Meta-Tool: Unleash Open-World Function Calling Capabilities of General-Purpose Large Language Models

Shengqian Qin[1,3], **Yakun Zhu**[1,3], **Linjie Mu**[1,3],
Shaoting Zhang[1], **Xiaofan Zhang**[1,2,3*]

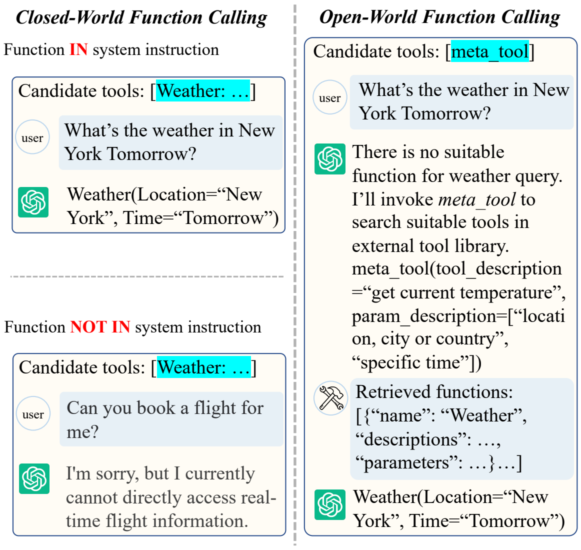
[1]Shanghai Jiao Tong University, [2]SII, [3]SPIRAL Lab
Correspondence: xiaofan.zhang@sjtu.edu.cn

Abstract

Large language models (LLMs) have showcased remarkable capabilities as autonomous agents when augmented with external tools. Equipped with fixed tool sets, LLMs struggle with addressing diverse user inquiries in open-world tasks. To evaluate and boost the performance of LLMs in dealing with complex demands in the real-world, we propose open-world function calling, where LLMs need to retrieve suitable tools from a pre-defined external tool library and use retrieved tools to resolve the user's problem. We introduce Meta-Tool, a versatile and plug-and-play tool retrieval system as the access of LLMs to external tool library. Drawing inspiration from the myriad of enhanced approaches associated with Retrieval-Augmented Generation (RAG), Meta-Tool employs a *hypothesize-retrieve-invoke* framework. We further propose Meta-Bench, a comprehensive benchmark for evaluating LLMs in open-world function calling and associated tasks. Meta-Bench encompasses $2,800$ dialogues and $7,361$ tools, spanning ten distinct scenarios to provide robust and diverse test categories. In conjunction, we present MT-LLaMA, a finetuned version of LLaMA-3.1, which exhibits remarkable performance improvements. Our empirical experiments reveal that Meta-Tool significantly enhances the ability of advanced LLMs to retrieve and leverage the most suitable tools compared to previous tool retrieval methods. Moreover, our fine-tuning enables even smaller-sized LLMs to achieve comparable even exceeding results to GPT-4o. Both the benchmark and the model are made publicly available at https://github.com/qinshengqian/Meta-Tool to foster further research and development in the field.

1 Introduction

In recent years, Large Language Models (LLMs) (Achiam et al., 2023; Dubey et al., 2024; Bai et al., 2023; Liu et al., 2024a) have made significant strides, demonstrating remarkable capabilities in various domains such as code generation (Roziere et al., 2023; Hui et al., 2024) and mathematical reasoning (Guo et al., 2025; Jaech et al., 2024), often comparable even exceeding human performance. This is largely due to their powerful knowledge bases and zero-shot abilities. Nevertheless, they struggle with specific tasks, such as calculator and calendar (Schick et al., 2023), which can be effectively accomplished through existing tools. Moreover, they face substantial limitations in interacting dynamically with the real world (Yao et al., 2022).



Candidate tools: [Weather: …]

What's the weather in New York Tomorrow?
Weather(Location="New York", Time="Tomorrow")

Candidate tools: [Weather: …]

Can you book a flight for me?
I'm sorry, but I currently cannot directly access real-time flight information.

Candidate tools: [meta_tool]

What's the weather in New York Tomorrow?
There is no suitable function for weather query. I'll invoke *meta_tool* to search suitable tools in external tool library.
meta_tool(tool_description="get current temperature", param_description=["location, city or country", "specific time"])
Retrieved functions: [{"name": "Weather", "descriptions": …, "parameters": …}…]
Weather(Location="New York", Time="Tomorrow")

Figure 1: Comparison between closed-world and open-world function calling. In closed-world function calling, the predefined tools are strictly matched with a limited set of scenarios. In contrast, integrated with an external tool library, open-world function calling offers a more dynamic and scalable solution. These tools can then be retrieved and utilized by meta-tool in a wide range of scenarios, allowing for greater adaptability and innovation.

To address these problems, Researchers have explored the integration of LLMs with external

---

[*] Corresponding author

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 30653–30677
July 27 - August 1, 2025

tools to expand the boundaries of LLMs' capacities. They endow LLMs with the ability to utilize one or several basic tools by embedding tool-related knowledge into the implicit parameters of the models (Schick et al., 2023; Komeili, 2021) or through the carefully designed prompts (Wu et al., 2023; Shen et al., 2024), employing both zero-shot and few-shot learning methods. With the increasingly accelerating progress in function calling techniques, subsequent advancements in tool-augmented LLMs focus on training these models to dynamically handle any tools referenced in the system instruction, pushing the boundaries towards supporting a wider array of tools and facilitating more intricate, multi-turn dialogues (Tang et al., 2023; Zhuang et al., 2023; Zhang et al., 2024; Lin et al., 2024).

LLMs equipped with a manually selected set of tools can operate effectively as agent systems within specific domains. However, their utility is based on the prerequisite that users are aware of which tools will be employed. This scenario essentially relegates the LLM to the roles of tool selection and slot filling, limiting LLMs to operate within a confined scope of intelligence, as shown in Figure 1. To evaluate and boost the performance of LLMs in dealing with complex demands in the real-world, we propose open-world function calling, where LLMs need to interact with a fully open-world environment, and discretionarily retrieve appropriate tools from the tool library in response to task-oriented user queries. With the advancement of LLM's ability to process long context (Ding et al., 2023; Su et al., 2024; Peng et al., 2023), one straightforward approach is to inject all tool definitions into prompts. While feasible, this approach is inefficient, as it significantly increases useless computational resource consumption. Integrating a retriever with LLMs has proven to be a practical solution (Qin et al., 2023; Patil et al., 2023; Li et al., 2023). However, the real challenge lies in enabling LLMs to discretionarily decide when to activate the retriever and accurately retrieve the most appropriate tools.

To bridge this gap, we introduce Meta-Tool, a comprehensive tool retriever system designed to empower LLMs with the ability to autonomously retrieve tools in a hypothesize-retrieve-invoke framework. LLMs need to first hypothesize a tool that is appropriate for the context, and describe the functionality of the tool as well as its parameters. Subsequently, the LLMs retrieve the specified tool and proceed to invoke it according to retrieval results. As is shown in Figure 4, encapsulated in a standard JSON format, Meta-Tool seamlessly integrates as a plug-and-play module within prompts, ensuring compatibility with both built-in tools and other tools present in prompts. In our quest to assess the performance of leading LLMs in open-world environments, we augment existing high-quality datasets to construct the Meta-Bench. This pioneering benchmark is tailored for open-world function calling, boasting a collection of 2,800 dialogues that span across 7,361 tools. Meta-Bench is structured with different difficulty levels and encompasses 10 distinct scenarios, encompassing both open-world and closed-world scenarios. To further boost the capabilities of open-source models in this domain, we propose Meta-Tool-LLaMA (MT-LLaMA), which is fine-tuned based on LLaMA-3.1-8B-Instruct, which yields a remarkable enhancement in the model's proficiency for open-world function calling.

In summary, our contributions are as follows:

- **Meta-Tool**: Based on the standard JSON format for tool definitions and advanced text embedding models, we introduce a novel tool retrieval approach in the *hypothesize-retrieve-invoke* framework.
- **Meta-Bench**: We present an automated framework designed for open-world augmentation of pre-existing high-quality tool-use datasets. We introduce Meta-Bench, a benchmark specifically tailored for open-world function calling scenarios.
- **MT-LLaMA**: Building upon these advancements, we fine-tune LLaMA-3.1.8B-Instruct, achieving a substantial advancement in its performance for open-world function calling.

2 Related Works

LLMs and Benchmarks for Function Calling. The exploration of LLM function calling stems from developing specialized tool-use datasets to train LLMs in mastering common tools, such as calculators (Schick et al., 2023) and web search engines (Komeili, 2021). Owing to the robust zero-shot capabilities of LLMs, by injecting tools into prompts, researchers guide LLMs to reason step by step, select, and execute external tools (Shen et al., 2024; Wu et al., 2023). Subsequent studies (Patil et al., 2023; Qin et al., 2023; Li et al., 2023;

Benchmark	**#Dialogues**	**#Tools**	**Multi-turn**	**Open-World Function Calling**	**Irrelevance Detection**
ToolQA (Zhuang et al., 2023)	1530	13	✗	✗	✗
ToolAlpaca (Tang et al., 2023)	3938	400	✓	✗	✗
API-Bank (Li et al., 2023)	2202	2211	✓	✓	✗
ToolBench (Qin et al., 2023)	12657	16464	✗	✓	✗
BFCL (Yan et al., 2024)	4751	3372	✓	✗	✓
Meta-Bench (Ours)	2800	7361	✓	✓	✓

* BFCL does not explicitly specify the number of tools. We calculate the number of functions in all the JSON files it provides.

