be granted.

Techniques to map intent to APIs and scopes

To evaluate the GoEx framework in its ability to map user intent to APIs and scopes to user-intent, we establish the baseline and implement various techniques to map human intent to specific API calls and their corresponding scopes that GoEx. Here, we describe the methodologies used in this study, each designed to capture a different aspect of the interaction between human intent and system execution capabilities. In our study, we use few-show learning (Section 4.5) and in-context learning with RAG (Section 4.5) as our baselines. We compare this baseline to GoEx's least-privilege mapper, which leverages self-consistency (Section 4.5) and scope constraints (Section 4.5). All the aforementioned policies can be represented in GoEx within 13 lines of code, which highlights the modularity of our system.

The primary input to our system is the user's intent expressed in natural language, such as *"I want to send a message in Slack."* The output from the system is twofold: the specific API call that accomplishes the intent, and the minimal scope necessary to successfully execute the call without exceeding the permissions required.

LLMs considered

To demonstrate the ease of deploying GoEx, for the experiments in this chapter, we consider three popular off the shelf LLM-models. We choose a dated checkpoint for all the three models to ensure the underlying model does not change through the course of our experiments. GPT-4 [1] refers to `gpt-4-turbo-2024-04-09` from OpenAI. Claude [8] refers to `claude-3-opus-20240229` from Anthropic, and Mistral [137] refers to `mistral-large-2402`. Note that Mistral is an open-source model while GPT-4 and Claude are proprietary models. For all our experiments, we set a constant temperature of 0.3.

Least-privilege enforcement

This subsection evaluates the effectiveness of the GoEx framework in enforcing the principle of least privilege, focusing on the ability of LLMs to select the minimal necessary permissions for executing tasks. Our goal is to ensure that the permissions assigned align closely with what is strictly required for the task, minimizing any overreach.

Table 4.1 presents how well GoEx is able to map user intent to the respective least-privilege scope. We measure the effectiveness of these strategies using three key metrics:

1. *Scope excessive*: We also report the unique case, where we assume there is no human-in-the-loop, or the users' naively said YES to all the scopes the LLM requested. Scope excessive then is a measure of the the percentage of instances where the selected scopes exceed what is necessary, reflecting over-privileging.

2. *Scopes not sufficient*: The percentage of instances where the scopes selected by the LLM are inadequate for the task, reflecting under-privileging.

3. *Scope accuracy*: The percentage of instances where the LLM selects the exact scope needed for the task. Scope accuracy measures the LLM's ability to select the correct scope when it is confident (Zero-shot column), and go back to the user to ask for additional clarifications or scope permissions (User conf. column), either because the LLM is not confident, or the user's intent is not clear. This is more a measurement of our prompting technique and the LLM itself. To enhance accuracy, the LLM is also allowed to request additional information from the user when the necessary scope is unclear. This interaction aims to reduce instances where the LLM accesses excessively broad scopes.

These metrics are critical for assessing both user experience and adherence to security principles. It's important to note that while scopes deemed 'not sufficient' adversely affect user experience by failing to perform the intended task, they do not compromise the security principle of least privilege, as they do not allow execution of unintended or excessive permissions. We observe that the GoEx's self-consistency technique achieves highly effective results, with only 0.5% of cases resulting in the assignment of excessive scopes beyond what is necessary for task completion. This technique consistently performs well across various LLM models, and across different hyperparameter settings. This demonstrates the robustness and adaptability of the self-consistency approach in ensuring that the scope of permissions remains tightly aligned with the intended tasks across different computational contexts.

Runtime overhead of GoEx system

In this subsection, we analyze the additional latency introduced by GoEx compared to the baseline scenario, where tasks are executed directly without asking the user for any permissions (credentials are assumed to be given). We aim to quantify the trade-offs between enhanced security through GoEx and the impact on job completion times.

To accurately measure the runtime overhead, we evaluated GoEx across a variety of common user tasks in different applications, including message sending in Slack, managing objects in cloud storage buckets, and controlling virtual machine (VM) instances on a cloud platform. These tasks were chosen to represent a mix of both read (R) and write (W) operations, reflecting typical user interactions in a cloud environment. All of these services were run on a single node with the configuration kept constant throughout the experiment.

Model	Method	Scope excessive (%) ↓	Scope not sufficient (%)	Scope accuracy (%) ↑		
				Zero-shot	User conf.	Total
GPT-4	Few-shot (no RAG)	22.9	25.8	43.4	0.0	43.4
	RAG in context	1.7	5.0	91.1	0.0	91.1
	Self-consistency (3)	0.9	2.4	82.1	13.1	95.1
	Self-consistency (5)	0.7	2.1	79.7	15.9	95.6
	Self-consistency (10)	**0.5**	1.9	75.3	20.4	95.7
	RAG with scopes	**0.3**	6.8	89.7	0.0	89.7
Claude	Few-shot (no RAG)	25.8	22.9	43.4	0.0	43.4
	RAG in context	2.1	9.2	84.8	0.0	84.8
	Self-consistency (3)	1.8	3.6	68.0	22.1	90.1
	Self-consistency (5)	2.7	1.0	62.7	26.1	88.8
	Self-consistency (10)	**2.0**	2.7	49.0	36.9	85.9
	RAG with scopes	**1.6**	5.3	87.6	0.0	87.6
Mistral	Few-shot (no RAG)	26.5	31.3	42.1	0.0	42.1
	RAG in context	1.3	10.9	39.9	0.0	39.9
	Self-consistency (3)	1.3	9.4	40.2	13.3	53.5
	Self-consistency (5)	0.7	5.7	40.4	18.4	58.9
	Self-consistency (10)	**1.1**	4.7	39.1	31.9	70.9
	RAG with scopes	**0.3**	6.2	32.0	0.0	32.0

Table 4.1: **Evaluating robustness of GoEx.** GoEx is robust at mapping user intent to the respective scope. Users are the ultimate source of truth in delegating scopes to GoEx. Scope excessive refers to the scenario where users' naively said *Yes* to all the scopes the LLM requests. Even in such an extreme scenarios, with our self-consistency (10) technique in less than 0.5% of the cases GoEx tends to acquire scopes in excess to what is required to accomplish the user's task. This generalizes across popular models.

