TOOLLLM: ENHANCING LARGE LANGUAGE MODELS TO MASTER 16000+ REAL-WORLD APIS

Yujia Qin[1*], **Shihao Liang**[1,2*], **Yining Ye**[1], **Kunlun Zhu**[1,2], **Lan Yan**[1,2], **Yaxi Lu**[1], **Yankai Lin**[3†], **Xin Cong**[1], **Xiangru Tang**[4], **Bill Qian**[4], **Sihan Zhao**[1], **Runchu Tian**[1], **Ruobing Xie**[5], **Jie Zhou**[5], **Mark Gerstein**[4], **Dahai Li**[2,6], **Zhiyuan Liu**[1†], **Maosong Sun**[1†]
[1]Tsinghua University [2]ModelBest Inc. [3]Renmin University of China
[4]Yale University [5]WeChat AI, Tencent Inc. [6]Zhihu Inc.
yujiaqin16@gmail.com

ABSTRACT

Despite the advancements of open-source large language models (LLMs) and their variants, e.g., LLaMA (Touvron et al., 2023) and Vicuna (Chiang et al., 2023), they remain significantly limited in performing higher-level tasks, such as following human instructions to use external tools (APIs). This is because current instruction tuning largely focuses on basic language tasks instead of the tool-use domain. This is in contrast to state-of-the-art (SOTA) LLMs, e.g., ChatGPT (OpenAI, 2022), which have demonstrated excellent tool-use capabilities but are unfortunately closed source. To elicit such capabilities within open-source LLMs, we present ToolBench, an instruction-tuning dataset for tool use. ToolBench is constructed automatically using ChatGPT through three phases: collecting a plethora of real-world APIs, generating diverse human-like instructions involving these APIs, and annotating high-quality responses. Notably, ToolBench encompasses $16,464$ real-world RESTful APIs spanning 49 categories, coupled with realistic human instructions involving both single-tool and multi-tool scenarios. To enhance model reasoning and facilitate more efficient answer annotation, we develop a novel depth-first search-based decision tree (DFSDT), enabling LLMs to evaluate multiple reasoning traces. When fine-tuned on ToolBench, LLaMA evolves into ToolLLaMA, demonstrating a remarkable ability to execute complex instructions and generalize to unseen APIs. We also introduce an automatic evaluator, ToolEval, for efficient performance evaluation. We find that ToolLLaMA exhibits comparable performance to `gpt-3.5-turbo-16k`. Moreover, we devise a neural API retriever to recommend appropriate APIs for each instruction, negating the need for manual API selection. All the codes and trained models are publicly available at `https://github.com/OpenBMB/ToolBench`.

1 INTRODUCTION

Tool learning (Qin et al., 2023b) aims to unleash the power of large language models (LLMs) to effectively interact with various tools (APIs) to accomplish complex tasks. By integrating LLMs with APIs, we can greatly expand their utility and empower them to serve as efficient intermediaries between users and the vast ecosystem of applications. Although open-source LLMs, e.g., LLaMA (Touvron et al., 2023), have achieved versatile capabilities through instruction tuning (Taori et al., 2023; Chiang et al., 2023; Ding et al., 2023), they still lack the sophistication in performing higher-level tasks, such as understanding human instructions and appropriately interacting with tools (APIs). This is because current instruction tuning largely focuses on basic language tasks (e.g., general conversation), instead of the tool-use domain. On the other hand, current state-of-the-art (SOTA) LLMs (e.g., ChatGPT (OpenAI, 2022) and GPT-4 (OpenAI, 2023)), which have demonstrated impressive competencies in skillfully utilizing tools (Qin et al., 2023b; Bubeck et al., 2023), are closed-source

[*] Indicates equal contribution.
[†] Corresponding author.