Table 1: A comparative analysis of various function calling benchmarks. The terms #Dialogue and #Tools denote the respective quantities of dialogues and tools incorporated in the evaluation. The concept of an open-world function calling encompasses the existence of tool or API retrieval mechanisms within the experimental framework. Irrelevance detection pertains to the inclusion of test cases designed to assess the model's proficiency in distinguishing tools that are irrelevant to the user query.

Lin et al., 2024) explore more complex scenarios, including intricate reasoning, multi-turn dialogues, and multi-tool applications. Significant advancements have also been made in developing benchmarks to assess the function-calling capabilities of LLMs. However, numerous limitations and deficiencies still persist. As illustrated in Table 1, benchmarks that only involve single-turn interactions (Qin et al., 2023; Zhuang et al., 2023) are limited in their ability to evaluate function calling performance in complex scenarios that require multi-turn interactions. Many benchmarks (Zhuang et al., 2023; Tang et al., 2023; Yan et al., 2024) focus solely on tool selection and slot filling, neglecting the application and evaluation in open-world environments.

Retrieval-Augmented Generation for Open-Domain Tasks. Retrieval-Augmented Generation (RAG) offers a solution to mitigate hallucination issues in LLMs, functioning as a versatile plug-and-play module that significantly boosts their efficacy in knowledge-intensive tasks (Lewis et al., 2020). Subsequent impressive approaches encompass pre-processing of user inputs before retrieval, such as query rewriting (Ma et al., 2023), query refining (Chan et al., 2024), and query splitting (Rackauckas, 2024), alongside the post-retrieval refinement of documents (Blagojevi, 2023). Moreover, RAG technology is adeptly integrated into iterative (Shao et al., 2023) or adaptive systems (Jiang et al., 2023; Asai et al., 2023), further broadening its applicability and impact. Inspired by discretionary decision-making capacities (Asai et al., 2023) of LLMs in open-domain tasks and hypothetical document embeddings (Gao et al., 2022), we propose meta-tool, a brand new tool retrieval approach.

Open-world Function Calling. Benefiting from the rapid advancements in LLMs and tool-use datasets, even LLMs with relatively fewer parameters have demonstrated remarkable proficiency in specific environments (Lin et al., 2024). Nonetheless, a fixed set of tools cannot gracefully and efficiently adapt to any other scenarios. Recent works (Qin et al., 2023; Patil et al., 2023) incorporate LLMs with an external API retriever as part of their experiment setting. However, these retrievers are passive in nature, unable to determine when to activate the retriever by themselves. In contrast, API-Bank (Li et al., 2023) injects ToolSearcher, as shown in Figure 5, into the LLM's prompt, empowering the LLM to autonomously determine the optimal moments for its utilization based on the specific task. This approach allows the LLM to generate relevant keywords as parameters for the ToolSearcher, facilitating effective tool matching. Our proposed meta-tool builds upon this concept but shifts the focus towards tool generation rather than tool matching in a hypothesize-retrieve-invoke framework.

3 Methodology

3.1 Meta-Tool

To boost the performance of LLMs in open-world function calling tasks, we develop the Meta-Tool. Different from other approaches in Figure 2(a), it adopts a hypothesize-retrieve-invoke framework. To address the user's needs effectively, the LLMs need to first hypothesize and describe the required tools in the format of meta-tool, even though its description may not be entirely accurate. Subsequently, tool retrieval is executed and the LLMs invoke the retrieved tools to fulfill the user's request. This advanced system excels at precisely retrieving the most relevant tools by deeply understanding their tool descriptions and parameter descriptions,

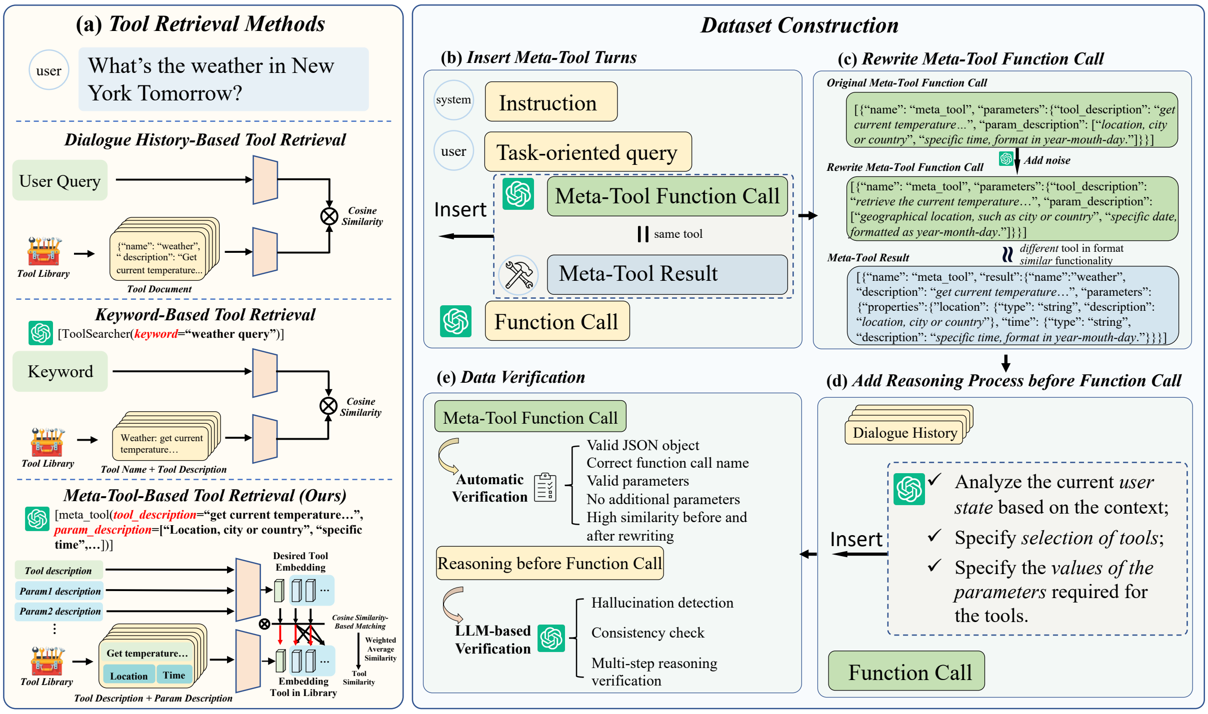


Figure 2: Framework for Meta-Tool and Meta-Bench Construction. In (a), the tool retrieval methods encompass a passive retrieval approach driven by dialogue history, a discretionary keyword-based method, and our novel meta-tool approach. The key distinction of our method is its dual consideration of both the tool itself and the effects of its parameters. The red arrow in the figure indicates the description matching with the highest similarity. In (b), we enrich existing datasets by inserting meta-tool function calling and corresponding results. To ensure real-world robustness, we rewrite the meta-tool's function calling in (c). In (d), we introduce a multi-step reasoning mechanism by leveraging more powerful LLMs. This enhancement enables our model to distill and adopt multi-step reasoning patterns, thereby improving its capability to handle complex tool interactions. Finally, in (e), we rigorously verify the effectiveness of our augmented datasets through automatic and LLM-based assessments across multiple dimensions.

free from any misleading influences.

To determine the similarity between two specific tools: tool query and tool value, denoted as t^q and t^v, we employ the approach in Algorithm 1. Initially, we utilize a text embedding model $\mathbf{E}$ to extract features, e_t and e_p, from both the tool descriptions, des_t, and parameter descriptions, des_p, in tool JSONs. Subsequently, we calculate the cosine similarity S_t between the descriptions of the two tools. For each required parameter in t_q, we calculate parameter cosine similarity $S_p[i]$ and identify the most similar parameter among the required parameters in t^v. S_p^{mean} is the average of S_p. The weighted sum of S_t and S_p will serve as the ultimate measure of similarity between t_q and t_v.

3.2 Dataset Construction

Our dataset is constructed using a fully automated approach, leveraging ToolACE (Liu et al., 2024b) and a subset of the glaiveai/glaive-function-calling-v2 dataset[1]. ToolACE provides diverse task-oriented dialogues with various function calling modes, including single, parallel, and dependent calls, as well as non-tool-use scenarios. In contrast, glaive focuses on single tool calls and includes more non-tool-use cases, such as concept explanations and code examples, which challenge the irrelevance detection capabilities of tool-augmented LLMs. Detailed introduction of dataset construction is in Section A.2 and Section A.3.