We report the job completions times of GoEx across these applications in Table 4.2. We chose APIs that demonstrate a high variance in the job completion times of the underlying applications varying from 0.29 seconds to 269.81 seconds. For all of these services we report the total time taken when the API calls are executed with no LLM in the loop, and the user is assumed to have all the permissions. This is represented in the Job completion column. Baseline latency refers to the common scenario of invoking the LLM to execute the calls as per the few-shot setting defined in Section 4.3. The results summarized in Tab. Table 4.2 show the job completion times under the baseline conditions and with GoEx, as well as the additional overhead introduced by GoEx. We see that GoEx introduces additional latency overheads varying from 0.12% to upto 22.24% for terminating the instance. The time to terminate an instance varies to such a large degree because of the high variance in input tokens resulting from including 777 APIs in the prompt of the LLM.

Application	Job completion (s)			Baseline latency (s)			GoEx latency (s)			Overhead (%)
	Mean	P95	P99	Mean	P95	P99	Mean	P95	P99	Mean
Slack send message (R)	0.29	0.32	0.32s	15.98	17.60	18.23	18.162	29.19	32.34	13.65
List all objects in bucket (R)	18.26	18.48	18.50	33.03	35.17	35.55	37.1	41.37	41.64	12.32
Upload objects to bucket (W)	269.81	291.11	294.08	281.86	286.64	288.134	294.04	297.54	297.55	4.32
Start a VM instance (W)	31.63	65.57	65.58	78.57	87.45	88.63	78.67	90.43	92.5	0.12
Terminate an VM instance (W)	49.02	50.29	50.45	64.28	68.67	69.89	78.58	93.45	97.63	22.24
Get Github Pull Req. details (R)	0.60	0.62	0.63	23.567	29.88	30.24	24.98	37.36	38.57	5.99

Table 4.2: **GoEx latency.** GoEx introduces minimal additional latency varying from $< 1\%$ to $\sim 22\%$ on average. We measure the additional overhead of using GoEx with self-consistency with ten generations over the fastest LLM-baseline of few-shot generation. We measure common tasks like sending Slack messages, listing and uploading objects to GCP cloud object store buckets, starting and terminating a GCP compute instance, and getting pull request details from Github. (R) represents read operations, and (W) represents write operations.

Model	Method	Latency (s)			Prompt Tokens			Completion Tokens		
		Mean	P95	P99	Mean	P95	P99	Mean	P95	P99
GPT-4	Few-Shot (No RAG)	14.9	25.9	34.3	1211.5	1219.0	1225.0	369.3	601.7	714.2
	RAG In Context	15.3	26.9	35.7	2911.7	8976.1	21935.4	370.4	632.1	801.4
	Self-Consistency (3)	18.1	31.3	42.8	2911.7	8976.1	21935.4	1105.8	1840.0	2314.5
	Self-Consistency (5)	19.2	33.6	45.7	2911.7	8976.1	21935.4	1849.6	3047.0	3889.7
	Self-Consistency (10)	23.1	40.7	55.1	2911.7	8976.1	21935.4	3700.5	6071.3	7871.5
	With scopes constraint	14.0	25.1	34.6	2943.5	8620.9	21970.6	337.2	574.3	797.9
Claude	Few-Shot (No RAG)	38.8	51.5	60.9	1407.8	1414.2	1419.0	854.4	1124.5	1315.7
	RAG In Context	29.8	40.8	51.1	4070.3	14882.1	26701.0	718.8	1017.0	1220.4
	Self-Consistency (3)	89.8	121.0	140.2	4070.3	14882.1	26701.0	2145.5	2983.8	3478.1
	Self-Consistency (5)	153.2	205.8	229.0	4070.3	14882.1	26701.0	3604.8	5044.7	5733.5
	Self-Consistency (10)	349.3	477.2	552.6	4070.3	14882.1	26701.0	7545.0	10293.2	11955.5
	With scopes constraint	29.7	42.3	47.6	4112.4	14913.4	26788.2	692.0	1004.4	1201.3
Mistral	Few-Shot (No RAG)	39.7	85.1	94.4	1561.4	1569.0	1576.3	619.4	961.3	1172.2
	RAG In Context	38.8	84.9	98.2	3147.1	10100.2	22856.5	553.6	872.2	1281.5
	Self-Consistency (3)	117.2	185.8	220.3	3147.1	10100.2	22856.5	1662.0	2489.0	3454.8
	Self-Consistency (5)	195.5	304.4	331.3	3147.1	10100.2	22856.5	2697.6	4011.5	5877.5
	Self-Consistency (10)	316.0	473.5	542.0	3147.1	10100.2	22856.5	5478.0	8239.0	9834.8
	With scopes constraint	37.7	83.2	95.4	3190.3	10145.2	22894.3	541.7	860.0	1184.6

Table 4.3: **GoEx overhead.** GoEx's novel adoption of self-consistency introduces minimal additional latency overhead and cost. This is unique to our setting, where the input tokens dominate over output tokens generated thereby keeping costs low. Further, our technique generalizes across models.

Latency overheads for LLM inference.

Building on top of Table 4.2, we study in-depth the latency implications associated with LLM inference within the GoEx framework, specifically focusing on the additional time required for mapping LLM inferences to actionable API calls and scopes based on the different techniques in Table 4.3. To effectively measure latency, it is crucial to understand the concepts of prompt tokens and completion tokens. Prompt tokens refer to the input tokens provided to the LLM, which include the user's intent and any context or constraints necessary for the LLM to understand the task. Completion tokens are the output tokens generated by the LLM, representing the API call and associated scopes necessary to execute the user's intent.

