A Taxonomy of Failures in Tool-Augmented LLMs

Cailin Winston
University of Washington
Seattle, USA
cailinw@cs.washington.edu

René Just
University of Washington
Seattle, USA
rjust@cs.washington.edu

***Abstract*—Large language models (LLMs) can perform a variety of tasks given a user prompt that contains a description of the task. To enhance the performance of LLMs, recent research has focused on augmenting LLMs with external tools, such as Python APIs, REST APIs, and other deep learning models. Much of the research on tool-augmented LLMs (TaLLMs) has focused on improving their capabilities and accuracy. However, research on understanding and characterizing the kinds of failures that can occur in these systems is lacking. To address this gap, this paper proposes a taxonomy of failures in TaLLMs and their root causes, details an analysis of the failures that occur in two published TaLLMs (Gorilla and Chameleon), and provides recommendations for testing and repair of TaLLMs.**

***Index Terms*—Large language models, tool-augmented LLMs, software testing, fault localization, repair.**

I. Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities across a wide range of domains, performing tasks simply through text inputs that contain instructions and examples for the task. However, their effectiveness becomes limited in task domains that require nuanced reasoning, such as mathematics, or in more complex tasks that require interacting with external databases, APIs, or other software systems. To address these limitations, tool-augmented LLMs (TaLLMs) extend the capabilities of LLMs by enabling them to use external tools, such as search engines, other deep learning models, modules for controlling physical agents, and domain-specific functionality implemented in traditional software libraries. By enabling an LLM to utilize tools to acquire the most relevant and up-to-date information as well as interact with real-world systems, TaLLMs can perform a wide range of tasks, such as question answering [1], embodied agent task planning [2], automatic program repair [3], and even customer-service chatbots [4].

The increased investment in research and deployment of TaLLMs in real-world settings underscores the need for robust methods for testing and repair of TaLLMs. Although prior work has developed benchmark datasets [5], [6] and failure detection methods [7], [8], systematic approaches to testing and repairing these systems are needed for a safe and wide-spread deployment of TaLLMs. To achieve this goal, it is important to develop a comprehensive understanding of TaLLM failure modes, which go beyond failures that can be localized to LLMs. Repair of TaLLMs requires more than simply re-training an LLM, and thus mapping failures to root causes in these compositional systems is a crucial first step.

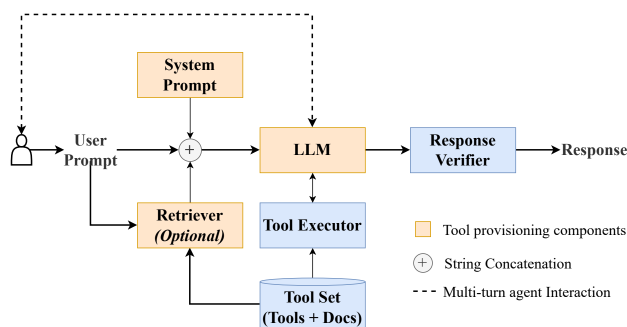

Fig. 1: Common components of TaLLMs.

This paper focuses on developing a systematic understanding of failures in TaLLMs. It enumerates possible failures and their root causes, based on a literature survey and an in-depth analysis of two published TaLLMs. Furthermore, it emphasizes the importance of end-to-end system testing and provides recommendations for leveraging fault localization and root cause analysis to evaluate, test, and repair TaLLMs.

Specifically, this paper makes the following contributions:

- A systems and software engineering perspective on TaLLMs in literature (Section II).
- A systematic analysis of failures and root causes observed in two published TaLLMs (Section III).
- Recommendations for TaLLM evaluation, testing, and repair (Section IV).

This paper aims to lay the foundation for enabling a more structured approach to building robust TaLLMs, based on insights into failure types and root causes. We hope that our work inspires future research on the development of systematic, comprehensive, and effective approaches to testing, debugging, and repairing TaLLMs.

II. A Systems and Software Engineering Perspective on TaLLMs

TaLLMs (Figure 1) are modular software systems composed of multiple components with well-defined interfaces, the central component being the LLM. This section provides a view of TaLLMs from a systems design and software engineering perspective that aims at informing systematic methods for TaLLM testing and repair.

TABLE I: Summary of TALLMs included in our literature survey.

TALLM	Tools				Agents	
	Examples	Tool provisioning	Timing of Invocation	# of Invocations	# of Agents	# of Turns
Chameleon [1]	LLMs, vision models, web search, Python functions, and heuristics-based modules	IC	PI	> 1	1	1
RepairAgent [3]	14 program repair-specific tools	IC	PI	> 1	1	> 1
React [9]	3 Wikipedia APIs	IC	PI	> 1	1	> 1
API-Bank [5]	Over 2000 APIs from github.com/public-apis	IC / FT	PI	1	1	> 1
HuggingGPT [10]	HuggingFace models	IC	PI	> 1	1	> 1
Gorilla [11]	ML APIs	FT / RAG	PI	1	1	1
ToolAlpaca [12]	400 real-world tool APIs	FT	PI	> 1	1	> 1
α**-UMI** [13]	Various APIs	FT	PI	> 1	> 1	> 1
ART [14]	Google search, codex, Python interpretor	IC	ILD	> 1	1	> 1
ToolkenGPT [2]	Math tools, database APIs, robot APIs	FT	ILD	> 1	1	1
TALM [15]	Text-to-text APIs	FT / RAG	ILD	> 1	1	1
Toolformer [16]	LLMs, calculator, Wikipedia search, calendar API	FT	ILD	> 1	1	1
ToolLLM [17]	REST APIs	FT / RAG	PI	> 1	1	> 1

IC: in-context learning, FT: fine-tuning, RAG: retrieval-augmented generation, PI: post-inference, ILD: in LLM decoding

A. System Components

Prior work has introduced numerous TALLMs, each tailored to specific use cases and designed to improve on previously introduced state-of-the-art systems. By comparing the designs of these systems, we can identify a design space for TALLMs and a common set of components essential to their functionality. Figure 1 summarizes these components:

LLM The LLM is the reasoning component of a TALLM and is trained and prompted to perform task planning, tool usage, and response generation. It is often deployed with a system prompt—text that is prepended to every user prompt with contextual information and instructions for the LLM.

Retriever An information retriever module is optionally used to retrieve a subset of tools that are most relevant to the user prompt and also to retrieve the latest tool documentation. An algorithm, such as BM25 [18], or a model-based approach, such as GPT-index [19], can be used.

Tool set The tool set is a collection of tools and their documentation. Tools are modules or APIs external to the LLM that can be invoked to enhance the LLM's capabilities. Tool usage allows the LLM to retrieve information not embedded in model training and to interact with third-party libraries or services. The tool set for a TALLM is the set of all tools that the LLM is aware of and can invoke; this set can be manually curated or scraped from an existing repository of tools, such as GitHub or HuggingFace. The tool set may be specifically created for a target use case, such as program repair and travel booking, or it may contain generic APIs.