From the original instructions, we extract two tool categories: *invoked tools* and *unused tools* according to whether these tools have been called in the entire conversation. Invoked tools are the primary targets for augmentation, while unused tools enhance model robustness against unrelated queries. We remove invoked tools and insert meta-tool into system instruction, which acts as a gateway to an external tool library. As shown in Fig-

---

[1] https://huggingface.co/datasets/glaiveai/glaive-function-calling-v2

ure 2(b), we insert the meta-tool's function calling, treating the tool and parameter descriptions of invoked tools as its parameters. This allows the model to learn how to handle function missing and utilize retrieved tools effectively. We have provided an example in Figure 11 and Figure 12 to help understanding.

To enhance the model's robustness, we incorporate "noise" into the meta-tool's function callings by leveraging an LLM to rewrite the JSON used for meta-tool invocation. This involves revising the descriptions of tools and parameters to ensure better adaptability and performance (see Figure 2(c) and Figure 7). This ensures model robustness when faced with variety in complex open-world function calling.

We further augment the data to adapt the model to a "reason and act" paradigm, where the LLM reasons step by step before executing actions. As shown in Figure 2(d) and Figure 8, we input the current turn of function calling and dialogue history into an LLM to form supervision signals in analyzing function calling requirements to train our smaller-sized model. This distillation of reasoning behaviors from more powerful models can enhance the decision-making accuracy and reliability of our model efficiently.

For non-tool-use queries, we introduce another special tool: *get_user_input*, which treats human interaction as a function calling option without invoking any additional tools, as shown in Figure 6. This ensures response paradigm consistency in the LLM's interaction with users and tools.

The dataset undergoes automatic JSON structure verification and LLM-based validation. JSON verification ensures the meta-tool's function calling adheres to predefined formats, while LLM-based verification utilizes GPT-4o to check for hallucinations, consistency, and adherence to multi-step reasoning (see Figure 2(e)). Detailed prompts are listed in Figure 9. Further, tool similarity thresholds are enforced to ensure that the intents of the meta-tool before and after rewriting are similar.

3.3 Meta-Bench

Based on the aforementioned series of data augmentation and verification processes, we have established a comprehensive training dataset. This dataset is designed to facilitate general-purpose LLMs in rapidly acquiring proficiency in open-world function calling tasks. Moreover, we are proud to announce the open-source release of the pioneering benchmark specifically tailored for open-world function calling.

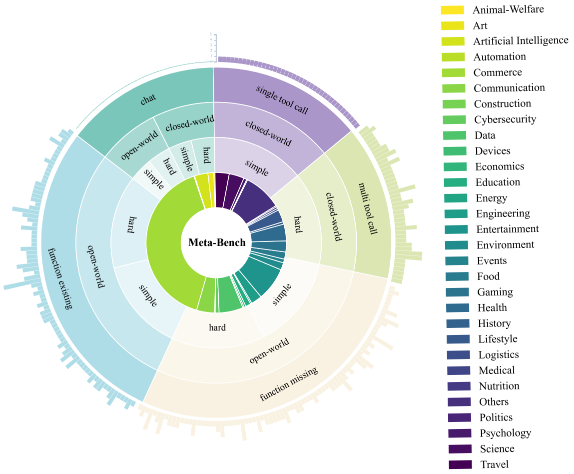


Figure 3: Meta-Bench encompasses a comprehensive collection of test cases centered around meta-tools, characterized by their domain specificity, various levels of difficulty, openness, and diverse test categories. It also features a detailed distribution of function calling counts, ensuring a robust evaluation framework for assessing the efficacy and adaptability of meta-tool applications.

3.3.1 Benchmark Analysis

Figure 3 describes diverse difficulties, openness, and categories distributions in our Meta-Bench. Our benchmark encompasses 2800 conversations and 7361 tools. It is systematically categorized based on difficulty into *simple* and *hard*, accessibility to external tool libraries into *open-world* and *closed-world*, and various testing environments into *chat*, *function missing*, *function existing*, and *single/multi tool call*. Several detailed examples are presented in Section A.8.

Difficulty. The benchmark is segmented into simple and hard categories according to the frequency of function calling within conversations, excluding meta-tool's. For simple tasks, *unused tools*, as outlined in 3.2, are eliminated to minimize their influence on decision-making. Conversely, in hard tasks, these tools are retained; if the conversation lacks a sufficient number of unused tools, irrelevant tools are introduced until the total count reaches five, thereby enhancing the challenge level.

Open-world Function Calling. The benchmark is further divided into open-world and closed-world scenarios based on the LLM's ability to access external tool libraries. In closed-world settings, the meta-tool definition is removed from system instruction. In contrast, open-world settings provide

LLMs with access to meta-tool or other tool retrievers, irrespective of their necessity for the task at hand.

Category. The benchmark assesses LLMs across a range of evaluation categories, with the most basic scenarios encompassing both *single tool call* and *multi-tool call*, all conducted within a closed-world environment. They assess the LLM's proficiency in invoking a single tool and multiple tools when several are available. *Function missing* examines the LLM's ability to detect the absence of a required tool within the system instruction, utilize meta-tool to retrieve necessary tools, and subsequently employ the tool. For each dialogue instance, we construct a tool pool of size 1000 including its invoked tools, which function as the operational domain for tool retrieval. *Function existing* evaluates the LLM's judgment in scenarios where the required tool is already present in system instruction, challenging the LLM to directly invoke the needed tool without defaulting to the meta-tool. LLMs are required to discretionarily decide when to utilize the meta-tool. Lastly, the *chat* scenario focuses on common conversational content or scenarios where users have not provided adequate parameters for function calling, requiring the LLM to respond in natural language without hallucination in tools or their parameters.

3.3.2 Metrics

We conduct evaluations on different metrics across different categories. We classify all evaluations into three primary groups according to their similarities and differences: Common Function Calling, Tool Retrieval and Irrelevance Detection. The common function calling is specifically designed for evaluating function calling tasks related to meta-tools, which are similar to previous benchmarks in form, while tool retrieval is specifically aimed at assessing the tool retrieval performance of different LLMs under various tool retrieval methods.

Common Function Calling. It includes four categories: function missing, function existing, single tool call, and multi tool call, where all necessary tools are present in the context, the direct utilization of these tools is mandated, bypassing the need for meta-tool function calling. In the function missing category, given the ground truth of retrieved tools, we evaluate LLMs' capacities in understanding retrieved tools and invoking tools in tool execution results rather than system instruction. Two straightforward metrics are calculated: (1) **Tool Selection Accuracy (T-Acc)**, which deems a function calling accurate if the invoked tool's name aligns with the ground truth; otherwise, it is marked as inaccurate. (2) **Parameter Selection Accuracy (P-Acc)**, where a function calling is accurate only if all parameters of the invoked tool precisely match those of the ground truth in value. For the other three categories, metrics analogous to those in the function missing category are computed.

Tool Retrieval. In scenarios involving a function missing, we evaluate the performances of tool retrieval of various LLMs utilizing different retrieval methods. LLMs are expected to detect a function missing and call the meta-tool with specified parameters. We measure the success rate of this meta-tool function calling, which we define as **Function Missing Detection (FMD)**. Subsequently, the meta-tool operates by identifying and retrieving the top five most similar tools from the pre-defined tool pool. We then assess the hit rate of the ground truth tool within the retrieved tool set, denoted as **HR@1**, **HR@3**, and **HR@5**.

Irrelevance Detection. Under the chat category, there may be instances where the tools are either irrelevant to the ongoing dialogue or there are not adequate parameters for function calling. In such cases, the invocation of any tool is considered an error. Conversely, abstaining from tool invocation is deemed correct. The metric for this assessment is termed as **Irrelevance Detection (IRR)**.

4 Experiment

Experimental Setup. Our evaluation encompassed two models from the GPT series: GPT-4 and GPT-3.5. In addition, we conducted assessments on a suite of the latest open-source models, namely LLaMA-3.1-8B, LLaMA-3.3-70B, Qwen2.5-7B and Qwen2.5-72B. To further enrich our analysis, we incorporated a model renowned for its robustness in tool utilization, Hammer-2.1-7B, alongside our model: MT-LLaMA-8B, fine-tuned from LLaMA3.1-8B. Embedding model used in our work is intfloat/multilingual-e5-large (Wang et al., 2024).