Latency in LLM generation is typically a linear combination of these two factors, as the LLM operates in an autoregressive manner, processing each token sequentially. Consequently, the cost associated with each inference also follows a similar linear relationship with prompt and completion tokens, with API providers charging around $10 for every 10 million prompt tokens and $30 for every 10 million completion tokens. We report tokens in Table 4.3 which is a more stable metric free from market forces.

Few-shot (no RAG) and RAG with scopes show minimal increase in prompt tokens and maintain lower latency, as these methods are streamlined to focus the LLM's generation process. Self-consistency approaches, where multiple outputs (3, 5, or 10) are generated for a single prompt, show a significant increase in completion tokens but no increase the number of input prompt tokens. This approach maintains the original prompt while seeking consistency in the generated outputs, which increase the cost proportionally, but *not* latency nominally. The introduction of scope constraints slightly increases the number of prompt tokens since additional information about permissible scopes must be included in the prompt. However, this method helps focus the LLM's generation process, potentially reducing the range of completion tokens and thereby controlling latency.

Qualitative examples

In this section we highlight scenarios where the LLMs, if left unattended to, can end up with more permissions than needed. These are examples from GPT-4 with self-consistency of $k = 10$, as reported in Table 4.1. For the prompt, *"Can you find nearby Italian restaurants around my current location?"*, the LLM picked the right set of APIs of `searchNearby` from `places.googleapis.com`. However, while the scope required to perform this API call is `maps-platform.places.nearbysearch`, the LLM requested the parent scope of `maps-platform.places`.

Similarly for a user's request of, *"How do I add a new video to a playlist on YouTube?"*, the LLM requested the unnecessary scope of `youtubepartner` while `youtube.upload` would have been sufficient. By using the LLM to narrow down the scope, and relying on the users as the ultimate source of truth, GoEx as a run-time is able to confine the LLM's actions.

4.9 Related Work

Existing literature can be broadly categorized into techniques to adavance the LLM performance to make them better reasoners, and instruction followers, and works towards isolating, and improving the trustworthiness of LLMs. We address each of these below:

Advancements in LLM

Although earlier LLMs primarily excelled at text generation and understanding, more recent advances have expanded their functionality into areas like tool usage [107, 121, 174], multimodal reasoning [53, 134, 156], and open-ended problem solving [149, 163]. This has led to LLMs being explored as problem solvers that can break down complex queries into relevant sub-tasks, leverage relevant tools, and synthesize the results.

Often called agents, or co-pilots, recent work has explored the use of LLMs in a variety of applications, including customer service [14, 99], data analysis [26, 27], software development [72, 119, 155, 171], planning robotics tasks [54], and simulating human behavior [104, 105]. These applications have demonstrated the potential of LLMs to automate complex tasks and improve efficiency in a variety of domains.

Isolation

Prior work on enabling automated LLM-powered systems [157] draws on concepts from existing computer systems [32, 83, 116, 152] and emphasizes isolation between LLM-powered applications in order to secure the overall system. Traditional methods such as using containers—which guarantee isolation through virtualization, falter when adjustments to the environment are required within the user's context. For example, a user might want to modify the state of their operating system. Further, isolation alone cannot ensure the final state aligns with the intended actions, especially under ambiguity or execution errors and ambiguity of stated intent. Both are common flaws when intent is being specified in natural-language as opposed to a more precise domain-specific language.

Trustworthiness in LLMs

There is a rich body of work benchmarking LLMs on their robustness [22]. Recently, trustworthiness [143] has been introduced as a multifaceted benchmark encapsulating robustness, stereotype bias, toxicity, privacy, ethics, and fairness. Wang et al. [143] found that more advanced LLMs (e.g., GPT-4) exhibit higher, albeit still imperfect, trustworthiness.

Attacks on LLMs and defenses

LLMs are also susceptible to prompt hacking attacks, of which include prompt injection [46, 88, 108, 123, 166] and jailbreaking [5, 62]. Such attacks can lead to unpredictable and malicious

decisions made by the LLM. There is also an active line of work on defending against such attacks on LLMs [24, 109, 132, 140, 165]. These attacks and defenses will continue to evolve, and consequently the potential of LLMs being susceptible to having their trustworthiness undermined necessitates a runtime that can provide execution of LLM-decided actions while limiting risk.

4.10 Conclusion

The evolution of LLMs from chatbots to deeply embedding them in applications and services for autonomous operation among themselves and other agents presents an exciting future. However, the potential for LLMs to make decisions that are not aligned with the user's intent, especially in the presence of complex APIs and scopes, necessitates a runtime that can provide execution of LLM-decided actions while limiting risk. We presented GoEx, a runtime that separates user credentials from the LLM, and enables LLMs to execute actions on behalf of the user while confining the LLM inside the scope of permissions the user has granted. In order to map API calls to their respective scopes, we leveraged self-consistency in order to improve robustness and a-priori knowledge of scopes the user has pre-approved. By relying on users as the ultimate source of truth, GoEx presents a secure system to power autonomous agents.

Chapter 5

Conclusion

The advancements brought forward by Gorilla, OpenFunctions, RAFT, and GoEx collectively represent a step forward in the practical deployment of LLMs for interactive and autonomous tasks. Gorilla, by enhancing the ability of LLMs to understand and generate API calls, addresses a crucial gap in their capability to interact with external tools and services. This transitions LLMs from merely impressive knowledge repositories to dynamic decision-making agents capable of performing complex tasks autonomously.