The tool documentation must be explicitly provided for all tools and must include information about the tool's interface.

Tool executor A tool executor is a software component that parses a structured output of the LLM and executes tools corresponding to the API calls produced by the LLM.

Response verifier A response verifier can be optionally added to check and/or augment the LLM response. This component may enforce safety constraints or apply formatting and gate-keeping rules to improve response quality.

B. System Design Considerations

TALLMs, such as those listed in Table I[1], differ across multiple characteristics. Based on our literature survey, we identified the following five key characteristics.

Tool provisioning There are three main approaches to enabling an LLM to interface with external tools. These approaches affect the orange components in Figure 1.

1) *IC*: In-context learning [20] enables an LLM to interface with tools by including tool documentation and example usages (from the tool set) in the system prompt. In-context learning is a fast way to augment an existing LLM with tool usage capabilities as it does not require additional model training for each added tool. However, the number of tools and the comprehensiveness of tool documentation that can be provided is limited by the context length of the LLM. Chameleon, RepairAgent, REACT, API-Bank, HuggingGPT, and ART utilize in-context learning and are thus limited to a smaller tool set ($<$ 80 tools for these TALLMs).
2) *FT*: Fine-tuning an LLM on examples of tool usage can be more effective and scalable to larger tool sets. This method embeds tool information in the LLM itself by training on a dataset that contains pairs of user prompts and example tool usage for all tools in the tool set. It is, however, more expensive due to dataset curation and training costs. Gorilla, ToolAlpaca, α-UMI, ToolkenGPT, TALM, and Toolformer utilize fine-tuning.

[1]We conducted a literature survey in January 2024. The literature on TALLMs has since grown, but the design considerations are still valid.

3) *RAG*: Retrieval-augmented generation [21] allows further scalability to retrieve the latest and most relevant tools and documentation for the requested task. However, this requires training and updating a dedicated retriever module. Reta-LLM [22] uses a retriever module in addition to in-context learning. TALM, Tool-LLM, and one version of the Gorilla model are fine-tuned and inferenced with a retriever module.

Timing of tool invocations TALLMs vary in the timing of tool invocations. Some TALLMs, such as Toolformer, ToolkenGPT, ART, and TALM, are trained to produce special tokens during LLM decoding that can trigger a tool invocation. The generation of such a token pauses LLM decoding to retrieve the output of the tool invocation. The output is then appended to the response, and the LLM continues decoding tokens. On the contrary, the majority of TALLMs (Chameleon, RepairAgent, REACT, API-Bank, HuggingGPT, Gorilla, ToolAlpaca, α-UMI) invoke tools after LLM inference. Such systems can produce executable code that can contain API calls, or the LLM could generate a plan or set of instructions for tool invocation that the tool executor must parse to invoke the actual tools.

Number of tool invocations The TALLM can be designed to accomplish a task with a single tool or with multiple tools that operate in sequence or in parallel. For example, Gorilla and API-Bank invoke exactly one tool for a given user prompt, whereas other systems may invoke multiple tools.

Number of agents A single LLM agent is often used for task planning, tool invocation, and response generation. Alternatively, multiple specialized LLM agents can be used. For example, the α-UMI framework uses three separate LLMs for tool planning, tool call construction, and response summarization. While the multi-agent approach can work better due to the agent specialization, it can increase the training and inference costs and introduce multiple points of failure.

Number of agent turns The LLM can accomplish the task in a single pass or might be trained to interact with itself, other components of the system, or the user. In such an interactive approach, the outputs of tool invocations are fed back to the LLM, which acts as a planning agent and decision-making agent to determine the next step to complete the task.

C. System Failures

Traditional software engineering distinguishes between functional and non-functional requirements [23]. Similarly, failures in TALLMs can result from violations of functional requirements, such as incorrect responses, or non-functional requirements, such as poor reliability or high latency. Here we provide examples specific to TALLMs.

Functional failures A functional failure in a TALLM is one in which the TALLM does not produce the expected response or accomplish the requested task. This can manifest in a raised exception or an incorrect response.

1) *Raised exception*: A raised exception can occur at various points in the execution of a TALLM. LLM inference itself might raise an exception, if the length of the text input exceeds the token limit for the model or if auxiliary input, which could include modalities such as image, video, or audio, exceed the memory requirements of the model inference. Hallucination of a non-existent tool or argument would result in an invalid tool invocation. Input type mismatches and internal errors can also raise exceptions at any point during tool execution.
2) *Empty or incorrect response*: A TALLM might be produce no or an incorrect response, even without raising an exception. For example, a TALLM may generate code that runs successfully but does not produce any output (e.g., due to a missing return or print statement).

Defining functional correctness for a TALLM can be tricky depending on the use case. For some tasks, such as question answering, there is one, known ground truth, while for other tasks such as code generation or chatbots, there may be a range of correct and valid responses. Furthermore, since the input contains free-form text, a deployed TALLM needs to correctly reject and accept inputs based on the defined and enforced set of requirements and use cases for the system.

Non-functional failures TALLMs, just like traditional software systems, are subject to non-functional requirements such as reliability, security, and cost/latency.

1) *Reliability*: While we have discussed individual failures, an important requirement for deploying a TALLM is correctness or reliability across a set of representative inputs, known as an evaluation dataset. This evaluation dataset should contain a sufficiently wide range of inputs, including edge and critical cases.
2) *Consistency*: There is often a tradeoff between creativity/exploration and hallucination/inconsistency for LLM-based systems. Due to the non-deterministic nature of LLMs, consistency across multiple executions of the same task prompt is important. However, multiple runs of the same prompt may result in different code execution paths or task plans.
3) *Security*: Since TALLMs interact with external tools, potential security risks can arise, especially if tools can modify databases and if data passed around can create various attack surfaces or affect privacy. The tool set must be vetted to prevent the TALLM from selecting tools that violate compliance requirements. Furthermore, a TALLM should respect user permissions or access rules for tool usage or data access.
4) *Cost/latency*: LLMs can select tools from a plethora of options and can plan multiple invocations. Without proper constraints, it is possible to exceed latency and cost budgets. For example, two tools with the same functionality may have different inference latencies and computational costs (e.g., a 7B vs. a 12B parameter foundation model), and the TALLM should be able to appropriately selected between them while balancing correctness and computational resources. This requires that the tool documentation contain such information.