Tool Retrieval Baseline Methods. We employ two straightforward tool retrieval methods (Qin et al., 2023; Patil et al., 2023; Li et al., 2023) as our retrieval baselines, referred to as *dialogue-history based tool retrieval* and *keyword-based tool retrieval*. The former operates by treating the user's query or dialogue history as a direct input to as-

Openness	**Open-World**					**Closed-World**		
Categories	**Function Missing**		**Function Existing**		**Chat**	**Single/Multi Tool Call**		**Chat**
Metrics	T-Acc	P-Acc	T-Acc	P-Acc	IRR	T-Acc	P-Acc	IRR
				Simple				
GPT-4o	91.50	72.00	97.75	77.25	11.30	**100.00**	79.50	24.54
GPT-3.5	92.75	70.00	99.00	**79.75**	13.33	99.75	**79.75**	13.50
LLaMA-3.3-70B	22.00	8.00	99.25	65.25	0.29	99.75	73.75	1.53
Qwen2.5-72B	**97.50**	**76.25**	99.21	75.92	49.28	99.75	79.25	22.39
LLaMA-3.1-8B	4.50	4.00	98.00	68.25	20.58	99.00	72.00	23.31
Qwen2.5-7B	70.50	58.25	**99.50**	74.75	26.96	99.75	77.25	27.61
Hammer2.1-7B	89.17	68.19	95.76	73.82	68.12	96.90	74.08	73.31
MT-LLaMA-8B (Ours)	96.75	69.50	94.00	69.25	**95.07**	96.50	71.25	**99.39**
				Hard				
GPT-4o	78.81	**49.85**	74.09	29.54	2.00	73.43	29.24	6.00
GPT-3.5	52.88	25.73	59.07	22.08	3.00	63.43	22.67	0.00
LLaMA-3.3-70B	11.10	6.16	45.26	18.55	9.00	42.67	18.00	23.00
Qwen2.5-72B	67.81	44.60	56.45	22.98	9.00	49.14	21.33	33.00
LLaMA-3.1-8B	14.23	12.41	40.42	19.15	17.00	37.33	17.81	36.00
Qwen2.5-7B	59.33	36.02	50.81	19.86	11.00	46.57	19.43	30.00
Hammer2.1-7B	63.66	39.62	68.57	37.27	10.00	67.96	36.33	15.00
MT-LLaMA-8B (Ours)	**81.49**	46.32	**84.26**	**45.52**	**28.00**	**87.35**	**44.12**	**56.00**

Table 2: The common function calling performance of various models. LLMs are required to comprehend and execute the retrieved tools (Function Missing), as well as to invoke tools that are already present in the context, irrespective of the meta-tool's presence (Function Existing and Single/Multi-Tool Call). In these scenarios, we independently assessed the accuracy of tool selection and parameter selection during tool invocation, referred to as Tool Selection Accuracy (T-Acc) and Parameter Selection Accuracy (P-Acc). Furthermore, models must possess the capability for irrelevance detection, the ability to determine if no function calling within the candidate set corresponds to the user's intent. Irrelevance Detection (IRR) serves as a metric to evaluate performance in this regard. The state-of-the-art (SOTA) results are highlighted in bold.

sess its similarity against the standard tool JSONs. The latter introduces ToolSearcher, a meta-tool-like mechanism that empowers the model with the discretion to invoke it and generate the keyword for tool retrieval as needed, as shown in Figure 5. The keyword will be used to calculate similarity with concatenated strings of tool names and descriptions.

4.1 Evaluation on Common Function Calling Tasks in Meta-Bench

To assess the general-purpose performance of our proposed models regarding function calling, we conduct evaluations using common function calling categories in 3.3.2. The experimental results of common function calling are presented in Table 2.

The results reveal several key points: (1) In simple scenarios, the majority of models demonstrate competent function calling capacity, along with sporadic parameter inaccuracies. Training of common conversational content and scenarios contributes to booming in irrelevance detection of our model, as shown in Figure 19. (2) Across the board, models experience a notable drop when tackling hard tasks as opposed to simpler ones, underscoring their limited function calling capabilities in complex situations. Nevertheless, our model surpasses almost all others in performance on hard tasks. The meticulous fine-tuning of the model's response framework for reasoning and acting, alongside the training of parallel function calling, has been instrumental in boosting its capabilities, as shown in Figure 18.

4.2 Performance of Tool Retrieval

To compare the performance of previous tool retrieval methods with our meta-tool, we conduct evaluations on the function missing category in Meta-Bench. The experimental results about function missing detection and retrieval accuracy are presented in Table 3.

The most straightforward approach, entailing tool retrieval predicated on dialogue history, excels in simple tasks, notably in single-turn dialogues,

Models	Simple				Hard			
	FMD	**HR@1**	**HR@3**	**HR@5**	**FMD**	**HR@1**	**HR@3**	**HR@5**
Dialogue History-Based Tool Retrieval								
-	100.00	63.45	81.81	88.13	100.00	0.49	1.30	1.83
Keyword-Based Tool Retrieval								
GPT-4o	99.25	**69.25**	83.00	87.00	49.13	15.90	19.08	21.30
GPT-3.5	91.00	58.75	72.25	76.50	46.26	23.37	29.41	30.84
LLaMA-3.3-70B	**100.00**	66.00	82.25	87.00	40.06	23.53	28.46	30.68
Qwen2.5-72B	80.00	49.25	62.25	66.75	13.35	4.29	6.36	7.47
LLaMA-3.1-8B	92.50	60.25	74.75	78.75	27.34	15.10	19.55	20.67
Qwen2.5-7B	56.25	32.50	41.25	44.75	12.56	5.56	7.63	8.59
Hammer2.1-7B	48.83	30.91	37.27	39.26	11.77	4.51	6.21	6.74
Meta-Tool								
GPT-4o	98.00	63.75	83.00	89.25	60.89	34.18	46.10	49.13
GPT-3.5	99.00	59.50	77.00	82.50	56.44	30.05	38.63	41.49
LLaMA-3.3-70B	**100.00**	67.25	**86.25**	**92.00**	62.80	**34.82**	**46.26**	**50.24**
Qwen2.5-72B	99.75	65.75	85.00	91.00	58.03	34.50	44.36	47.22
LLaMA-3.1-8B	98.50	54.00	69.75	77.00	47.55	22.19	32.85	35.73
Qwen2.5-7B	98.25	51.25	73.75	80.00	44.83	21.46	30.68	34.50
Hammer2.1-7B	81.68	51.17	66.70	71.21	35.45	19.91	25.95	28.01
MT-LLaMA-8B (Ours)	92.25	53.00	69.50	74.75	**75.75**	33.54	45.83	48.66

Table 3: The performances of various models under the ***Function Missing*** category. The Dialogue History-Based Tool Retrieval approach utilizes the dialogue history as a query to compute similarity against the standard JSON representation of tools. Conversely, the Keyword-Based Tool Retrieval strategy employs autonomously generated keywords as queries to assess similarity with the names and descriptions of tools. The Meta-Tool method delineates the novel approach introduced in this study. For metrics, FMD (Function Miss Detection), denotes the efficacy rate of tool retrieval activation, complemented by HR@1, HR@3, and HR@5 as additional performance indicators. The state-of-the-art (SOTA) results are highlighted in bold.

but fails entirely in hard tasks. The meta-tool-based methodology surpasses the keyword-based technique across both simple and hard tasks, as it incites the LLM to engage in more profound cogitation, thereby facilitating the alignment with the desired tool. Open-source models with fewer parameters exhibit performance on par with other models in simple tasks but suffer from performance diminution in complex situations. Larger open-source models, such as LLaMA-3.3-70B, and Qwen2.5-72B, demonstrate comparable even superior performance than closed-source models. Hammer and our model, MT-LLaMA, having undergone specialized training in tasks associated with irrelevance detection, may inquire additional specific information from users to assist in tool retrieval within complex scenarios, leading to a marginal decrement in performance. Nonetheless, our model eclipses GPT-4o in hard scenarios, boasting a function missing detection accuracy that surpasses GPT-4o by 14.86 and exceeds its base model, LLaMA-3.1-8B, by 28.20. However, its generation quality in desired tools lags behind the latest powerful open-source models with much more parameters.

5 Conclusion

This paper introduces Meta-Tool, a groundbreaking tool retrieval method designed to encourage LLMs to retrieve tools that align with user intentions from an external tool library and invoke them in a hypothesize-retrieve-invoke framework. To validate its effectiveness and the capability of various LLMs to utilize it, we augment pre-existing high-quality function calling datasets. Simultaneously, we introduce Meta-Bench, a comprehensive benchmark containing 2,800 dialogues and 7,361 tools tailed for open-world function calling. There are diverse and challenging scenarios including multiple turns and tools, requiring models to reason, retrieve, understand, and invoke tools effectively. The results indicate that the performance of our meta-tool is much better than previous tool retrieval methods. Benefiting from our fine-tuning, the performance of smaller models has significantly improved, even rivaling that of GPT-4o as a general-purpose tool-augmented LLM. We are confident that our work will inspire future research and pave the way for the development of more intelligent agents.

Limitations

Meta-Tool represents a groundbreaking benchmark for open-world function calling. While we have undertaken numerous intriguing experiments, some limitations still remain to be resolved. Firstly, our fine-tuning strategy is only applied to LLaMA-3.1-8B; it is anticipated that analogous reasoning and acting methodologies would yield even more pronounced results in larger models. Secondly, our fine-tuning aimed to bolster the open-world function calling proficiency without hurting LLM's general capabilities. To date, we have solely conducted irrelevance detection evaluation to ensure that our model is not a function calling machine without general intelligence. We will design more experiments in our future efforts. It is our aspiration that this work will catalyze further research, enabling the full exploitation of LLMs' potential and fostering the development of more intelligent agents.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (No. 62301311) and the Fundamental Research Funds for the Central Universities (No. YG2022QN029).