Furthermore, the integration of RAG through OpenFunctions and the enhancements provided by RAFT makes function-calling easy to use without the need for custom fine-tuning or constant retraining. These systems refine the ability of LLMs to access and use real-time and relevant "fresh" information and also ensures that the generated outputs are increasingly accurate and grounded in actual data. RAFT enhances the enables LLMs to discriminate between relevant and distracting information, a critical skill in many applications involving real-world retrievers.

Lastly, GoEx is a runtime system that safely executes LLM-generated API calls in user-centric applications. By incorporating tried-and-true systems security principles that prevent misuse through least-privilege access-control, as well as presenting undo and damage confinement abstractions to address the post-facto validation setting, GoEx facilitates deployment of LLMs as decision making agents.

5.1 Lessons Learnt

Sidecar LLMs

The evolution of Large Language Models (LLMs) into specialized roles such as tool-calling, long-context, reasoning, and planning necessitates an orchestration strategy. One effective approach has been the deployment of sidecar LLMs. In this configuration, each LLM autonomously evaluates a user's prompt to determine if it should intervene by injecting additional context before the prompt is processed by a central LLM. For example, consider

two LLMs: a specialized tool-use Gorilla LLM and a general-purpose Gen-LLM. When faced with a prompt like "give me a joke about the current weather in Berkeley," the Gorilla LLM could proactively fetch the weather data and supply it to the Gen-LLM, thereby reducing the number of exchanges required. This *push* mechanism as opposed to the Gen-LLM invoking other specialized LLMs *pull* mechanism was popular among users of Gorilla.

Fine-tuning and RAG

The choice of fine-tuning or RAG systems has been a point of debate. Our findings indicate that while fine-tuning is highly efficient in embedding knowledge into LLMs, its application cannot be a daily routine due to both high costs and the complexities involved in LLM evaluation. As LLMs undergo fine-tuning only intermittently, retrievers become essential for maintaining the freshness of the data. However, retrievers alone are hindered by low recall rates, even within their domains. Although difficult for many users to accept, we find the synergy of fine-tuning and RAG to be necessary for achieving robust performance.

5.2 Future Work

Gorilla, OpenFunctions, RAFT, and GoEx present a first step in designing a system for executing LLM-powered actions such as APIs. Here, we describe some open questions in the design of such a system.

Beyond Human and AI Feedback to Perpetual Feedback Loops

Current reinforcement learning techniques, such as Reinforcement Learning through Human Feedback [102, 113] and AI Feedback [71], do not account for the temporal evolution of user preferences. An intriguing direction for future research would be to develop methodologies that can adaptively discern, reflect and incorporate changes in preferences over time.

Designing LLM-friendly APIs

The conversation around LLM-powered systems design is predominantly centered around designing systems to conform with the API semantics of existing applications and services. However, an equally interesting question is what API design in an LLM-centric world would looks like. LLMs introduce a paradigm where applications and services can anticipate and adapt to the intricacies of LLM interactions. A notable feature that embodies this adaptability is the implementation of "dry-run" semantics, akin to the functionality commonly visible in infrastructure products such as [11], [67], and where API calls can be tested to predict their success without executing any real changes. This concept can be extended beyond mere prediction, serving as a bridge between LLMs' proposed actions and user consent. By repurposing "dry-run" operations, service providers can offer a preview of the uncommitted

state resulting from an LLM's actions, allowing users to evaluate and approve these actions before they are finalized. This process adds an essential layer of user oversight, ensuring that actions align with user expectations and intentions.

Chaining-aware. Applications and services should by default expect their APIs to be chained when used by agents. To support such a scenario, there needs to be a way to express which APIs can be commutative, associative or distributive with a given set of APIs.

Tracking LLM agents

The introduction of a nonce mechanism (i.e., a session identifier) would enable LLMs to present their identity and facilitate smoother interactions with API providers. This could serve various purposes, such as identifying a session initiated by an LLM or providing a context for transactions. A transaction ID, for example, can enable a system to identify and potentially rollback actions based on this ID. This not only improves the traceability of interactions but also contributes to the overall robustness and reliability of the system by providing an auditable framework for LLM-powered systems.

Cross-platform evaluation

Our GoEx evaluation framework measures the accuracy of API call generation within the realm of web APIs, but our evaluation does not extend to other types of APIs, such as those used in SQL databases or Python libraries. While the GoEx framework presents abstractions for executing *any* API, evaluation of other APIs is notably absent. Conducting cross-platform evaluations to understand how GoEx performs under varied operational conditions and with different types of APIs will help identify platform-specific challenges and opportunities for improving the system's accuracy and reliability. Further, GoEx only evaluated those cases where the user's request can be addressed with a few API calls due to manual inspection needed to build the evaluation dataset. LLM-agents of the future might interact with tens, if not hundreds of APIs and their corresponding scopes in a single session.

Translating scopes for users

GoEx relies on developer provided description to prompt the user about certain scopes when requesting their permission to delegate access. However, it remains unclear to what extent a non-expert audience can understand the description. For example, the scope `team:read` has the description, *"View the name, email domain, and icon for workspaces your slack app is connected to,"* which is arguably well-described. However, the description of *"Read new Platform triggers"* for the scope `triggers:read` is slightly more opaque. A system that is responsible for negotiating scopes with an user also needs to be able to present the user's with the right vocabulary to help them make the right decisions.

Bibliography

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *VLDB*, 2015.

[4] Daniel Andor, Luheng He, Kenton Lee, and Emily Pitler. Giving bert a calculator: Finding operations and arguments with reading comprehension. *arXiv preprint arXiv:1909.00109*, 2019.

[5] Cem Anil, Esin Durmus, Mrinank Sharma, Joe Benton, Sandipan Kundu, Joshua Batson, Nina Rimsky, Meg Tong, Jesse Mu, Daniel Ford, et al. Many-shot jailbreaking. 2024.

[6] Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.