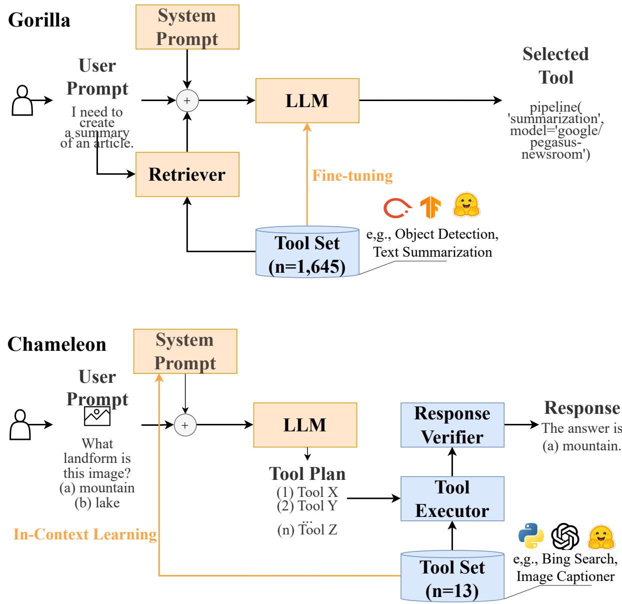

Fig. 2: Design differences between Gorilla and Chameleon.

III. TaLLM Failures and Root Causes

To gain insights into the failures that arise in TaLLMs, we selected two published TaLLMs as case studies to analyze failures and enumerate possible root causes.

We selected two TaLLMs, Gorilla [11] and Chameleon [1] (compared in Figure 2), based on the following criteria:

- Published model and evaluation code.
- Published evaluation results, including inference results and the outputs of intermediate steps.
- Differences in the TaLLM system design (Section II).

Gorilla is a fine-tuned LLaMA-based model that generates code to invoke machine learning APIs given a prompt describing a task [11]. We refer to these APIs as "tools". Specifically, the output of the Gorilla LLM is a single ML tool invocation, including the tool name and the required arguments. Invoking one of these tools downloads the requested ML model from the repository and enables local inference. The output of the LLM can be integrated into a code generation component that embeds the selected tool invocation in the correct code context and an executor component that executes the code. While Gorilla is only trained on generating the correct tool call, the model is also prompted to return example code as well. Gorilla is trained and evaluated on *three datasets* that contain ML tool calls in Python from HuggingFace Model Hub (*HuggingFace*), TensorFlow Hub (*TFHub*), or PyTorch Hub (*TorchHub*). The tool calls span various ML domains, varying in the input modality (e.g., image, language, audio) and the task type (e.g., classification, question answering).

Chameleon uses an off-the-shelf LLM (e.g., GPT-4) that is made aware of tools through in-context learning to solve complex reasoning tasks [1]. Instead of generating a single line of code, like Gorilla, the LLM is prompted to produce a tool plan to achieve the task using tools from an inventory. These tools include a knowledge retriever, image captioner model, web search, Python program generator, and program executor, some of which are LLMs themselves. These are then executed sequentially according to the plan after LLM inference completes, with the output of one tool feeding into the next as input. Chameleon is evaluated on *two question answering datasets*: *ScienceQA* [24], a multimodal scientific benchmark, and *TabMWP* [25], a mathematical benchmark involving tabular data.

We answered three research questions to gain insights into common failures observed in TaLLMs and their root causes:

- **RQ1:** What is the distribution of failures observed in TaLLMs?
- **RQ2:** What data characteristics are associated with correctness in TaLLMs?
- **RQ3:** What are the root causes for failures in TaLLMs?

To answer these research questions, we used a combination of manual and automated analysis of the published evaluation results for Gorilla and Chameleon. The published results from Gorilla for each of the three evaluation datasets contain the user prompt, the raw LLM output, the parsed LLM output. The LLM output is parsed into the tool call and the example code. Evaluation results for Chameleon on the two datasets include tool plans, intermediate outputs of each tool invocation, and final results.

A. What is the distribution of failures observed in TaLLMs?

1) Methodology: To identify failures for each TaLLM, we used the evaluation methodology and correctness criteria defined by the respective authors of the TaLLM. At a high level, Gorilla's evaluation methodology focuses only on tool plan generation by the LLM, and Chameleon's evaluation methodology focuses on evaluating response generation by the system. For Gorilla, failures are identified based on whether the tool extracted from the generated output (1) is contained in the tool set and (2) is in the same ML domain as the ground truth tool (e.g., audio classification, NLP summarization). The authors of Gorilla selected this evaluation methodology to reduce the number of false negatives as multiple tools might be suitable for the same user prompt. Furthermore, this eliminates the need for (potentially expensive) execution of ML tool calls. However, the downside to this approach is that it only partially evaluates correctness. In contrast, Chameleon is evaluated on multiple-choice question-answering datasets, with exactly one ground-truth answer for every question. The authors of Chameleon used the expected answers to define correctness.

We automatically identified the distribution of the failures in Gorilla and Chameleon by implementing Python functions that classify the LLM output according to the evaluation methodology and correctness definitions described above:

- Incorrect Output: Incorrect tool selection (Gorilla) or the tools produced an incorrect answer (Chameleon).
- No Output: No tool or a hallucinated tool (Gorilla) or the tools produced an empty answer (Chameleon).

TABLE II: Aggregation of ML domains into groups for Gorilla.

Group	HuggingFace	TFHub	TorchHub
Image	CV Depth Estimation, CV Image Classification, CV Image Segmentation, CV Image-to-Image, CV Object Detection, CV Unconditional Image Gen, CV Zero-Shot Image Classification	Image classification, Image feature vector, Image object detection, Image pose detection, Image segmentation	Semantic Segmentation, Object Detection
Language	NLP Conversational, NLP Feature Extraction, NLP Fill-Mask, NLP QA, NLP Sentence Similarity, NLP Summarization, NLP Table QA, NLP Text Classification, NLP Text Gen, NLP Text2Text Gen, NLP Token Classification, NLP Translation, NLP Zero-Shot Classification	Text classification, Text embedding, Text language model, Text preprocessing	Text-To-Speech, Text-to-Speech
Audio	Audio Audio Classification, Audio Audio-to-Audio, Audio Automatic Speech Recognition, Audio Classification, Audio Text-to-Speech, Audio Voice Activity Detection	Audio embedding, Audio event classification	Audio Separation
Video	CV Video Classification	Video classification	Video Classification
Tabular	Tabular Tabular Classification, Tabular Tabular Regression		
Multimodal	Multimodal Document QA, Multimodal Feature Extraction, Multimodal Graph Machine Learning, Multimodal Image-to-Text, Multimodal Text-to-Image, Multimodal Text-to-Video, Multimodal Visual QA, Multimodal Zero-Shot Image Classification		
RL	RL, RL Robotics		
General			Classification

RL: Reinforcement Learning; CV: Computer Vision; NLP: Natural Language Processing

TABLE III: Summary of observed failures.