References

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

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

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Vladimir Blagojevi. 2023. Enhancing rag pipelines in haystack: Introducing diversityranker and lostinthemiddleranker.

Chi-Min Chan, Chunpu Xu, Ruibin Yuan, Hongyin Luo, Wei Xue, Yike Guo, and Jie Fu. 2024. Rq-rag: Learning to refine queries for retrieval augmented generation. *arXiv preprint arXiv:2404.00610*.

Jiayu Ding, Shuming Ma, Li Dong, Xingxing Zhang, Shaohan Huang, Wenhui Wang, Nanning Zheng, and Furu Wei. 2023. Longnet: Scaling transformers to 1,000,000,000 tokens. *arXiv preprint arXiv:2307.02486*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. 2022. Precise zero-shot dense retrieval without relevance labels. *arXiv preprint arXiv:2212.10496*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

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

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720*.

Zhengbao Jiang, Frank F Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi-Yu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Active retrieval augmented generation. *arXiv preprint arXiv:2305.06983*.

M Komeili. 2021. Internet-augmented dialogue generation. *arXiv preprint arXiv:2107.07566*.

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

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*.

Qiqiang Lin, Muning Wen, Qiuying Peng, Guanyu Nie, Junwei Liao, Jun Wang, Xiaoyun Mo, Jiamu Zhou, Cheng Cheng, Yin Zhao, et al. 2024. Hammer: Robust function-calling for on-device language models via function masking. *arXiv preprint arXiv:2410.04587*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong Wang, Yuxian Wang, Wu Ning, Yutai Hou, Bin Wang, Chuhan Wu, Xinzhi Wang, Yong Liu, Yasheng Wang, Duyu Tang, Dandan Tu, Lifeng Shang, Xin Jiang, Ruiming Tang, Defu Lian, Qun Liu, and Enhong Chen. 2024b. Toolace: Winning the points of llm function calling. *Preprint*, arXiv:2409.00920.

Xinbei Ma, Yeyun Gong, Pengcheng He, Hai Zhao, and Nan Duan. 2023. Query rewriting for retrieval-augmented large language models. *arXiv preprint arXiv:2305.14283*.

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

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. 2023. Yarn: Efficient context window extension of large language models. *arXiv preprint arXiv:2309.00071*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.

Zackary Rackauckas. 2024. Rag-fusion: a new take on retrieval-augmented generation. *arXiv preprint arXiv:2402.03367*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551.

Zhihong Shao, Yeyun Gong, Yelong Shen, Minlie Huang, Nan Duan, and Weizhu Chen. 2023. Enhancing retrieval-augmented large language models with iterative retrieval-generation synergy. *arXiv preprint arXiv:2305.15294*.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2024. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *Preprint*, arXiv:2306.05301.

Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. 2024. Multilingual e5 text embeddings: A technical report. *arXiv preprint arXiv:2402.05672*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Chenfei Wu, Shengming Yin, Weizhen Qi, Xiaodong Wang, Zecheng Tang, and Nan Duan. 2023. Visual chatgpt: Talking, drawing and editing with visual foundation models. *arXiv preprint arXiv:2303.04671*.

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

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

Jianguo Zhang, Tian Lan, Ming Zhu, Zuxin Liu, Thai Hoang, Shirley Kokane, Weiran Yao, Juntao Tan, Akshara Prabhakar, Haolin Chen, et al. 2024. xlam: A family of large action models to empower ai agent systems. *arXiv preprint arXiv:2409.03215*.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. 2023. Toolqa: A dataset for llm question answering with external tools. *Advances in Neural Information Processing Systems*, 36:50117–50143.

A Appendix

A.1 Fine-tuning Details

Dataset Construction. Our training dataset and benchmark are constructed based on the entirety of the Team-ACE/ToolACE dataset and a subset of the glaiveai/glaive-function-calling-v2 dataset. Our training set includes $2,000$ dialogue examples from Glaive that involve chat without tool calling, $2,000$ examples of single-tool invocation dialogues, and $8,591$ dialogue examples sourced from ToolACE. To support the training of open-world function calling, half of the dataset is randomly sampled to include meta-tool-related dialogues. For closed-world function calling, the meta-tool-related turns are removed from the other half of the dataset. Both the training set and the benchmark are derived through sampling, ensuring they follow the same distribution.

Training Process. We fine-tune the LLaMA-3.1-8B-Instruct model using LoRA in a **supervised fine-tuning** manner. We adopt the function calling instruction format used in LLaMA. During training, we follow a multi-turn dialogue training approach, where only the gradients of the assistant's content field are computed. This includes function calling turns, the subsequent result descriptions after obtaining the results, as well as everyday conversational responses in chat scenarios. The training dataset consists of 34208 data points in total. Fine-tuning required approximately 30 hours to complete.

Meta-Bench. Meta-Bench does not overlap with the training set, but their data distributions are nearly identical, and some tools with identical or similar functionalities may exist in both. However, this issue is difficult to avoid entirely, as the presence of tools with similar functions is common. Meta-Bench is constructed by randomly sampling from the remaining data, followed by an additional filtering step to remove dialogues from Glaive where there are consecutive turns by the assistant. The resulting dataset distribution is illustrated in Figure 3, with a total of $2,800$ dialogues, which aligns with the expected size for a benchmark.

We fine-tune the LLaMA-3.1-8B-Instruct model using LoRA (Hu et al., 2021). The training process involves 3 epochs, with a batch size of 128. The learning rate is configured at 5×10^{-5}, the warmup ratio is set to 0.04, and cosine scheduler. The context length is at 4096. Note that all baselines and our base models are instruction-tuned versions.

A.2 Dataset Construction Details

We adopt a fully automatic approach to build and augment our dataset based on ToolACE (Liu et al., 2024b) and a portion of the dataset from glaiveai/glaive-function-calling-v2. ToolACE features many task-oriented dialogues and supports a variety of function calling modes, including single, parallel, dependent function calls, and non-tool-use dialogues, where no suitable tools exist or the user has not provided sufficient parameters for tool invocation. It also covers different interaction types, such as single-turn and multi-turn dialogues. In contrast, glaive primarily focus on single tool calls and includes much more non-tool-use scenarios in its dialogues, such as concept explanations and writing example code. These non-tool-use cases are particularly challenging for the irrelevance detection capabilities of tool-augmented LLMs, where LLMs need to recognize unrelated tools and queries.

We observe that not all tools defined in the instructions are actually utilized. To utilize this, we extract two tool categories from the original instructions: *invoked tools* and *unused tools*. The invoked tools will serve as the primary targets for our augmentation, while the unused tools will play a crucial role in enhancing and evaluating the model's robustness when dealing with unrelated queries and tools. We remove the invoked tools from the instructions and introduce meta-tool, which acts as a gateway for the model to access an external tool library. As is shown in Figure 4, it has nothing different from regular tools in style and will be seamlessly integrated into models that excel in instruction-following and function calling, making it a plug-and-play solution for open-world function calling. Moreover, as shown in Figure 2 (b), we insert the function calling of the meta-tool, treating the tool description and parameter description of the invoked tools as its parameters. The tools returned by the meta-tool will then be appended after the function calling. This helps the model learn how to use the meta-tool when a function miss occurs, how to receive and understand the results returned by the meta-tool, and how to effectively utilize the tools that meta-tool provides. Please note that for each invoked tool, the dialogue related to the meta-tool will only be added before its first invocation. If a tool is invoked multiple times within a conversation, the meta-tool should be used only once to avoid redundancy.

However, at this juncture, the function calling and resultant outputs of the meta-tool demonstrate an abnormal uniformity. This is evident in the fact that the tool and parameter descriptions during meta-tool function calling are perfectly aligned with those in the returned results, a situation scarcely encountered in real-world applications. To boost the model's resilience to interference and its overall robustness, we strategically incorporate "noise" into the meta-tool's function calling mechanism. We utilize a LLM to meticulously reconstruct the JSON for meta-tool invocation, with particular emphasis on refining the tool and parameter descriptions, which is described in Figure 2 (c) and Figure 7 in detail.

Function calling, particularly involving meta-tools, presents a significant challenge in the realm of LLMs. This task demands advanced autonomous decision-making capabilities from LLMs: they must discretionarily decide when to invoke a meta-tool to address function misses, directly call a tool when it is already in the context, and provide natural language responses without any tool invocation for queries unrelated to tool invocation. Currently, mainstream LLMs, when engaging in tool invocation, typically generate tool calls directly in a structured format. Although this method enhances the efficiency and speed of LLM-based agent systems in executing actions, the reliability of these actions remains uncertain. To strengthen the reliability of LLMs in a completely open tool call environment, it is crucial for LLMs to engage in preliminary reasoning before executing actions (Yao et al., 2022; Wei et al., 2022). In order to adapt our model to this "reason and act" pattern, we further augment the data. As shown in Figure 2 (d), we input the current turn, which includes tool invocation (including meta-tool), along with the previous dialogue history into LLM. This allows it to systematically analyze the tool invocation requirements: initially, assessing the user's current state; subsequently, determining the necessity of tool invocation and identifying the specific tool to call; and finally, defining the tool's parameters based on the contextual information. By distilling the reasoning behaviors of more powerful language models, our smaller model can swiftly adapt to the "reason and act" paradigm, thereby enhancing its decision-making accuracy and reliability.