[7] Anthropic. Prompt engineering for claude's long context window. 2023.

[8] https://www.anthropic.com/index/introducing-claude Anthropic. Claude, 2022.

[9] Apple. Control access to information in apps on iPhone. `https://support.apple.com/guide/iphone/control-access-to-information-in-apps-iph251e92810/ios`, 2024.

[10] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-rag: Learning to retrieve, generate, and critique through self-reflection. *arXiv preprint arXiv:2310.11511*, 2023.

[11] AWS. Testing your AWS KMS API calls. `https://docs.aws.amazon.com/kms/latest/developerguide/programming-dryrun.html`.

[12] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. Autopandas: neural-backed generators for program synthesis. *OOPSLA*, 2019.

[13] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*, pages 2206–2240. PMLR, 2022.

[14] Petter Bae Brandtzaeg and Asbjørn Følstad. Why people use chatbots. In *INSCI*. Springer, 2017.

[15] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *NeurIPS*, 2020.

[16] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

[17] Collin Burns, Haotian Ye, Dan Klein, and Jacob Steinhardt. Discovering latent knowledge in language models without supervision. *arXiv preprint arXiv:2212.03827*, 2022.

[18] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering*, 2015.

[19] Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. Quantifying memorization across neural language models. In *The Eleventh International Conference on Learning Representations*, 2022.

[20] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. The secret sharer: Evaluating and testing unintended memorization in neural networks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 267–284, 2019.

[21] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al.

Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2633–2650, 2021.

[22] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. A survey on evaluation of large language models. *ACM TIST*, 2023.

[23] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[24] Sizhe Chen, Julien Piet, Chawin Sitawarin, and David Wagner. StruQ: Defending against prompt injection with structured queries. *arXiv preprint arXiv:2402.06363*, 2024.

[25] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

[26] Liying Cheng, Xingxuan Li, and Lidong Bing. Is GPT-4 a good data analyst? *arXiv preprint arXiv:2305.15038*, 2023.

[27] Robert Chew, John Bollenbacher, Michael Wenger, Jessica Speer, and Annice Kim. Llm-assisted content analysis: Using large language models to support deductive coding. *arXiv preprint arXiv:2306.14924*, 2023.

[28] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023.

[29] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[30] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*, 2022.

[31] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[32] Ellis Cohen and David Jefferson. Protection in the Hydra operating system. In *SOSP*, 1975.

[33] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed database. In *OSDI*, 2012.

[34] Franck Dernoncourt and Ji Young Lee. Pubmed 200k rct: a dataset for sequential sentence classification in medical abstracts. *arXiv preprint arXiv:1710.06071*, 2017.

[35] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR, 2017.

[36] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*, 2022.

[37] Vitaly Feldman. Does learning require memorization? a short tale about a long tail. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 954–959, 2020.

[38] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *ACM CCS*, 2011.

[39] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdatta Akhawe, and David Wagner. How to ask for permission. In *HotSec*, 2012.

[40] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *USENIX WebApps*, 2011.

[41] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *USENIX SOUPS*, 2012.

[42] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. *arXiv preprint arXiv:2211.10435*, 2022.

[43] Google. Google Workspace APIs. `https://developers.google.com/workspace/explore`.

[44] Google. OAuth 2.0 scopes for Google APIs. `https://developers.google.com/identity/protocols/oauth2/scopes`.

[45] Google. Public certificate authority CA, 2024. `https://cloud.google.com/certificate-manager/docs/public-ca`.

[46] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In *ACM AISec*, 2023.

[47] Andrey Gubarev, Dan Delorey, Geoffrey Michael Romer, Hossein Ahmadi, Jeff Shute, Jing Jing Long, Matt Tolton, Mosha Pasumansky, Narayanan Shivakumar, Sergey Melnik, Slava Min, and Theo Vassilakis. Dremel: A decade of interactive sql analysis at web scale. pages 3461–3472, 2020.

[48] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. Retrieval augmented language model pre-training. In *International conference on machine learning*, pages 3929–3938. PMLR, 2020.

[49] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM CSUR*, 1983.

[50] Dick Hardt. The OAuth 2.0 authorization framework. `https://datatracker.ietf.org/doc/html/rfc6749`.

[51] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019.

[52] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[53] Rongjie Huang, Mingze Li, Dongchao Yang, Jiatong Shi, Xuankai Chang, Zhenhui Ye, Yuning Wu, Zhiqing Hong, Jiawei Huang, Jinglin Liu, et al. Audiogpt: Understanding and generating speech, music, sound, and talking head. In *AAAI*, 2024.

[54] Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*, 2022.

[55] Umar Iqbal, Tadayoshi Kohno, and Franziska Roesner. LLM platform security: Applying a systematic evaluation framework to OpenAI's ChatGPT plugins. *arXiv preprint arXiv:2309.10254*, 2023.

[56] Srinivasan Iyer, Xi Victoria Lin, Ramakanth Pasunuru, Todor Mihaylov, Dániel Simig, Ping Yu, Kurt Shuster, Tianlu Wang, Qing Liu, Punit Singh Koura, et al. Opt-iml: Scaling language model instruction meta learning through the lens of generalization. *arXiv preprint arXiv:2212.12017*, 2022.

[57] Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. Atlas: Few-shot learning with retrieval augmented language models. *Journal of Machine Learning Research*, 24(251):1–43, 2023.

[58] Charlie Cheng-Jie Ji, Huanzhi Mao, Fanjia Yan, Tianjun Zhang Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Gorilla openfunctions v2. 2024.

[59] Qiao Jin, Bhuwan Dhingra, Zhengping Liu, William W Cohen, and Xinghua Lu. Pubmedqa: A dataset for biomedical research question answering. *arXiv preprint arXiv:1909.06146*, 2019.

[60] Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. *arXiv preprint arXiv:1705.03551*, 2017.