Dataset		Output		
Name	**Size**	**Correct**	**Incorrect**	**None**
TALLM: Gorilla				
HuggingFace	904	626 (69.0%)	178 (20.0%)	100 (11.0%)
TFHub	685	564 (82.0%)	74 (11.0%)	47 (7.0%)
TorchHub	186	140 (75.0%)	15 (8.0%)	31 (17.0%)
TALLM: Chameleon				
TabMWP	7686	7592 (98.8%)	65 (0.8%)	29 (0.4%)
ScienceQA	4241	3670 (86.5%)	571 (13.5%)	0 (0.0%)

2) Results: For **Gorilla**, Table III shows between 18% and 31% failures for each of the three evaluation datasets, in line with Gorilla's reported results. Considering the overall distribution of failures, we observe that for HuggingFace and TFHub, there are roughly twice as many failures in which an incorrect output is produced as no output, but the opposite trend holds for TorchHub. This is likely because TorchHub has fewer ML domains represented, and thus the tool selection is an easier task for the TALLM.

For **Chameleon**, the failure counts in Table III align with Chameleon's reported results on the two datasets. Chameleon has a higher accuracy on TabMWP compared to ScienceQA. This could be due to focused modality of tabular data and the inclusion of tabular-specific tools for TabMWP as opposed to the broader domain of science with different input modalities. We also note that roughly 30% (29/94) of all failures in TabMWP produce no output instead of an incorrect output, compared to 0% for ScienceQA. RQ3 investigates this further.

3) Conclusion: 40% of all Gorilla failures and 4% of all Chameleon failures manifest as empty output, which may indicate masked failures or inadequate failure handling.

B. What data characteristics are associated with correctness in TALLMs?

1) Methodology: To gain a more comprehensive understanding of the distribution of failures in the two TALLMs, we studied associations between data characteristics and correctness. By identifying data characteristics that are negatively associated with correctness, we can identify areas in which better training data or specialized tools can improve performance. For each TALLM, we fit a logistic regression model—modeling correctness as a function of data characteristics.

For Gorilla, we fit a single model across datasets on the following data characteristics:

- Source dataset: HuggingFace, TFHub, or TorchHub.
- ML domain group: Aggregated ML domain group, as summarized in Table II.
- Number of tools: Number of tools in a particular ML domain.

For Chameleon, we fit separate models on the two datasets as they differ in question type and data characteristics. For TabMWP, we consider the following data characteristics:

- Question difficulty ("grade"): Grade levels, ranging from 1 to 8, used as a proxy for difficulty, where higher grade levels are considered more difficult.
- Question type ("question_type"): Indicates whether a free text response or a multiple-choice selection is expected.
- Answer type ("answer_type"): The format of the correct answer, such as text or number (see [25] for details).

For ScienceQA, we consider the following data characteristics:

- Question difficulty ("grade"): Same as TabMWP above. Here, values range from 1 to 12.
- Question type ("has_image"): Indicates whether the question is asked about an attached image.
- Question domain ("subject"): The subject (language science, natural science, and social science) of the question.

TABLE IV: Correctness model for Gorilla.

Feature	Coefficient	P value
Intercept	0.83	< 0.01
Dataset: TFHub	-0.70	< 0.01
Dataset: TorchHub	0.04	0.92
ML Domain Group: General	-0.48	0.34
ML Domain Group: Image	-0.24	0.29
ML Domain Group: Language	-0.41	0.05
ML Domain Group: Multimodal	-0.03	0.92
ML Domain Group: RL	-1.29	< 0.01
ML Domain Group: Tabular	-1.24	< 0.01
ML Domain Group: Video	1.37	0.03
Num. Tools in ML Domain	0.01	< 0.01

TABLE V: Correctness model for Chameleon.

Feature	Coefficient	P value
Dataset: TabMWP		
Intercept (Answer Type: True/False)	8.30	< 0.01
Grade	-0.23	< 0.01
Answer Type: Decimal	-0.42	0.19
Answer Type: Extractive	0.24	0.61
Answer Type: Integer	0.66	0.06
Answer Type: Other	0.09	0.89
Num. Table Columns	-0.91	< 0.01
Num. Table Rows	-0.10	0.14
Dataset: ScienceQA		
Intercept (Subject: Language Science)	2.15	< 0.01
Grade	0.03	0.36
Image	-2.17	< 0.01
Subject: Natural Science	1.62	< 0.01
Subject: Social Science	4.13	< 0.01
Grade x Subject: Natural Science	-0.03	0.53
Grade x Subject: Social Science	-0.63	< 0.01

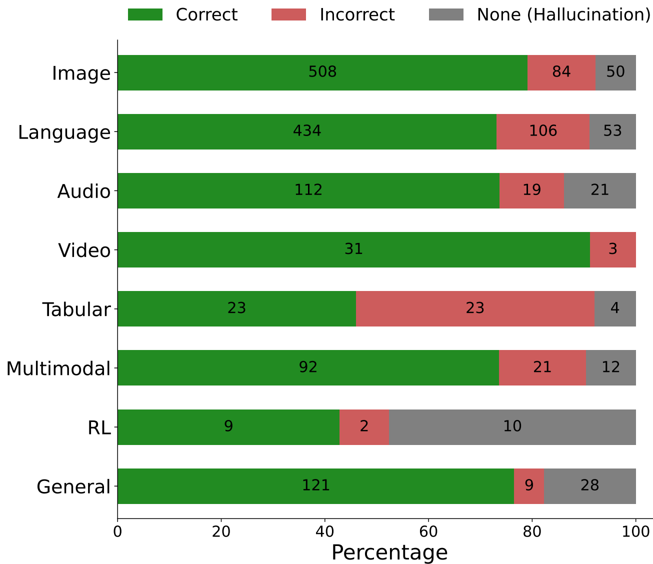

Fig. 3: Gorilla failure distribution per ML domain group.

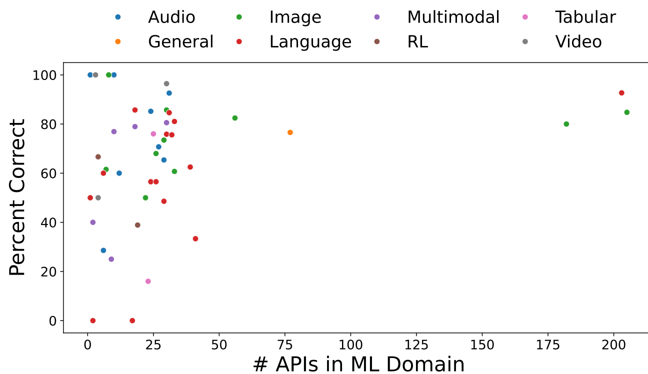

Fig. 4: Correctness vs. number of APIs for Gorilla.