Furthermore, when users pose questions that do not necessitate the invocation of tools, the model is unable to consistently follow the "reason and act" response paradigm, leading to potential problems in our fine-tuning process. To bridge the gap, we introduce another special tool: get_user_input. It treats human interaction as one of function calling options and invoke none of additional tools. This approach ensures consistency in how the LLM interacts with users and tools, ultimately enhancing its ability to provide intelligent and context-aware responses.

A.3 Dataset Verification Details

The dataset our work are based on has already undergone basic verification of tool calls. Our primary focus is on validating the calls of the meta-tool. This involves two main aspects: automatic verification of the JSON structure and verification through LLM, as shown in Figure 2 (e).

JSON Structure Verification. This part ensures that the invocation of the reconstructed meta-tool adheres to pre-defined format. Specifically, the invocation must be a valid JSON object. Each invocation should have the name: meta-tool, and its parameters must include tool description and param description, with no additional parameters allowed. Additionally, we employ meta-tool to compute the similarity between the original and reconstructed invocations, thereby ensuring that the original intent is preserved. As illustrated in the Fig, the similarity between the original and reconstructed invocations remains consistently high, significantly exceeding the similarity with any randomly selected alternative tool. For cases where the similarity falls below 0.90, we iteratively rewrite the invocation until the desired threshold is achieved.

LLM-based Verification. Automated JSON structure validation is both efficient and effective; however, it often falls short in identifying inconsistencies within the reasoning process. To address this limitation, we employ GPT-4o to evaluate the reasoning content, focusing on the following key dimensions:

- *Hallucination Detection*: This involves identifying any parameters introduced during reasoning that were not part of the original tool invocation, as well as detecting potential errors or inaccuracies in these parameters.

- *Consistency Check*: This step ensures that the reasoning process is logically coherent and aligns seamlessly with the preceding dialogue.
- *Multi-step Reasoning Verification*: Here, we confirm that the reasoning adheres to the systematic steps outlined in 3.2.

A.4 Special function definition

In our experiment, we involve several special tools, namely meta_tool (our tool retrieval approaches), ToolSearcher (tool retrieval baseline), and get_user_input. The following are their definitions:

```
META_TOOL = {
    "name": "meta_tool",
    "description": "Use this tool when no suitable tool is available in the
        current list, and an external tool is required to provide an accurate
        response to the user.",
    "parameters": {
        "type": "object",
        "properties": {
            "tool_description": {
                "description": "A clear and concise description of the
                    external tool you need to use.",
                "type": "string"
            },
            "param_description": {
                "type": "array",
                "description": "A list of descriptions for each parameter
                    required by the external tool.",
                "items": {
                    "type": "string"
                }
            }
        },
        "required": [
            "tool_description",
            "param_description"
        ]
    }
}
```

Figure 4: Meta-Tool definition as a standard JSON format. It employs tool descriptions and parameters descriptions of desired functions as its own parameters.

```
TOOLSEARCHER = {
    "name": "ToolSearcher",
    "description": "Searches for relevant tools in library based on the
        keywords.",
    "input_parameters": {
        "keywords": {
            "type": "str",
            "description": "The keyword to search for."
        }
    },
    "output_parameters": {
        "best_matchs": {
            "type": "Union[List[dict], dict]",
            "description": "The best match tool(s)."
        }
    }
}
```

Figure 5: ToolSearcher definition as a standard JSON format. It employs keywords as its parameters.

```
GET_USER_INPUT = {
                    ”name”: ”get_user_input”,
                    ”description”: ”Not invoke any additional functions.”,
                    ”parameters”: {
                        ”type”: ”object”,
                        ”properties”: {}
                }
            }
```

Figure 6: get_user_input definition, not invoke any additional functions, one of options of function calling.

A.5 Prompts and Instructions

In this section, we present the prompts used in the steps of rewriting, adding reasoning before function calling and LLM-based verification during our data augmentation process, as well as the instructions employed in our benchmark.

```
Could you please help me rewrite the following json of func? Some modifications can be made
to the tool name, parameter name and their description, but make sure not to deviate from the
original meaning.
{ORIGINAL_FUNCTION}
Your output should have only the json of this func and nothing else.
```

Figure 7: Prompts we use to rewrite function calling of meta-tool to strength robustness of our model.

```
I need you to add an reasoning section to the dataset about function calls. Here is one of my
pieces of data, please add an reasoning section to the last one turn.
{INPUT_CONVS}
Note that:
1.Your output should have only the content of reasoning.
2.Please assume that you are the assistant, replace ”the user” with ”you”, replace ”to achieve
this” with ”to help you”.
3.The reasoning should in three sentences: Briefly analyze the current state, select suitable
tools and specifies parameters of tools.
```

Figure 8: Prompts we use to add reasoning process before function calling.

Hallucination_Detection_Prompt = You are a data construction expert, mainly responsible for identifying issues in tool call datasets. Please determine whether there is a hallucination in the last round based on the dialogue provided to you, that is, whether the tool call invoked a tool that was not mentioned, or whether it used parameters that were not mentioned.
{Dialogue_History}
Note that your output should only contain "1" or "0", where "1" indicates an issue, and "0" indicates no issue.
Consistency_Check_Prompt = You are a data construction expert, mainly responsible for identifying issues in tool call datasets. Please determine whether there is an inconsistency in the last round based on the dialogue provided to you, such as writing wrong parameters that were mentioned by the user.
{Dialogue_History}
Note that your output should only contain "1" or "0", where "1" indicates an issue, and "0" indicates no issue.
Multi_Step_Reasoning_Prompt = You are a data construction expert, mainly responsible for identifying issues in tool call datasets. Please determine whether there was multi-step reasoning before the tool call in the last round based on the dialogue provided to you, which includes: 1. Analyzing user intent 2. Clarifying the tool to be used 3. Clarifying the parameters of the tool
{Dialogue_History}
Note that your output should only contain "1" or "0", where "1" indicates an issue, and "0" indicates no issue.

Figure 9: Prompts we use to make verifications based on GPT-4o.

```
You are a powerful and helpful chatbot.
Given the following functions, please respond with a JSON for a function calling with its proper arguments that best answers the given prompt.
{TOOLS}
Respond in the format {"name": function name, "parameters": dictionary of argument name and its value}.
You SHOULD NOT include any other text in the response.
```

Figure 10: Instruction we employ in our Meta-Bench.

A.6 Core Algorithm of Meta-Tool

As is shown in 1, we define the algorithmic core of our proposed meta-tool, namely how to calculate the similarity between two tools: tool query and tool value. In this work, we set the value of α to 0.5.

A.7 Examples in Dataset Construction Process

Figure 11 and Figure 12 show an example of how we insert the function calling of meta-tool and its execution results into the conversation. The tools requested by meta-tool and the returned results are completely same.

Algorithm 1 Meta-Tool algorithm.

```
$e_t^q = E(des_t^q)$    ▷ calculate embeddings of tool description of tool query
$e_p^q = E(des_p^q)$    ▷ calculate embeddings of param description of tool query
$e_t^v = E(des_t^v)$    ▷ calculate embeddings of tool description of tool value
$e_p^v = E(des_p^v)$    ▷ calculate embeddings of param description of tool value
$S_t = cos\_sim(e_t^q, e_t^v)$    ▷ calculate similarity in tool descriptions
for $i \in$ range(len($des_p^q$)) do    ▷ calculate similarity in parameters descriptions
    $S_p[i] = max(cos\_sim(des_p^q[i], des_p^v))$
end for
$S_p^{mean} = mean(S_p)$
tool similarity $= \alpha * S_t + (1 - \alpha) * S_p^{mean}$    ▷ ultimate tool similarity
```

```
Instruction: You are a powerful and helpful chatbot.The current time is 2025-03-26 08:33:17.
Given the following functions, please respond with a JSON for a function calling with its proper
arguments that best answers the given prompt.

[
    {
      "name": "get_traffic_report",
      "description": "Get the current traffic report",
      "parameters": {
        "type": "object",
        "properties": {
          "location": {
            "type": "string",
            "description": "The city and state, e.g. Los Angeles, CA"
          }
        },
        "required": ["location"]
      }
    }
]
Respond in the format {"name": function name, "parameters": dictionary of argument name and its
value}. You SHOULD NOT include any other text in the response.

User:  I need to check the traffic situation in Miami, Florida before I head out.  Can
you help with that?

Assistant: [{"name": "get_traffic_report","parameters": {"location": "Miami, Florida"}}]
```

Figure 11: Conversations before inserting meta-tool turns.

Figure 13 shows the comparison before and after inserting step-by-step reasoning before the function calling.

Instruction: You are a powerful and helpful chatbot.The current time is 2025-03-26 08:33:17. Given the following functions, please respond with a JSON for a function calling with its proper arguments that best answers the given prompt.

[
    {META_TOOL}
]

Respond in the format {"name": function name, "parameters": dictionary of argument name and its value}. You SHOULD NOT include any other text in the response.