[61] Nikhil Kandpal, Eric Wallace, and Colin Raffel. Deduplicating training data mitigates privacy risks in language models. In *International Conference on Machine Learning*, pages 10697–10707. PMLR, 2022.

[62] Daniel Kang, Xuechen Li, Ion Stoica, Carlos Guestrin, Matei Zaharia, and Tatsunori Hashimoto. Exploiting programmatic behavior of LLMs: Dual-use through standard security attacks. *arXiv preprint arXiv:2302.05733*, 2023.

[63] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through memorization: Nearest neighbor language models. *arXiv preprint arXiv:1911.00172*, 2019.

[64] Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. *arXiv preprint arXiv:2303.17491*, 2023.

[65] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *arXiv preprint arXiv:2205.11916*, 2022.

[66] Mojtaba Komeili, Kurt Shuster, and Jason Weston. Internet-augmented dialogue generation. *arXiv preprint arXiv:2107.07566*, 2021.

[67] Kubernetes. kubectl usage conventions. `https://kubernetes.io/docs/reference/kubectl/conventions/`.

[68] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:453–466, 2019.

[69] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*, 2020.

[70] Angeliki Lazaridou, Elena Gribovskaya, Wojciech Stokowiec, and Nikolai Grigorev. Internet-augmented language models through few-shot prompting for open-domain question answering. *arXiv preprint arXiv:2203.05115*, 2022.

[71] Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Lu, Thomas Mesnard, Colton Bishop, Victor Carbune, and Abhinav Rastogi. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. *arXiv preprint arXiv:2309.00267*, 2023.

[72] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *IEEE/ACM ICSE*, 2023.

[73] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.

[74] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[75] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *NeurIPS*, 2020.

[76] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[77] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*, 2021.

[78] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[79] Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, et al. Taskmatrix. ai: Completing tasks by connecting foundation models with millions of apis. *arXiv preprint arXiv:2303.16434*, 2023.

[80] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

[81] Xi Victoria Lin, Xilun Chen, Mingda Chen, Weijia Shi, Maria Lomeli, Rich James, Pedro Rodriguez, Jacob Kahn, Gergely Szilvasy, Mike Lewis, et al. Ra-dit: Retrieval-augmented dual instruction tuning. *arXiv preprint arXiv:2310.01352*, 2023.

[82] Xi Victoria Lin, Xilun Chen, Mingda Chen, Weijia Shi, Maria Lomeli, Rich James, Pedro Rodriguez, Jacob Kahn, Gergely Szilvasy, Mike Lewis, et al. Ra-dit: Retrieval-augmented dual instruction tuning. *arXiv preprint arXiv:2310.01352*, 2023.

[83] Theodore A Linden. Operating system structures to support security and reliable software. *CSUR*, 1976.

[84] Steven B Lipner. A comment on the confinement problem. In *ACM SIGOPS Operating Systems Review*, 1975.

[85] Hao Liu, Carmelo Sferrazza, and Pieter Abbeel. Chain of hindsight aligns language models with feedback. *arXiv preprint arXiv:2302.02676*, 3, 2023.

[86] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*, 2023.

[87] Xiao Liu, Kaixuan Ji, Yicheng Fu, Weng Tam, Zhengxiao Du, Zhilin Yang, and Jie Tang. P-tuning: Prompt tuning can be comparable to fine-tuning across scales and tasks. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 61–68, 2022.

[88] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, and Yang Liu. Prompt injection attack against LLM-integrated applications. *arXiv preprint arXiv:2306.05499*, 2023.

[89] Zihan Liu, Wei Ping, Rajarshi Roy, Peng Xu, Mohammad Shoeybi, and Bryan Catanzaro. Chatqa: Building gpt-4 level conversational qa models. *arXiv preprint arXiv:2401.10225*, 2024.

[90] Ziming Liu, Ouail Kitouni, Niklas S Nolte, Eric Michaud, Max Tegmark, and Mike Williams. Towards understanding grokking: An effective theory of representation learning. *Advances in Neural Information Processing Systems*, 35:34651–34663, 2022.

[91] David Mazieres, Michael Kaminsky, M Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *SOSP*, 1999.

[92] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *ICML*, 2013.

[93] Microsoft. Azure REST API reference. `https://learn.microsoft.com/en-us/rest/api/azure/`.

[94] Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. Cross-task generalization via natural language crowdsourcing instructions. *arXiv preprint arXiv:2104.08773*, 2021.

[95] Niklas Muennighoff, Thomas Wang, Lintang Sutawika, Adam Roberts, Stella Biderman, Teven Le Scao, M Saiful Bari, Sheng Shen, Zheng Xin Yong, Hailey Schoelkopf, Xiangru Tang, Dragomir Radev, Alham Fikri Aji, Khalid Almubarak, Samuel Albanie, Zaid Alyafeai, Albert Webson, Edward Raff, and Colin Raffel. Crosslingual generalization through multitask finetuning. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15991–16111, Toronto, Canada, July 2023. Association for Computational Linguistics.

[96] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

[97] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*, 2023.

[98] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[99] City of New York. MyCity business services chatbot. `https://chat.nyc.gov/`.

[100] OpenAI. ChatGPT, 2022. `https://openai.com/blog/chatgpt`.

[101] OpenAI. ChatGPT plugins, 2023. `https://openai.com/blog/chatgpt-plugins`.

[102] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

[103] Xudong Pan, Mi Zhang, Shouling Ji, and Min Yang. Privacy risks of general-purpose language models. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1314–1331. IEEE, 2020.

[104] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *UIST*, 2023.

[105] Joon Sung Park, Lindsay Popowski, Carrie Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Social simulacra: Creating populated prototypes for social computing systems. In *UIST*, 2022.