2) Results: For **Gorilla**, Table IV shows the coefficients and p values for the logistic regression model, which models correctness as a function of the data characteristics. These results indicate that the domains of reinforcement learning (RL) and tabular data (Tabular) have a significant negative influence on correctness (relative to Audio), while video inputs (Video) have a positive influence. Figure 3 visualizes the failure distributions per ML domain group and supports this observation. Taking a closer look, we observe that RL has many hallucinations, but 6 of these were incorrectly classified as hallucinations because the LLM output was not able to be parsed even though a tool call was selected. In the Tabular domain, many of the failures are due to selecting a tabular tool for classification instead of regression, indicating that the LLM may not understand the semantic differences in the user prompt between the two tasks. The video domain is most likely to have a high accuracy due to only having one subdomain (Table II).The three failures in this domain were due to incorrectly selecting a tool from the NLP domain.

Furthermore, we observe only a weak correlation between the number of tools in a ML domain and correctness (Figure 4). The correlation is even weaker when the three outliers are removed. We leave a deeper investigation and a controlled study for future work.

For **Chameleon**, Table V shows the logistic regression models for predicting correctness on the two datasets. The results indicate that for the TabMWP dataset, grade level and number of table columns have a significant negative influence on correctness, indicating that question difficulty and complexity reduce correctness for Chameleon. For ScienceQA, we observe that the presence of an image in the question has a significant negative influence on correctness. By adding an interaction term between grade level and social science questions, we see that its negative and significant value indicates that correctness decreases as grade level increases for social science topics, while not for the other subjects. The number of questions that ask to identify a highlighted country, continent, or ocean from an image with no text increase as grade levels increase for social sciences. Given the available tools, this task seems especially hard for Chameleon, as reflected in Figure 5. For other subjects, we do not observe such a trend.

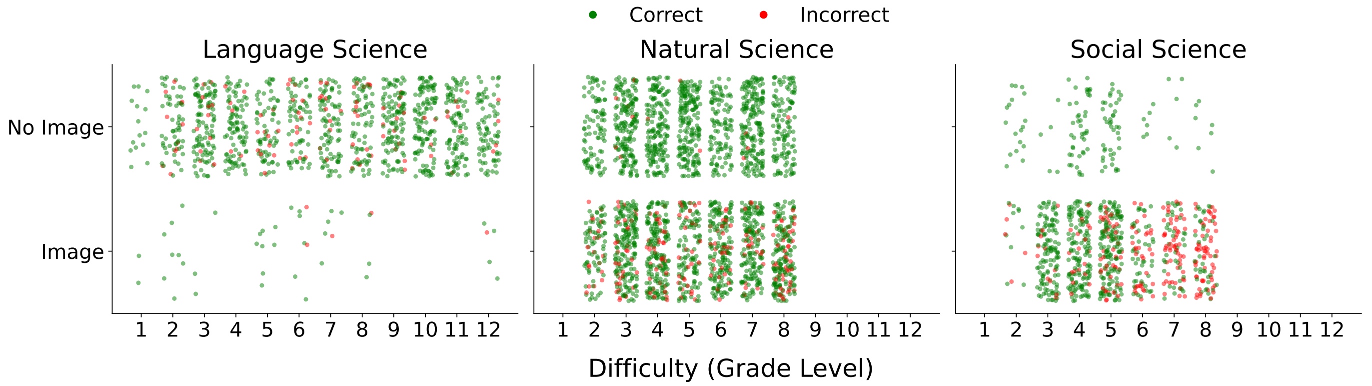

Fig. 5: Distribution of correctness in Chameleon broken down by data characteristics.

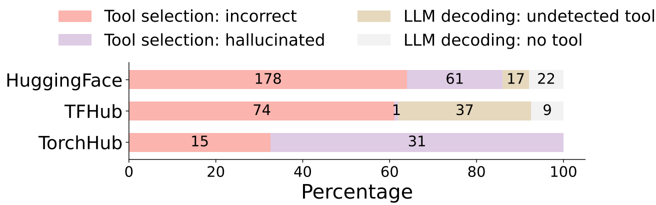

Fig. 6: Distribution of root causes for all Gorilla failures.

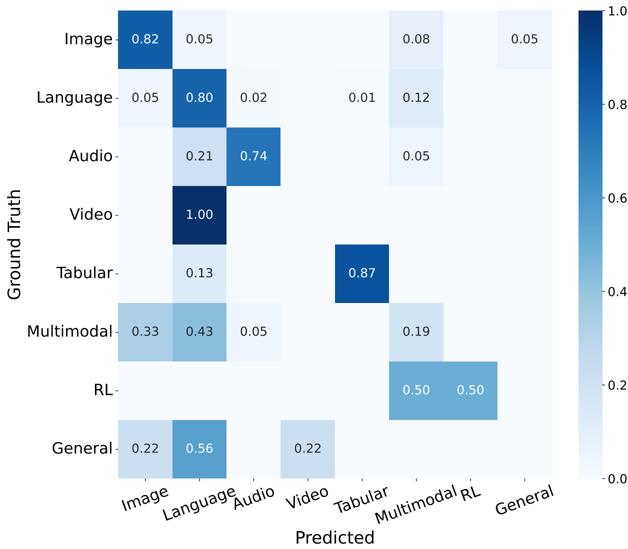

Fig. 7: Distribution of predicted ML domain groups for all "incorrect tool" failures in Gorilla.

3) Conclusion: We observe a significant negative association between the domain groups of reinforcement learning and tabular data for the Gorilla model. For Chameleon, we observe significant negative associations for grade and number of table columns or TabMWP, and for the presence of images for ScienceQA. We also observe an interaction effect specifically between grade and the subject of social science.

C. What are the root causes for failures in TALLMs*?*

1) Methodology: We analyzed the failures observed in Gorilla and Chameleon (last two columns in Table III) to identify root causes. For Gorilla, we automatically classified the entire dataset by iteratively implementing classification rules. First, we started with a random sample of about 5% of failures in each dataset. Second, we manually inspected each failure and added a free-text failure description. Third, we categorized the text descriptions into a smaller set of failure modes and implemented corresponding classification rules.

For Chameleon, the dataset is substantially larger and not amenable to automatic classification. Thus, we randomly sampled 10% of the failures in each dataset and categorized the failure descriptions for each into a smaller set of failure modes. While the sample may have missed rare failure modes, we observed quick saturation during our manual analysis and our failure modes match those reported by the Chameleon authors.

2) Results: For **Gorilla**, we identified four root causes—two related to tool selection and two related to LLM decoding. Figure 6 summarizes the results.