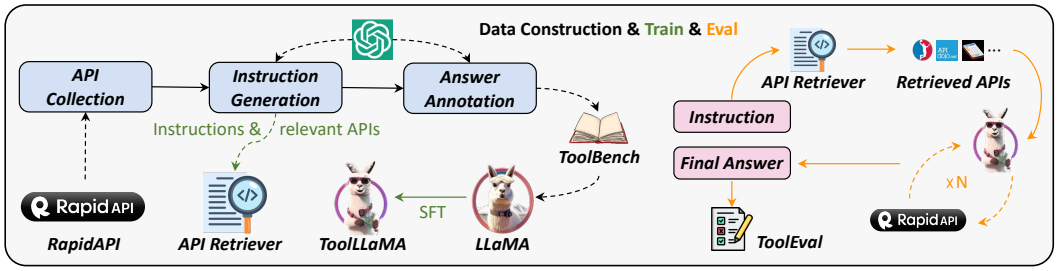

Figure 1: Three phases of constructing ToolBench using ChatGPT and how we train our API retriever and ToolLLaMA. During inference, the API retriever recommends relevant APIs for an instruction, then ToolLLaMA performs multiple rounds of reasoning to derive the final answer.

with their inner mechanisms opaque. This limits the democratization of AI technologies and the scope of community-driven innovation and development. In this regard, we deem it urgent to *empower open-source LLMs to skillfully master APIs.*

Although prior works have explored building instruction tuning data for tool use (Li et al., 2023a; Patil et al., 2023; Tang et al., 2023; Xu et al., 2023b), they fail to fully stimulate the tool-use capabilities within LLMs and have inherent limitations: (1) **limited APIs**: they either fail to involve real-world APIs (e.g., RESTAPI) (Patil et al., 2023; Tang et al., 2023) or consider only a small scope of APIs with poor diversity (Patil et al., 2023; Xu et al., 2023b; Li et al., 2023a); (2) **constrained scenario**: existing works are confined to instructions that only involve one single tool. In contrast, real-world scenarios may require that multiple tools are interleaved together for multi-round tool execution to solve a complex task. Besides, they often assume that users specify the ideal API set for a given instruction in advance, which is infeasible when a large collection of APIs are provided; (3) **inferior planning and reasoning**: existing works adopted simple prompting method (e.g., chain-of-thought (CoT) reasoning (Wei et al., 2023) or ReACT (Yao et al., 2022)) for model reasoning, which cannot fully elicit the capabilities stored in LLMs and fail to handle complex instructions. This issue is particularly severe for open-source LLMs, which exhibit markedly inferior reasoning ability compared with their SOTA counterparts. In addition, some works do not even execute APIs to obtain real responses (Patil et al., 2023; Tang et al., 2023), which serve as important information for subsequent model planning.

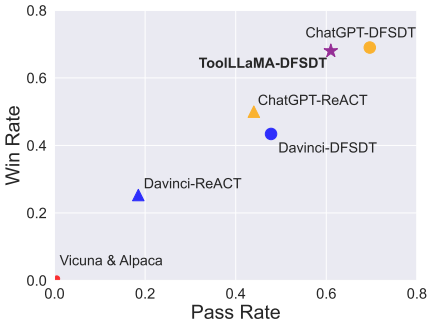

Figure 2: Pass rate and win rate of different methods in tool-use evaluation (higher is better). For win rate, we compare each method with ChatGPT-ReACT. DFSDT is an improved reasoning strategy over ReACT. ToolLLaMA surpasses Text-Davinci-003 and almost performs on par with ChatGPT.

In this paper, we introduce how to empower open-source LLMs to master diverse real-world APIs. As illustrated in Figure 1, we achieve this by collecting a high-quality instruction-tuning dataset **ToolBench**. It is constructed automatically using the latest ChatGPT (`gpt-3.5-turbo-16k`), which is upgraded with enhanced function call[1] capabilities. The comparison between ToolBench and prior works is listed in Table 1. Specifically, the construction of ToolBench entails three phases:

- **API Collection**: we gather **16**, **464** representational state transfer (REST) APIs from RapidAPI[2], a platform that hosts massive real-world APIs provided by developers. These APIs span **49** diverse categories such as social media, e-commerce, and weather. For each API, we crawl detailed API documents from RapidAPI, including the functionality descriptions, required parameters, code snippets for API calls, etc. We expect LLMs to learn to use APIs by comprehending these documents so that the model can generalize to APIs unseen during training;

[1] `https://openai.com/blog/function-calling-and-other-api-updates`
[2] `https://rapidapi.com/hub`

Resource	ToolBench (this work)	APIBench (Patil et al., 2023)	API-Bank (Li et al., 2023a)	ToolAlpaca (Tang et al., 2023)	T-Bench (Xu et al., 2023b)
Real-world API?	✓	✗	✓	✗	✓
Real API Response?	✓	✗	✓	✗	✓
Multi-tool Scenario?	✓	✗	✗	✗	✗
API Retrieval?	✓	✓	✗	✗	✗
Multi-step Reasoning?	✓	✗	✓	✓	✓
Number of tools	**3451**	3	53	400	8
Number of APIs	**16464**	1645	53	400	232
Number of Instances	12657	**17002**	274	3938	2746
Number of Real API Calls	**37204**	0	568	0	0
Avg. Reasoning Traces	4.1	1.0	2.1	1.0	**5.9**

Table 1: A comparison of our ToolBench to representative works. "–" means the information is unknown.