[106] Shishir G Patil, Tianjun Zhang, Vivian Fang, Roy Huang, Aaron Hao, Martin Casado, Joseph E Gonzalez, Raluca Ada Popa, Ion Stoica, et al. Goex: Perspectives and designs towards a runtime for autonomous llm applications. *arXiv preprint arXiv:2404.06921*, 2024.

[107] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. *arXiv preprint arXiv:2305.15334*, 2023.

[108] Fábio Perez and Ian Ribeiro. Ignore previous prompt: Attack techniques for language models. *arXiv preprint arXiv:2211.09527*, 2022.

[109] Julien Piet, Maha Alrashed, Chawin Sitawarin, Sizhe Chen, Zeming Wei, Elizabeth Sun, Basel Alomair, and David Wagner. Jatmo: Prompt injection defense by task-specific finetuning. *arXiv preprint arXiv:2312.17673*, 2023.

[110] Alethea Power, Yuri Burda, Harri Edwards, Igor Babuschkin, and Vedant Misra. Grokking: Generalization beyond overfitting on small algorithmic datasets. *arXiv preprint arXiv:2201.02177*, 2022.

[111] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs, 2023.

[112] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.

[113] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18290*, 2023.

[114] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. In-context retrieval-augmented language models. *arXiv preprint arXiv:2302.00083*, 2023.

[115] Vipula Rawte, Amit Sheth, and Amitava Das. A survey of hallucination in large foundation models. *arXiv preprint arXiv:2309.05922*, 2023.

[116] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: Process separation for web sites within the browser. In *USENIX Security*, 2019.

[117] Rabbit Research. Learning human actions on computer applications, 2023.

[118] Victor Sanh, Albert Webson, Colin Raffel, Stephen H Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, et al. Multitask prompted training enables zero-shot task generalization. *arXiv preprint arXiv:2110.08207*, 2021.

[119] Divyanshu Saxena, Nihal Sharma, Donghyun Kim, Rohit Dwivedula, Jiayi Chen, Chenxi Yang, Sriram Ravula, Zichao Hu, Aditya Akella, Sebastian Angel, Joydeep Biswas, Swarat Chaudhuri, Isil Dillig, Alex Dimakis, P. Brighten Godfrey, Daehyeok Kim, Chris Rossbach, and Gang Wang. On a foundation model for operating systems, 2023.

[120] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.

[121] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

[122] Timo Schick and Hinrich Schütze. Exploiting cloze questions for few shot text classification and natural language inference. *arXiv preprint arXiv:2001.07676*, 2020.

[123] Sander Schulhoff, Jeremy Pinto, Anaum Khan, Louis-François Bouchard, Chenglei Si, Svetlina Anati, Valen Tagliabue, Anson Liu Kost, Christopher Carnahan, and Jordan Boyd-Graber. Ignore this title and hackaprompt: Exposing systemic vulnerabilities of LLMs through a global scale prompt hacking competition. *arXiv preprint arXiv:2311.16119*, 2023.

[124] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. *arXiv preprint arXiv:2303.17580*, 2023.

[125] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*, pages 31210–31227. PMLR, 2023.

[126] Weijia Shi, Anirudh Ajith, Mengzhou Xia, Yangsibo Huang, Daogao Liu, Terra Blevins, Danqi Chen, and Luke Zettlemoyer. Detecting pretraining data from large language models. *arXiv preprint arXiv:2310.16789*, 2023.

[127] Weijia Shi, Sewon Min, Maria Lomeli, Chunting Zhou, Margaret Li, Victoria Lin, Noah A Smith, Luke Zettlemoyer, Scott Yih, and Mike Lewis. In-context pretraining: Language modeling beyond document boundaries. *arXiv preprint arXiv:2310.10638*, 2023.

[128] Weijia Shi, Sewon Min, Michihiro Yasunaga, Minjoon Seo, Rich James, Mike Lewis, Luke Zettlemoyer, and Wen-tau Yih. Replug: Retrieval-augmented black-box language models. *arXiv preprint arXiv:2301.12652*, 2023.

[129] Noah Shinn, Beck Labash, and Ashwin Gopinath. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366*, 2023.

[130] Kurt Shuster, Jing Xu, Mojtaba Komeili, Da Ju, Eric Michael Smith, Stephen Roller, Megan Ung, Moya Chen, Kushal Arora, Joshua Lane, et al. Blenderbot 3: a deployed conversational agent that continually learns to responsibly engage. *arXiv preprint arXiv:2208.03188*, 2022.

[131] Sanjay Subramanian, Medhini Narasimhan, Kushal Khangaonkar, Kevin Yang, Arsha Nagrani, Cordelia Schmid, Andy Zeng, Trevor Darrell, and Dan Klein. Modular visual question answering via code generation. *arXiv preprint arXiv:2306.05392*, 2023.

[132] Xuchen Suo. Signed-Prompt: A new approach to prevent prompt injection attacks against LLM-integrated applications. *arXiv preprint arXiv:2401.07612*, 2024.

[133] Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. *arXiv preprint arXiv:2303.08128*, 2023.

[134] Zineng Tang, Ziyi Yang, Chenguang Zhu, Michael Zeng, and Mohit Bansal. Any-to-any generation via composable diffusion. *NeurIPS*, 2024.

[135] Michael Tänzer, Sebastian Ruder, and Marek Rei. Memorisation versus generalisation in pre-trained language models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7564–7578, 2022.

[136] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. `https://github.com/tatsu-lab/stanford_alpaca`, 2023.

[137] Mistral AI team. Au large. `https://mistral.ai/news/mistral-large/`.

[138] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*, 2022.

[139] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[140] Sam Toyer, Olivia Watkins, Ethan Adrian Mendes, Justin Svegliato, Luke Bailey, Tiffany Wang, Isaac Ong, Karim Elmaaroufi, Pieter Abbeel, Trevor Darrell, et al. Tensor trust: Interpretable prompt injection attacks from an online game. *arXiv preprint arXiv:2311.01011*, 2023.