1) *Tool selection: incorrect tool*: One cause for failures is selecting an incorrect tool—one that exists in the tool set but in the wrong ML domain. Figure 7 shows the distribution of predicted ML domain groups for each ground truth group of all "incorrect tool" failures. For many groups (image, language, audio, tabular), the majority of failures involve selecting tools that are still within the same group (e.g., 82% of image "incorrect tool" failures did select an image tool, even though the specific ML domain was incorrect). For tabular, multimodal, and general, the most common predicted group was language.
2) *Tool selection: hallucinated tool*: Another cause for failures is hallucination—selecting a tool that does not exist in the tool set and resulting in no output. As shown in Table VI, the hallucinated tool names often resemble tools in the tool set, with a few words replaced by keywords from the question. Of the 61 hallucinations observed in the HuggingFace dataset, 13 were script

TABLE VI: Examples of hallucinations in Gorilla.

Question	Hallucinated Tool	Closest Real Tool
Generate a new image based on the online database of bedroom art.	lllyasviel/control_v11p_sd15_bedroom	lllyasviel/control_v11p_sd15_inpaint
We want to develop a chatbot that can engage with multilingual users.	microsoft/GODEL-v1_1-base-multilingual-encoder-decoder	microsoft/GODEL-v1_1-base-seq2seq
We are building a robot for hopping in an environment, trained using Decision Transformers.	edbeeching/decision-transformers-gym-hopper-medium	edbeeching/decision-transformer-gym-walker2d-expert

executions (run.sh) or binary invocations (mlagents-load-from-hf, load_model_ensemble_and_task_from-_hf_hub) were present in the training data unable to be parsed from the LLM output in the evaluation.

3) *LLM decoding: undetected tool*: Another root cause involves LLM constrained decoding. The LLM is expected to produce output in a constrained format that can be parsed to identify the selected tool. The tool invocation is identified by a string token in the LLM output, and the prompt to the LLM asks the model to produce output abiding by this format. However, even though the response may contain the correct tool call, this identifier token may be missing, preventing the tool executor from detecting the selected tool.
4) *LLM decoding: no tool*: The LLM might also fail to select a tool, preventing the tool executor from executing any tools. For instance, for a user prompt requesting to summarize a news article, the LLM hallucinated an article instead of selecting a text summarization tool.

For **Chameleon**, we identified three root causes.

1) *Tool selection: missing tool*: One root cause for failures in Chameleon was suboptimal tool plan generation by the LLM. For example, in several instances, Chameleon did not select tools for gaining semantic information about the input question. In other cases, Chameleon generated a tool plan that directly generated the final answer without using external tools to retrieve knowledge or process certain data, thus resulting in misunderstanding of the question and an incorrect result.
2) *Tool: faulty tool*: Another root cause for failures are defects within tools. In Chameleon, we observed the image captioning tool providing incorrect captions for some images, leading to overall failures. Additionally, we discovered a subtle bug in the program generator and executor tools (Python implementation that operates on a string representing code) which affected some inputs.
3) *Tool set: missing tool*: A third root cause is where the tool set lacks certain required tools. The TALLM might select the best tool available in the tool set, but this may not be able to accomplish the task correctly.

3) Conclusion: We identified six unique root causes for failures in Gorilla and Chameleon, covering tool selection (incorrect tool, hallucinated tool, missing tool), LLM decoding (undetected tool, no tool), tool execution (faulty tool), and tool set (missing tool). Not all of these lie within the LLM and require model retraining, underscoring the need for holistic system testing.

TABLE VII: Analysis of sampled correct responses in Gorilla.

Dataset	Tool Selection		Code Execution		Code Correctness
	Functional	Available	w/o Repair	w/ Repair	
HuggingFace	10	9	0	5	4
TFHub	10	9	0	6	0
TorchHub	10	10	0	0	0

IV. Recommendations

From our literature survey of TALLMs and our analysis of failures in Gorilla and Chameleon, we observed a need for contextualized evaluation methodologies, a comprehensive understanding of failure modes and root causes, and robust testing and repair mechanisms for deployment and adoption. In this section, we outline recommendations in these areas.

A. Evaluation Methodologies

Methodologies that assess the accuracy of TALLMs must precisely define correctness and clearly identify what components are being evaluated. This is crucial to enable fair comparisons and proper contextualization of results. The definition of correctness affects which components of the system are evaluated and what types of failures are surfaced or masked.

For example, Gorilla's definition of correctness buckets all tools within the same ML domain and does not execute tools—effectively measuring the proportion of how often a selected tool exists and falls into the expected ML domain, as opposed to the proportion of correct tool selections and system responses. While this minimizes execution costs and false negatives (multiple tools may be suitable for the same task), it does not fully capture task correctness. Our manual analysis of 30 sampled examples (Table VII) demonstrates this:

- *Tool functionality*: Whether the tool selected provides the necessary functionality to accomplish the user task. This is determined manually by comparing the user prompt to the tool documentation.
- *Tool availability*: Whether the selected tool is still available in the tool set and not deprecated.
- *Code execution*: Whether the code can run successfully without runtime errors either with no changes (w/o Repair) or with trivial changes (w/ Repair). A trivial change is strictly one of the following: updating placeholders

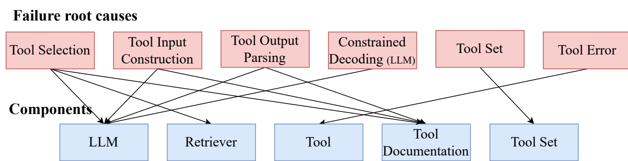

Fig. 8: TALLM failure root causes and affected components.

with actual paths to input data, fixing escaped quotations from the string parsing of the LLM output.

- *Code correctness*: Whether the generated example code fully accomplishes the intended task and produces the correct output requested in the user prompt. Note that a prerequisite for code correctness is that the code is executable (with or without trivial repairs).

While the tool selection for all examples was adequate, none of the examples contained code that could be executed without repairs. 19 out of the 30 examples required non-trivial repairs to be executed, and only 4 examples produced code that actually accomplished the task. These results are expected, as the training objective for Gorilla was to produce a single tool invocation and not the surrounding code context. However, this demonstrates how such a correctness metric limits the evaluation to only the LLM and not the entire system.

As another example, in Chameleon's evaluation, 28% of correct answers resulted from a design rule defaulting to the first choice if the tools fail to generate a valid result. This artificially inflates performance in an nontransparent manner and demonstrates how implicit and non-evaluated design choices may distort evaluation results. While such a design choice might be suitable, it must be documented and accounted for in the evaluation methodology to avoid skewed results. Furthermore, it may be better for a system to issue no response instead of guessing answers.

B. Root Causes

Based on our empirical analysis and our literature survey, we identified six main TALLM failure root causes and what system components may be affected (Figure 8). While these may not be comprehensive, they provide a starting point for designing testing and repair approaches for TALLMs.