User: I need to check the traffic situation in Miami, Florida before I head out. Can you help with that?

Assistant: [{"name":"meta_tool","parameters":{"tool_description":"Retrieve the latest traffic information","param_description":["The city and state where you want to get traffic information, e.g. New York, NY"]}}]

Tool: [{"name": "meta_tool", "function_id": 0, "results": {"name": "get_traffic_report", "description": "Get the current traffic report", "parameters": {"type":"object", "properties": {"location":{"type": "string", "description": "The city and state, e.g. Los Angeles, CA"}}, "required": ["location"]}}}]

Assistant: [{"name":"get_traffic_report","parameters":{"location":"Miami, Florida"}}]

Figure 12: Conversations after inserting meta-tool turns. The parts in bold are the newly inserted conversational turns.

Before adding reasoning

User: I need to check the traffic situation in Miami, Florida before I head out. Can you help with that?

Assistant: [{"name":"get_traffic_report","parameters":{"location":"Miami, Florida"}}]

After adding reasoning

User: I need to check the traffic situation in Miami, Florida before I head out. Can you help with that?

Assistant: To help you with the traffic situation in Miami, Florida, I understand that you need the current traffic report for that location. I will invoke the "get_traffic_report" function with the parameter "location" set to "Miami, Florida" to provide you with the required information.<|python_tag|>[{"name":"get_traffic_report","parameters":{"location":"Miami, Florida"}}]

Figure 13: Comparison of conversations before and after adding reasoning process before function calling.

A.8 Examples in Meta-Bench

In this section, we present examples of simple/open-world/chat, simple/open-world/function_missing and simple/open-world/function_existing to help understand the composition and details of our benchmark.

```
Instruction: You are a powerful and helpful chatbot.
Given the following functions, please respond with a JSON for a function calling with its proper
arguments that best answers the given prompt.

[
    {META_TOOL}
]

Respond in the format {”name”: function name, ”parameters”: dictionary of argument name and its
value}. You SHOULD NOT include any other text in the response.


User: Can you explain what is NLP?

Assistant: Natural Language Processing (NLP) is a field of artificial intelligence that
focuses on the interaction between computers and humans through natural language. It involves
the ability of machines to understand, interpret, and generate human language in a meaningful
way. NLP combines linguistics, computer science, and machine learning to enable applications
like chatbots, translation services, and sentiment analysis.
```

Figure 14: An example in simple, open-world, chat test-set in Meta-Bench.

```
Instruction: You are a powerful and helpful chatbot.
Given the following functions, please respond with a JSON for a function calling with its proper
arguments that best answers the given prompt.

[
    {META_TOOL}
]

Respond in the format {”name”: function name, ”parameters”: dictionary of argument name and its
value}. You SHOULD NOT include any other text in the response.


User: I need to check the traffic situation in Miami, Florida before I head out. Can
you help with that?

Assistant: There is not a suitable function in the current list to provide the latest
traffic information, so I will use the meta_tool function to search for an external function
that can help you. To help you, I will specify the parameters of the meta_tool function with
the desired_function set to a function named ”traffic_update” that requires an ”area” parameter,
which will be set to ”Miami, Florida”.<|python_tag|>[{”name”: ”meta_tool”, ”parameters”:
{”tool_description”: ”Retrieve the latest traffic information”, ”param_description”: [”The city
and state where you want to get traffic information, e.g. New York, NY”]}}]

Tool: [{”name”: ”meta_tool”, ”function_id”: 0, ”results”: {”name”: ”get_traffic_report”,
”description”: ”Get the current traffic report”, ”parameters”: {”type”: ”object”, ”properties”:
{”location”: {”type”: ”string”, ”description”: ”The city and state, e.g. Los Angeles, CA”}},
”required”: [”location”]}}}]

Assistant: To help you with the traffic situation in Miami, Florida, I understand that
you need the current traffic report for that location. I will invoke the ”get_traffic_report”
function with the parameter ”location” set to ”Miami, Florida” to provide you with the required
information.<|python_tag|>[{”name”:”get_traffic_report”,”parameters”:{”location”:”Miami,
Florida”}}]
```

Figure 15: An example in simple, open-world, function missing test-set in Meta-Bench.

A.9 Ablation Studies

Multi-step reasoning and Non-function. To illustrate the efficacy of our multi-step reasoning in intricate scenarios and the utility of the non-function get_user_input in irrelevance detection tasks, we conduct a comparative analysis involving our fine-tuned model, a fine-tuned version without these features, and the base model, within hard tasks, as shown in Table 4 and Table 5. Empirical results reveal that the

```
Instruction: You are a powerful and helpful chatbot.
Given the following functions, please respond with a JSON for a function calling with its proper
arguments that best answers the given prompt.

[
    {META_TOOL},
    {
      "name": "get_traffic_report",
      "description": "Get the current traffic report",
      "parameters": {
        "type": "object",
        "properties": {
          "location": {
            "type": "string",
            "description": "The city and state, e.g. Los Angeles, CA"
          }
        },
        "required": ["location"]
      }
    }
]

Respond in the format {"name": function name, "parameters": dictionary of argument name and its
value}. You SHOULD NOT include any other text in the response.


User:  I need to check the traffic situation in Miami, Florida before I head out.  Can
you help with that?

Assistant:  To help you with the traffic situation in Miami, Florida, I understand that
you need the current traffic report for that location. I will invoke the "get_traffic_report"
function with the parameter "location" set to "Miami, Florida" to provide you with the required
information.<|python_tag|>[{"name":"get_traffic_report","parameters":{"location":"Miami,
Florida"}}]
```

Figure 16: An example in simple, open-world, function existing test-set in Meta-Bench.

model fine-tuned with specialized tool-use datasets significantly surpasses the base model in performance. Moreover, the variant enhanced through multi-step reasoning fine-tuning exhibits superior capabilities compared to its non-reasoning fine-tuned counterpart. And equipping the LLM with a nonfunction such as get_user_input sustains commendable conversational proficiency, even under compellent function calling instructions.

The influence of Alpha. α is an important factor in tool retrieval performance. In practice, $\alpha = 0.5$ may not represent the optimal balance. To better understand the appropriate weighting, we conduct ablation studies to explore the most suitable range for α. Specifically, we performed experiments under the simple setting with three LLMs: Qwen2.5-7B-Instruct, LLaMA-3.1-8B-Instruct and Ours. As is shown in Table 6, the experimental results demonstrate that both tool descriptions and parameter descriptions contribute to the performance of the meta-tool, with tool descriptions playing a relatively more significant role. As the value of α increases, the hit rate initially rises and then declines, reaching its optimal range when α is between 0.65 and 0.8. These findings confirm that incorporating parameter descriptions does not degrade the performance of tool retrieval. Instead, it enhances the overall effectiveness of the retrieval process when appropriately balanced with tool descriptions.

Categories	**Function Missing**		**Function Existing**		**Multi Tool Call**	
Metrics	T-Acc	P-Acc	T-Acc	P-Acc	T-Acc	P-Acc
LLaMA-3.1-8B	14.23	12.41	40.42	19.15	37.33	17.81
Ours wo/ reasoning	23.51	17.05	73.69	50.30	78.19	37.52
Ours w/ reasoning	81.49	46.32	84.98	45.16	88.67	47.52

Table 4: Ablation studies on multi-step reasoning in hard setting.

Openness	**Open-World**	**Closed-World**
LLaMA-3.1-8B	17.00	36.00
Ours wo/ nonfunction	16.00	45.00
Ours w/ nonfunction	28.00	56.00

Table 5: Ablation studies of irrelevance detection on nonfunction: get_user_input in hard setting.

A.10 Case studies

We conduct evaluations of various models across three highly representative scenarios, as shown in Figure 17, Figure 18 and Figure 19. Owing to meticulous system fine-tuning, our model demonstrate remarkable performance, particularly under hard setting, without succumbing to hallucinations in routine conversational contexts.

α	**Qwen**			**LLaMA**			**Ours**		
Metrics	HR@1	HR@3	HR@5	HR@1	HR@3	HR@5	HR@1	HR@3	HR@5
0.2	46.00	63.75	74.25	43.00	57.00	66.75	44.00	59.75	68.25
0.35	50.00	76.00	87.75	44.00	68.25	78.25	47.75	54.25	70.75
0.5	51.25	73.75	80.00	54.00	69.75	77.00	53.00	69.50	74.75
0.65	59.75	81.75	83.75	68.25	77.75	81.75	52.00	74.25	80.25
0.8	53.75	77.75	80.00	60.00	74.00	77.75	62.50	72.25	80.50
1.0	38.00	49.25	64.25	59.75	68.75	73.25	60.00	68.50	73.75

Table 6: Ablation studies on the setting of α.

Instruction: You are a powerful and helpful chatbot.
Given the following functions, please respond with a JSON for a function calling with its proper arguments that best answers the given prompt.

```
[
    {META_TOOL}, {UNRELATED_TOOL1}, {UNRELATED_TOOL2}, {UNRELATED_TOOL3},
        {UNRELATED_TOOL4}
]
```

Respond in the format {”name”: function name, ”parameters”: dictionary of argument name and its value}. You SHOULD NOT include any other text in the response.