[141] Sai Vemprala, Rogerio Bonatti, Arthur Bucker, and Ashish Kapoor. Chatgpt for robotics: Design principles and model abilities. *2023*, 2023.

[142] Tu Vu, Mohit Iyyer, Xuezhi Wang, Noah Constant, Jerry Wei, Jason Wei, Chris Tar, Yun-Hsuan Sung, Denny Zhou, Quoc Le, et al. Freshllms: Refreshing large language models with search engine augmentation. *arXiv preprint arXiv:2310.03214*, 2023.

[143] Boxin Wang, Weixin Chen, Hengzhi Pei, Chulin Xie, Mintong Kang, Chenhui Zhang, Chejian Xu, Zidi Xiong, Ritik Dutta, Rylan Schaeffer, et al. DecodingTrust: A comprehensive assessment of trustworthiness in GPT models. *arXiv preprint arXiv:2306.11698*, 2023.

[144] Boxin Wang, Wei Ping, Lawrence McAfee, Peng Xu, Bo Li, Mohammad Shoeybi, and Bryan Catanzaro. Instructretro: Instruction tuning post retrieval-augmented pretraining. *arXiv preprint arXiv:2310.07713*, 2023.

[145] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *ICLR*, 2023.

[146] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560*, 2022.

[147] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In *ACL*, 2023.

[148] Yizhong Wang, Swaroop Mishra, Pegah Alipoormolabashi, Yeganeh Kordi, Amirreza Mirzaei, Atharva Naik, Arjun Ashok, Arut Selvan Dhanasekaran, Anjana Arunkumar, David Stap, et al. Super-naturalinstructions: Generalization via declarative instructions

on 1600+ nlp tasks. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5085–5109, 2022.

[149] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

[150] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Permission evolution in the Android ecosystem. In *ACSAC*, 2012.

[151] Jason Weston and Sainbayar Sukhbaatar. System 2 attention (is something you might need too). *arXiv preprint arXiv:2311.11829*, 2023.

[152] Maurice Vincent Wilkes and Roger Michael Needham. The Cambridge CAP computer and its operating system. 1979.

[153] Yotam Wolf, Noam Wies, Yoav Levine, and Amnon Shashua. Fundamental limitations of alignment in large language models. *arXiv preprint arXiv:2304.11082*, 2023.

[154] BigScience Workshop, Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.

[155] Scott Wu. Introducing Devin, the first AI software engineer. `https://www.cognition-labs.com/introducing-devin`.

[156] Shengqiong Wu, Hao Fei, Leigang Qu, Wei Ji, and Tat-Seng Chua. Next-GPT: Any-to-any multimodal LLM. *arXiv preprint arXiv:2309.05519*, 2023.

[157] Yuhao Wu, Franziska Roesner, Tadayoshi Kohno, Ning Zhang, and Umar Iqbal. SecGPT: An execution isolation architecture for LLM-based systems. *arXiv preprint arXiv:2403.04960*, 2024.

[158] Wenhan Xiong, Jingyu Liu, Igor Molybog, Hejia Zhang, Prajjwal Bhargava, Rui Hou, Louis Martin, Rashi Rungta, Karthik Abinav Sankararaman, Barlas Oguz, et al. Effective long-context scaling of foundation models. *arXiv preprint arXiv:2309.16039*, 2023.

[159] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10, 2022.

[160] Peng Xu, Wei Ping, Xianchao Wu, Lawrence McAfee, Chen Zhu, Zihan Liu, Sandeep Subramanian, Evelina Bakhturina, Mohammad Shoeybi, and Bryan Catanzaro. Retrieval meets long context large language models. *arXiv preprint arXiv:2310.03025*, 2023.

[161] Fanjia Yan, Huanzhi Mao, Charlie Cheng-Jie Ji, Tianjun Zhang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Berkeley function calling leaderboard. `https://gorilla.cs.berkeley.edu/blogs/8_berkeley_function_calling_leaderboard.html`, 2024.

[162] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W Cohen, Ruslan Salakhutdinov, and Christopher D Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600*, 2018.

[163] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

[164] Xi Ye and Greg Durrett. The unreliability of explanations in few-shot prompting for textual reasoning. In *NeurIPS*, 2022.

[165] Jingwei Yi, Yueqi Xie, Bin Zhu, Keegan Hines, Emre Kiciman, Guangzhong Sun, Xing Xie, and Fangzhao Wu. Benchmarking and defending against indirect prompt injection attacks on large language models. *arXiv preprint arXiv:2312.14197*, 2023.

[166] Jiahao Yu, Yuhang Wu, Dong Shu, Mingyu Jin, and Xinyu Xing. Assessing prompt injection risks in 200+ custom GPTs. *arXiv preprint arXiv:2311.11538*, 2023.

[167] Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, et al. Glm-130b: An open bilingual pre-trained model. *arXiv preprint arXiv:2210.02414*, 2022.

[168] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[169] Tianjun Zhang, Fangchen Liu, Justin Wong, Pieter Abbeel, and Joseph E Gonzalez. The wisdom of hindsight makes language models better instruction followers. *arXiv preprint arXiv:2302.05206*, 2023.

[170] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. Siren's song in the AI ocean: a survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219*, 2023.

[171] Yusheng Zheng, Yiwei Yang, Maolin Chen, and Andrew Quinn. KEN: Kernel extensions using natural language. *arXiv preprint arXiv:2312.05531*, 2023.

[172] Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, et al. Lima: Less is more for alignment. *arXiv preprint arXiv:2305.11206*, 2023.

[173] Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao Jiang, and Graham Neubig. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*, 2022.

[174] Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. ToolQA: A dataset for LLM question answering with external tools. *NeurIPS*, 2024.