1) Tool selection: Root causes in this category include incorrectly selecting no tool, selecting a wrong tool (or tools), and selecting a tool that does not exist (hallucination). These can occur due to bugs in the LLM, the retriever module, or the tool documentation.

2) Tool input construction: Failures can also be caused by poor tool input/argument construction. For example, the argument name might be hallucinated, or the input might fail a precondition (e.g., data type). Failures with this root cause often stem from missing, incorrect, or misinterpreted documentation. Alternatively, they may also indicate a poorly generalizing LLM or retriever.

3) Tool output parsing: Another root cause is incorrect parsing of the output of an invoked tool. This could also occur due to missing, incorrect, or misinterpreted documentation. Similar to tool input construction, failures with this root cause are due to bugs in the LLM or the tool documentation.

4) LLM constrained decoding: Constrained decoding is a common method to ensure that the LLM produces output in a constrained and consistent format such as JSON, which can be parsed for tool invocations. Failures in LLM constrained decoding can manifest in different ways. One way is that the output of the LLM might contain a correctly selected tool but formatted incorrectly and thus undetected. Alternatively, the output of the LLM might be completely invalid and not even contain any tool selection. Failures with this root cause often localize to a bug in the LLM and can be repaired with either LLM training or prompt tuning.

5) Tool set: Some failures might be root caused to a poor composition of tools in the tool set. Specifically, missing functionality in the tool set would require new tools to be added; deprecated tools must be removed.

6) Tool error: Failures can also occur within specific tools and can manifest by exceptions raised during usage or sub-optimal performance as detected by inspecting stack traces and intermediate outputs.

C. End-to-End Testing

The structured analysis of TALLM failures highlights diverse points of failure and corresponding root causes, emphasizing the need for holistic testing beyond the LLM itself.

Similar to traditional software, TALLMs have multiple points of failure. Thus, comprehensive test coverage across all the components of the system, the whole tool set, and various input arguments is critical. Furthermore, testing various combinations of tools used and these different factors improves test coverage. As discussed by [26], intrinsic evaluation and real-world evaluation can diverge significantly, as observed in their work in deploying a TALLM for code review, and hence end-to-end testing is critical for bridging the gap between beating a benchmark and usable deployment (see also Table VII).

D. Continuous Testing

Another area of testing that is critical for TALLMs is continuous testing. As we noticed when executing some of the tool invocations produced by Gorilla, tool functionality, documentation, and availability can change over time. And if the overall system is not robust to these changes, it can cause runtime failures. Thus, TALLM testing must account for the evolution of tools and documentation in the real-world.

E. Non-Functional Testing

It is also important to test TALLMs for their non-functional requirements. One of the non-functional requirements previously discussed is consistency across similar invocations, which is not typically covered by LLM tests. Many TALLM systems operate on free text user input, and slight variations in phrasing could result in different LLM responses. A form

of fuzz testing could be used to test such robustness. Such fuzz testing can even be used to test and expose security vulnerabilities in TALLMs [27]. Furthermore, TALLMs can be tested on their ability to respect constraints regarding the cost and accuracy of tools [11]. Depending on the target use case, TALLMs should be able to balance the cost of tool invocations against performance and accuracy.

F. Repair Methods

The failure root cause can inform the method of repair.

1) Repairing the LLM: A common repair method is repairing the LLM itself, either through in-context learning or fine-tuning. In-context learning modifies the system prompt with additional details or constraints, while fine-tuning augments training data with failure cases or corner cases. If failures can be correlated with specific data characteristics, the LLM may be fine-tuned on additional data with the same characteristics.

2) Repairing the tool set: Failures can also be caused by inadequacies in the tool set, which can be repaired by adding, removing, or replacing tools. In some cases, improving documentation quality and coverage may also improve the TALLM performance. Furthermore, another approach to make a TALLM more resilient to issues in the tool set is to combine usage with a document retriever that retrieves the latest documentation and tools at inference time [11].

3) Runtime assertions and heuristics: Another method of repair involves runtime assertions and rules that can be applied to the output of LLM inference or tool invocations (e.g., [7]). This can be used to modify the LLM prompt or output of the LLM to re-trigger LLM inference with a slightly different input and constraint. For example, as demonstrated by Chameleon, this can be used to restrict the TALLM from using a certain deprecated tool without LLM re-training or enforce a certain pre-condition for a tool that the LLM is observed to not respect. However, applying such rules can mask underlying failures, as we observed.

V. Related Work

A. Benchmarks for TALLM*s*

Recent work has developed benchmarks to evaluate the task planning and tool usage capabilities of LLMs. APIBank [5] was one of the first benchmarks for TALLMs, providing training/evaluation datasets and defining evaluation metrics specifically for TALLMs. Other benchmarks developed includebToolQA [28], TaskBench [29], MetaTool [30], and UltraTool [6]. Furthermore, various datasets (APIBench [11], ToolQA [28], ToolBench [17], ToolAlpaca [12]) have been constructed to be used for fine-tuning and evaluation of TALLMs. These benchmark datasets can be manually created or generated by an LLM. Often, these benchmarks evaluate some combination of the tool planning, tool retrieval, and tool usage capabilities of LLMs. However, such benchmarks contain a fixed set of example inputs and downstream tasks, and the evaluation metrics are often limited. TALLM evaluation sometimes avoids executing external tools due to complexity, cost, and LLM non-determinism, and instead relies on on heuristics or another LLM for evaluation. With regards to failure analysis, ToolQA identified common failures like argument errors, hallucinations, and infeasible actions, but lacked detailed analysis to uncover their root causes [28].

B. Evaluation Metrics for TALLM*s*

TALLMs are often evaluated on benchmark datasets constructed for certain tasks. Depending on the type of TALLM, the text output of the LLM might be the final response to the task prompt or the text output might contain a tool usage plan or generated code that must be executed by a separate module. Evaluation may include computing accuracy and quality metrics on either the text generated or the final response [28]. Prior works have observed hallucinations in both the API name and the argument names [11], and this has become a common metric, in addition to accuracy and quality. Using GPT-4 or another LLM as a quality and correctness evaluator for both the process and final answer has been a common paradigm as well [31].

VI. Conclusion

Augmenting LLMs with tools provides a powerful way to boost capabilities of LLMs. However, testing TALLMs is challenging due to their interactions and complex execution paths involving external tools and due to potentially expensive tool invocations. Our analyses highlight the need for better testing and repair methods for TALLMs. Our analyses informed a set of actionable recommendations around the need for end-to-end testing and repair methods.

Future work should focus on developing approaches for creating comprehensive test suites for TALLMs for deployment as well methods for localizing failures to data characteristics and components of the system and repairing them. Given the powerful applications unlocked by TALLMs and the increasing research in this space, such work around understanding TALLM failures and developing testing and repair approaches is critical for successful and safe deployment of TALLMs.