- **Instruction Generation**: we first sample APIs from the pool and then prompt ChatGPT to generate diverse human instructions for these APIs. To cover practical use-case scenarios, we curate instructions that involve both **single-tool** and **multi-tool** scenarios. This ensures that our model learns not only how to interact with individual tools but also how to combine them to accomplish complex tasks;
- **Solution Path Annotation**: we annotate high-quality responses to these instructions. Each response may contain multiple rounds of model reasoning and real-time API calls to derive the final answer. Due to the inherent difficulty of tool learning, even the most sophisticated LLM GPT-4 has a low pass rate for complex instructions, making data collection inefficient. To this end, we develop a novel **depth-first search-based decision tree** (DFSDT) to bolster the planning and reasoning ability of LLMs. Compared with conventional chain-of-thought (CoT) (Wei et al., 2023) and ReACT (Yao et al., 2022), DFSDT enables LLMs to evaluate a multitude of reasoning paths and make deliberate decisions to either retract steps or proceed along a promising path. In experiments, DFSDT significantly improves the annotation efficiency and successfully completes those complex instructions that cannot be answered using CoT or ReACT.

To assess the tool-use capabilities of LLMs, we develop an efficient machine evaluator, **ToolEval**[3], backed up by ChatGPT. It comprises two key metrics: (1) pass rate, which measures the model's ability to successfully execute an instruction within limited budgets, and (2) win rate, which measures the preference of two answer sequences by assessing their quality and usefulness, not just the completion. ToolEval demonstrates a high correlation with human experts and provides a robust, scalable, and reliable assessment for tool learning.

By fine-tuning LLaMA on ToolBench, we obtain **ToolLLaMA**. After evaluation based on our ToolEval, we derive the following findings:

- ToolLLaMA demonstrates a compelling capability to handle both single-tool and complex multi-tool instructions. Uniquely, ToolLLaMA exhibits **robust generalization to previously unseen APIs**, requiring only the API documentation to adapt to new APIs effectively. This flexibility allows users to incorporate novel APIs seamlessly, thus enhancing the model's practical utility. As depicted in Figure 2, ToolLLaMA achieves comparable performance to the "teacher model" ChatGPT in tool use, despite being fine-tuned on merely 12k+ instances.
- We show that our DFSDT serves as a general decision-making strategy to enhance the reasoning capabilities of LLMs. DFSDT broadens the search space by considering multiple reasoning traces and achieves significantly better performance than ReACT.
- Besides, we prompt ChatGPT to recommend relevant APIs for each instruction, and use these information to train a neural **API retriever**. This design alleviates the need for manual selection from the large API pool in practice. We successfully integrate the API retriever with ToolLLaMA. As shown in Figure 1, given an instruction, the API retriever recommends a set of relevant APIs, which are sent to ToolLLaMA for multi-round decision making to derive the final answer. The whole reasoning process will be evaluated by our ToolEval. We show that despite sifting through a large pool of APIs, the retriever exhibits remarkable retrieval precision, returning APIs closely aligned with the ground truth.