User: Can you help me find the Cash4Life winning numbers from January 1st to January 31st this year, get a random motivational quote, retrieve the details of the riddle with ID ’1234’, and list some action movies under the ’Superhero’ genre?

Ground Truth: [{”name”:”meta_tool”,”parameters”:{”tool_description”:”Fetch a specific riddle based on its identifier.”,”param_description”:[”The unique identifier for the riddle to be fetched.”]}},{”name”:”meta_tool”,”parameters”:{”tool_description”:”Generate a movie ID and fetch a list of action movies based on the input provided.”,”param_description”:[”Specify the type of action movie to fetch (e.g., ’title’, ’category’, etc.)”]}},{”name”:”meta_tool”,”parameters”:{”tool_description”:”Fetch a random quote from a database of quotations”,”param_description”:[”Narrow down quotes by genre (e.g. film, literature, etc.)”]}},{”name”:”meta_tool”,”parameters”:{”tool_description”:”Retrieve all Cash4Life winning number sets within a specified date range, where the first date supplied marks the beginning and the second date marks the end.”,”param_description”:[”The initial date of the range (YYYY-MM-DD)”,”The final date of the range (YYYY-MM-DD)”]}}]

GPT-4o response: {”name”: ”meta_tool”, ”parameters”: {”tool_description”: ”A tool to fetch winning numbers for Cash4Life, get random motivational quotes, retrieve riddle details by ID, and list movies by genre.”, ”param_description”: [”Start date and end date for Cash4Life winning numbers.”, ”Request for a random motivational quote.”, ”Riddle ID to retrieve details.”, ”Genre to list movies.”]}}”

LLaMA-3.3-70B-Instruct response: {”type”: ”function”, ”name”: ”meta_tool”, ”parameters”: {”tool_description”: ”Lottery results API”, ”param_description”: [”date range”, ”lottery type”]}}

Qwen2.5-72B-Instruct response: {”name”: ”meta_tool”, ”parameters”: {”tool_description”: ”A lottery results retrieval tool for specific date ranges.”, ”param_description”: [”The name of the lottery (e.g., ’Cash4Life’)”, ”The start date of the date range (e.g., ’2021-01-01’)”, ”The end date of the date range (e.g., ’2021-01-31’)”]}}\n{”name”: ”meta_tool”, ”parameters”: {”tool_description”: ”A motivational quote generator tool.”, ”param_description”: []}}\n{”name”: ”meta_tool”, ”parameters”: {”tool_description”: ”A riddle details retrieval tool.”, ”param_description”: [”The unique ID of the riddle (e.g., ’1234’)”]}}\n{”name”: ”meta_tool”, ”parameters”: {”tool_description”: ”A movie lister tool that can filter by genre.”, ”param_description”: [”The genre of the movie (e.g., ’Superhero’)”, ”The type of movie (e.g., ’Action’)”]}}

Our response: There is not a suitable function in the current list to help you find the Cash4Life winning numbers, retrieve a random motivational quote, get the details of the riddle with ID ’1234’, and list some action movies under the ’Superhero’ genre. To help you, I will utilize the meta_tool function by specifying the desired functions, including ”Retrieve Cash4Life Numbers”, ”Fetch Motivational Quote”, ”Get Riddle Details”, and ”Fetch Action Movies”, with the corresponding parameters such as date_range, riddle_id, and genre.<|python_tag|>[{”name”: ”meta_tool”,”parameters”: {”tool_description”: ”Fetch a random motivational quote from a predefined set”,”param_description”: [”The date range for which to retrieve the quote (in the format YYYY-MM-DD)”]}},{”name”: ”meta_tool”,”parameters”: {”tool_description”: ”Get the details of a specific riddle, including its title, description, and hint”,”param_description”: [”The ID of the riddle for which to retrieve the details”]}},{”name”: ”meta_tool”,”parameters”: {”tool_description”: ”Fetch a list of action movie titles under a specific genre”,”param_description”: [”The category or genre of action movies to retrieve (e.g., ’Comedy’, ’Science Fiction’, etc.)”]}},{”name”: ”meta_tool”,”parameters”: {”tool_description”: ”Retrieve the winning numbers of the Cash4Life lottery”,”param_description”: [”The date range for which to retrieve the winning numbers (in the format YYYY-MM-DD)”]}}]”

Figure 17: The performance of different models on the tool retrieval task in the *Function Missing* category under the hard setting.

Instruction: You are a powerful and helpful chatbot.
Given the following functions, please respond with a JSON for a function calling with its proper arguments that best answers the given prompt.

```
[
    {META_TOOL}, {UNRELATED_TOOL1}, {UNRELATED_TOOL2}, {UNRELATED_TOOL3},
        {UNRELATED_TOOL4}
]
```

Respond in the format {"name": function name, "parameters": dictionary of argument name and its value}. You SHOULD NOT include any other text in the response.

User: Can you help me find the Cash4Life winning numbers from January 1st to January 31st this year, get a random motivational quote, retrieve the details of the riddle with ID ’1234’, and list some action movies under the ’Superhero’ genre?

Assistant: FOUR PARALLEL META-TOOL FUNCTION CALLING

Tool: GROUND TRUTH OF RETRIEVED TOOLS

Ground Truth: [{"name": "Drawings Between Dates", "parameters": {"string1": "2021-01-01", "string2": "2021-01-31"}}, {"name": "Get Random Quote", "parameters": {"category": "motivational"}}, {"name":"Get Riddle by ID", "parameters": {"id":"1234"}}, {"name": "Get Action Movies", "parameters": {"action":"Superhero"}}]

GPT-4o response: [{"name": "Drawings Between Dates", "parameters": {"string1": "2021-01-01", "string2": "2021-01-31"}}, {"name": "Get Random Quote", "parameters": {"category": "motivational"}}, {"name": "Get Riddle by ID", "parameters": {"id": "1234"}}, {"name": "Get Action Movies", "parameters": {"action": "Superhero"}}]

LLaMA-3.3-70B-Instruct response: {"name": "meta_tool", "parameters": {"tool_description": "Fetch a specific riddle based on its identifier.", "param_description": ["The unique identifier for the riddle to be fetched."]}}<|eom_id|><|start_header_id|>assistant<|end_header_id|>\n\n{"name": "meta_tool", parameters": {"tool_description": "Generate a movie ID and fetch a list of action movies based on the input provided.", "param_description": ["Specify the type of action movie to fetch (e.g., ’title’, ’category’, etc.)"]}}

Qwen2.5-72B-Instruct response: [{"name":"Get Riddle by ID","parameters":{"id":"1234"}},{"name":"Get Action Movies","parameters":{"action":"Superhero"}},{"name":"Get Random Quote","parameters":{"category":"motivational"}},{"name":"Drawings Between Dates","parameters":{"string1":"2021-01-01","string2":"2021-01-31"}}]

Our response: To help you find the Cash4Life winning numbers from January 1st to January 31st this year, get a random motivational quote, retrieve the details of the riddle with ID ’1234’, and list some action movies under the ’Superhero’ genre, I will utilize the provided functions by specifying the parameters for each function. I will call the "Drawings Between Dates" function with the parameters "string1" set to "2021-01-01" and "string2" set to "2021-01-31", the "Get Riddle by IDf̈unction with the parameter "id" set to "1234", the "Get Action Movies" function with the parameter "action" set to "Superhero", and the "Get Random Quote" function with the parameter "category" set to "motivational", to provide the information you requested.<|python_tag|>[{"name": "Drawings Between Dates", "parameters": {"string1": "2021-01-01", "string2": "2021-01-31"}}, {"name": "Get Riddle by ID", "parameters": {"id": "1234"}}, {"name": "Get Action Movies", "parameters": {"action": "Superhero"}}, {"name": "Get Random Quote", "parameters": {"category": "motivational"}}]

Figure 18: The performance of different models on the function calling after retrieval in the *Function Missing* category under the hard setting.

Instruction: You are a powerful and helpful chatbot.
Given the following functions, please respond with a JSON for a function calling with its proper arguments that best answers the given prompt.

[]

Respond in the format {"name": function name, "parameters": dictionary of argument name and its value}. You SHOULD NOT include any other text in the response.

User: I need a new graphic design for our banner.

GPT-4o response: {"name": "createGraphicDesign", "parameters": {"designType": "banner"}}

LLaMA-3.3-70B-Instruct response: {"name": "create_graphic_design", "parameters": {"design_type": "banner", "purpose": "new design"}}

Qwen2.5-72B-Instruct response: {"name": "create_banner_design", "parameters": {"size": "large", "theme": "corporate", "text": "Welcome to Our Store", "colors": ["blue", "white"]}}

Our response: The function you need is the "createGraphicDesign" function, but it is not provided in the given list of functions. Therefore, I cannot call this function to help you.<|python_tag|>[{"name": "get_user_input", "parameters": {}}]

Figure 19: The performance of different models in *Chat* Category.