Our work demonstrates that even the state of the art TALLMs, such as Gorilla, suffer from various failures and hallucinations that need to be addressed to improve reliability and robustness. Furthermore, the observed failures are often in very similar ML domains, making it difficult to evaluate whether a selected tool would actually produce a correct result without execution.

Future work in this area could explore different methods for test data generation and data sampling for retraining. The combination of repeated testing and repair can help improve both the performance of TALLMs and their robustness, in the presence of distribution shifts.

Acknowledgments

This work is supported in part by National Science Foundation grant CCF-1942055.

REFERENCES

[1] P. Lu, B. Peng, H. Cheng, M. Galley, K.-W. Chang, Y. N. Wu, S.-C. Zhu, and J. Gao, "Chameleon: Plug-and-play compositional reasoning with large language models," 2023.

[2] S. Hao, T. Liu, Z. Wang, and Z. Hu, "ToolkenGPT: Augmenting frozen language models with massive tools via tool embeddings," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[3] I. Bouzenia, P. Devanbu, and M. Pradel, "Repairagent: An autonomous, llm-based agent for program repair," 2024.

[4] S. K. Dam, C. S. Hong, Y. Qiao, and C. Zhang, "A complete survey on llm-based ai chatbots," 2024. [Online]. Available: https://arxiv.org/abs/2406.16937

[5] M. Li, Y. Zhao, B. Yu, F. Song, H. Li, H. Yu, Z. Li, F. Huang, and Y. Li, "Api-bank: A comprehensive benchmark for tool-augmented llms," 2023.

[6] S. Huang, W. Zhong, J. Lu, Q. Zhu, J. Gao, W. Liu, Y. Hou, X. Zeng, Y. Wang, L. Shang, X. Jiang, R. Xu, and Q. Liu, "Planning, creation, usage: Benchmarking llms for comprehensive tool utilization in real-world complex scenarios," 2024.

[7] S. Shankar, H. Li, P. Asawa, M. Hulsebos, Y. Lin, J. D. Zamfirescu-Pereira, H. Chase, W. Fu-Hinthorn, A. G. Parameswaran, and E. Wu, "Spade: Synthesizing data quality assertions for large language model pipelines," 2024.

[8] S. Arcadinho, D. Aparicio, and M. Almeida, "Automated test generation to evaluate tool-augmented llms as conversational ai agents," 2024. [Online]. Available: https://arxiv.org/abs/2409.15934

[9] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *The Eleventh International Conference on Learning Representations*, 2023.

[10] Y. Shen, K. Song, X. Tan, D. Li, W. Lu, and Y. Zhuang, "Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face," 2023.

[11] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez, "Gorilla: Large language model connected with massive apis," 2023.

[12] Q. Tang, Z. Deng, H. Lin, X. Han, Q. Liang, B. Cao, and L. Sun, "Toolalpaca: Generalized tool learning for language models with 3000 simulated cases," 2023.

[13] W. Shen, C. Li, H. Chen, M. Yan, X. Quan, H. Chen, J. Zhang, and F. Huang, "Small llms are weak tool learners: A multi-llm agent," 2024.

[14] B. Paranjape, S. Lundberg, S. Singh, H. Hajishirzi, L. Zettlemoyer, and M. T. Ribeiro, "Art: Automatic multi-step reasoning and tool-use for large language models," 2023.

[15] A. Parisi, Y. Zhao, and N. Fiedel, "Talm: Tool augmented language models," 2022.

[16] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language models can teach themselves to use tools," 2023.

[17] Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, S. Zhao, L. Hong, R. Tian, R. Xie, J. Zhou, M. Gerstein, D. Li, Z. Liu, and M. Sun, "Toolllm: Facilitating large language models to master 16000+ real-world apis," 2023.

[18] S. E. Robertson and H. Zaragoza, "The probabilistic relevance framework: Bm25 and beyond," *Found. Trends Inf. Retr.*, vol. 3, pp. 333–389, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:207178704

[19] J. Liu, "LlamaIndex," 11 2022. [Online]. Available: https://github.com/jerryjliu/llama_index

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

[21] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," 2021.

[22] J. Liu, J. Jin, Z. Wang, J. Cheng, Z. Dou, and J.-R. Wen, "Reta-llm: A retrieval-augmented large language model toolkit," 2023.

[23] A. van Lamsweerde, "Requirements engineering in the year 00: a research perspective," in *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*, 2000, pp. 5–19.

[24] P. Lu, S. Mishra, T. Xia, L. Qiu, K.-W. Chang, S.-C. Zhu, O. Tafjord, P. Clark, and A. Kalyan, "Learn to explain: Multimodal reasoning via thought chains for science question answering," in *Advances in Neural Information Processing Systems*, A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, Eds., 2022.

[25] P. Lu, L. Qiu, K.-W. Chang, Y. N. Wu, S.-C. Zhu, T. Rajpurohit, P. Clark, and A. Kalyan, "Dynamic prompt learning via policy gradient for semi-structured mathematical reasoning," in *International Conference on Learning Representations (ICLR)*, 2023.

[26] M. Vijayvergiya, M. Salawa, I. Budiselić, D. Zheng, P. Lamblin, M. Ivanković, J. Carin, M. Lewko, J. Andonov, G. Petrović, D. Tarlow, P. Maniatis, and R. Just, "Ai-assisted assessment of coding practices in modern code review," in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, ser. AIware '24, vol. 24. ACM, Jul. 2024, p. 85–93. [Online]. Available: http://dx.doi.org/10.1145/3664646.3665664

[27] D. Yao, J. Zhang, I. G. Harris, and M. Carlsson, "Fuzzllm: A novel and universal fuzzing framework for proactively discovering jailbreak vulnerabilities in large language models," in *ICASSP 2024 - 2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2024, pp. 4485–4489.

[28] Y. Zhuang, Y. Yu, K. Wang, H. Sun, and C. Zhang, "Toolqa: A dataset for llm question answering with external tools," 2023.

[29] Y. Shen, K. Song, X. Tan, W. Zhang, K. Ren, S. Yuan, W. Lu, D. Li, and Y. Zhuang, "Taskbench: Benchmarking large language models for task automation," 2023.

[30] Y. Huang, J. Shi, Y. Li, C. Fan, S. Wu, Q. Zhang, Y. Liu, P. Zhou, Y. Wan, N. Z. Gong, and L. Sun, "Metatool benchmark for large language models: Deciding whether to use tools and which to use," 2024.

[31] S. Dhuliawala, M. Komeili, J. Xu, R. Raileanu, X. Li, A. Celikyilmaz, and J. Weston, "Chain-of-verification reduces hallucination in large language models," 2023. [Online]. Available: https://arxiv.org/abs/2309.11495