[3] https://openbmb.github.io/ToolBench/

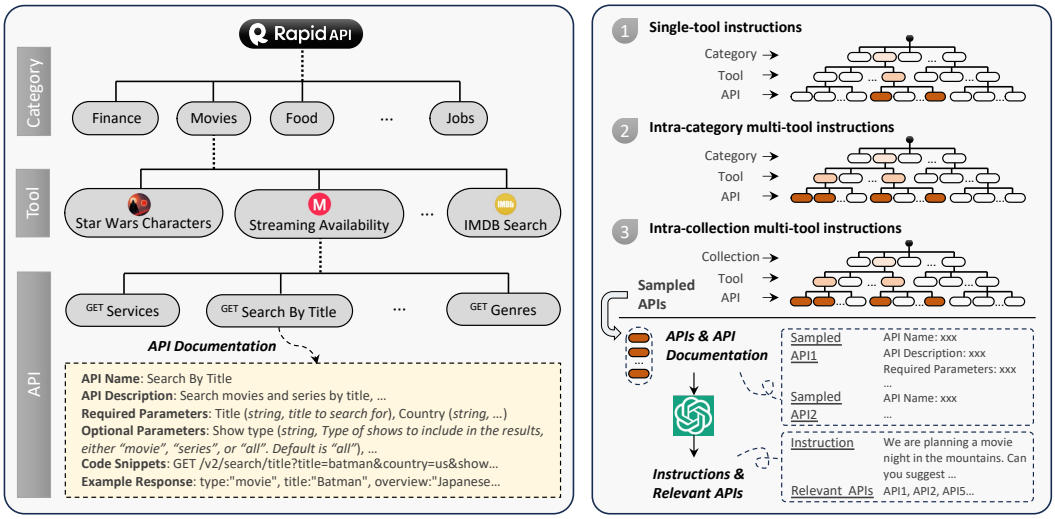

Figure 3: The hierarchy of RapidAPI (left) and the process of instruction generation (right).

In summary, this work targets empowering open-source LLMs to execute complex instructions involving diverse APIs in practical scenarios. We hope this work will inspire further research in the intersection of instruction tuning and tool use.

2 Dataset Construction

In this section, we introduce the construction process of ToolBench, which comprises three stages: API collection (§ 2.1), instruction generation (§ 2.2), and answer generation (§ 2.3). The whole procedure is finished purely using ChatGPT (`gpt-3.5-turbo-16k`), which requires minimal human supervision and can be easily extended to new APIs.

2.1 API Collection

We start by introducing RapidAPI and its hierarchy, followed by how we crawl and filter APIs.

RapidAPI Hub RapidAPI is a leading API marketplace that connects developers with thousands of real-world APIs, streamlining the process of integrating diverse services and data sources into applications. On the platform, developers can discover, test, and connect to various APIs by registering only a RapidAPI key. All APIs in RapidAPI can be classified into 49 *coarse-grained* **categories**[4], such as sports, finance, and weather. The categories are used to associate the API with the most relevant topic. Additionally, the hub also provides a more *fine-grained* categorization called **collections**[5], e.g., Chinese APIs, Top AI-based APIs, and database APIs. APIs in the same collection share a common characteristic and often have similar functionalities or goals.

Hierarchy of RapidAPI As shown in Figure 3, each tool may be composed of multiple APIs. For each tool, we crawl the following information: the name and description of the tool, the URL of the host, and all the available APIs belonging to the tool; for each API, we record its name, description, HTTP method, required parameters, optional parameters, request body, executable code snippets for API call, and an example API call response. This rich and detailed metadata serves as a valuable resource for LLMs to understand and effectively use the APIs, even in a zero-shot manner.

API Filtering Initially, we gather 10,853 tools (53,190 APIs) from RapidAPI. However, the quality and reliability of these APIs can vary significantly. In particular, some APIs may not be

[4] `https://rapidapi.com/categories`
[5] `https://rapidapi.com/collections`

well-maintained, such as returning 404 errors or other internal errors. To this end, we perform a rigorous filtering process to ensure that the ultimate tool set of ToolBench is reliable and functional.

The filtering process is as follows: (1) *initial testing*: we begin by testing the basic functionality of each API to ascertain whether they are operational. We discard any APIs that do not meet this basic criterion; (2) *example response evaluation*: we make API calls to obtain an example response. Then we evaluate their effectiveness by response time and quality. APIs that consistently exhibit a long response time are omitted. Also, we filter out the APIs with low-quality responses, such as HTML source codes or other error messages; Finally, we only retain $3,451$ high-quality tools ($16,464$ APIs).

API Response Compression When examining the response returned by each API, we discover that some responses may contain redundant information and are too long to be fed into LLMs. This may lead to problems due to the limited context length of LLMs. Therefore, we perform a response compression to reduce the length of API responses while maintaining their critical information.

Since each API has a fixed response format, we use ChatGPT to analyze one response example and remove unimportant keys within the response to reduce its length. The prompt of ChatGPT contains the following information for each API: (1) tool documentation, which includes tool name, tool description, API name, API description, parameters, and an example API response. This gives ChatGPT a hint of the API's functionality; (2) 3 in-context learning examples, each containing an original API response and a compressed response schema written by experts. In this way, we obtain the response compression strategies for all APIs. During inference, when the API response length exceeds 2048 tokens, we compress the response by removing unimportant information. If the compressed response is still longer than 2048, we only retain the first 2048 tokens.

2.2 Instruction Generation

Generating high-quality instructions requires two crucial aspects: **diversity**, to ensure the LLMs handle a wide range of API usage scenarios, thereby boosting their generalizability and robustness; and **multi-tool usage**, to mirror real-world situations that often demand the interplay of multiple tools, improving the practical applicability and flexibility of LLMs. To this end, we adopt a bottom-up approach for instruction generation. Instead of brainstorming instructions from scratch and then searching for relevant APIs, we start from the collected APIs in § 2.1 and craft various instructions that involve them. This strategy provides coverage for all collected APIs. Furthermore, we use the RapidAPI hierarchy to identify tool relationships, which aids in the generation of multi-tool instructions.

Generating Instructions for All APIs and their Combinations Define the total set of APIs as $\mathbb{S}_{\text{API}}$, at each time, we sample a few APIs: $\mathbb{S}_{\text{N}}^{\text{sub}} = \{\text{API}_1, \cdots, \text{API}_\text{N}\}$ from $\mathbb{S}_{\text{API}}$. We use different sampling strategies (introduced later) to cover all APIs and most of their combinations. Then we prompt ChatGPT to understand the functionalities of these APIs and their interplay, and then generate (1) possible instructions Inst_* that involve APIs in $\mathbb{S}_{\text{N}}^{\text{sub}}$, and (2) relevant APIs ($\mathbb{S}_{*}^{\text{rel}} \subset \mathbb{S}_{\text{N}}^{\text{sub}}$) for each instruction ($\text{Inst}_*$), i.e., $\{[\mathbb{S}_1^{\text{rel}}, \text{Inst}_1], \cdots, [\mathbb{S}_{\text{N'}}^{\text{rel}}, \text{Inst}_{\text{N'}}]\}$, where N' denotes the number of generated instances at each time. These instruction-relevant API pairs will be used for training the API retriever in § 3.2.

The prompt for ChatGPT is composed of (1) a general description of the intended instruction generation task, (2) comprehensive documentation of each API in $\mathbb{S}_{\text{N}}^{\text{sub}}$, which helps the model understand their functionality, and (3) three in-context seed examples $\{\text{seed}_1, \text{seed}_2, \text{seed}_3\}$. Each seed example is an ideal instruction generation written by human experts. These seed examples are leveraged to better regulate ChatGPT's behavior through in-context learning. In total, we wrote 12 / 36 diverse seed examples ($\mathbb{S}_{\text{seed}}$) for the single-tool / multi-tool setting, and randomly sampled three examples at each time. Detailed prompts for instruction generation are described in appendix A.2. Overall, the generation process can be formulated as follows:

$$\underset{\{\text{API}_1,\cdots,\text{API}_\text{N}\}\in\mathbb{S}_{\text{API}},\{\text{seed}_1,\cdots,\text{seed}_3\}\in\mathbb{S}_{\text{seed}}}{\text{ChatGPT}}(\{[\mathbb{S}_1^{\text{rel}}, \text{Inst}_1], \cdots, [\mathbb{S}_{\text{N'}}^{\text{rel}}, \text{Inst}_{\text{N'}}]\}|\text{API}_1, \cdots, \text{API}_\text{N}, \text{seed}_1, \cdots, \text{seed}_3).$$

Sampling Strategies for Different Scenarios As shown in Figure 3, for the **single-tool instructions (I1)**, we iterate over each tool and generate instructions for its APIs. However, for the **multi-tool setting**, since the interconnections among different tools in RapidAPI are sparse, random sampling

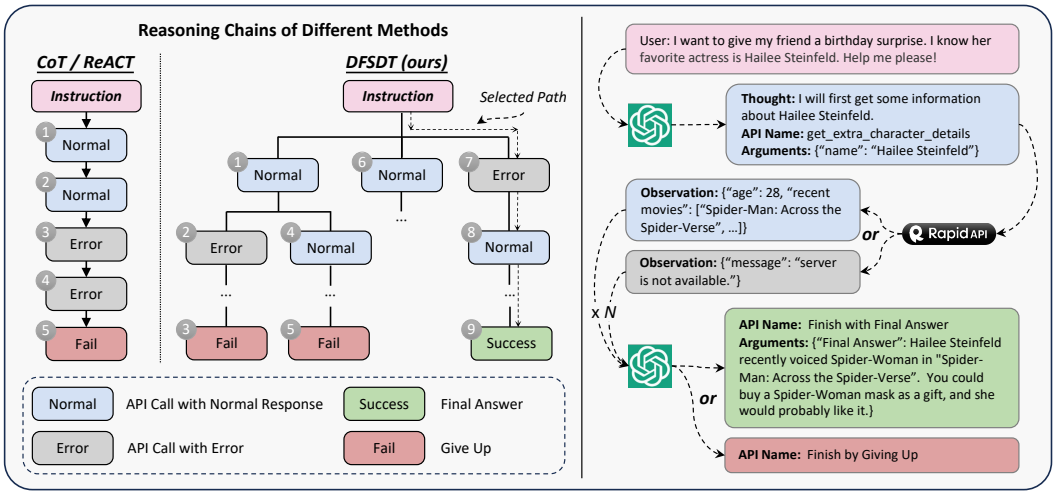

Figure 4: A comparison of our DFSDT and conventional CoT or ReACT during model reasoning (left). We show part of the solution path annotation process using ChatGPT (right).

from the whole tool set often leads to a series of irrelevant tools that a single instruction in a natural way cannot cover. To address the sparsity issue, we leverage the RapidAPI hierarchy information. Since tools belonging to the same category or collection are generally related to each other in the functionality and goals, we randomly select 2-5 tools from each category / collection and sample at most 3 APIs from each tool to generate the instructions. We denote the generated instructions as **intra-category multi-tool instructions (I2)** and **intra-collection multi-tool instructions (I3)**, respectively. Through rigorous human evaluation, we find that instructions already have a high diversity that covers various practical scenarios. We also provide visualization for instructions using Atlas[6].

After generating the initial set of instructions based on the aforementioned method, we further filter those instructions with the hallucinated relevant APIs by assessing whether they exist in $\mathbb{S}_{\mathrm{N}}^{\mathrm{sub}}$. Finally, we collect a total of over 200k qualified instruction-relevant API pairs, including $87,413$, $84,815$, and $25,251$ instances for I1, I2, and I3, respectively.

2.3 Solution Path Annotation

As shown in Figure 4, given an instruction Inst_*, we prompt ChatGPT to search for a valid action sequence: $\{a_1, \cdots, a_{\mathrm{N}}\}$. Such a multi-step decision-making process is cast as a multi-round conversation for ChatGPT. At each time step t, the model generates an action a_t based on previous interactions, i.e., ChatGPT$(a_t|\{a_1, r_1, \cdots, a_{t-1}, r_{t-1}\}, \mathrm{Inst}_*)$, where r_* denotes the real API response. For each action a_t, we prompt ChatGPT to specify which API to use, the specific parameters for an API call, and its "thought". In other words, a_t has the following format: "Thought: $\cdots$, API Name: $\cdots$, Parameters: $\cdots$". Hence the decision space is the Cartesian product of the thought, available APIs, and possible parameters, which is infinite by nature. Detailed prompts are described in appendix A.3.

To leverage the **function call** feature of `gpt-3.5-turbo-16k`, we treat each API as a special function and feed its API description and required / optional parameters into the ChatGPT's function field. In this way, the model understands how to call the API. For each instruction Inst_*, we feed all the sampled APIs $\mathbb{S}_{\mathrm{N}}^{\mathrm{sub}}$ to ChatGPT's function call field instead of only its relevant APIs $\mathbb{S}_*^{\mathrm{rel}}$. In this way, the model gains access to a broader scope of APIs and expands the search space. To finish an action sequence, we define two additional functions for ChatGPT, i.e., "Finish with Final Answer" and "Finish by Giving Up". The former function has a parameter that corresponds to a detailed final answer to the original instruction based on previous actions; while the latter function has no

[6] `https://atlas.nomic.ai/map/58aca169-c29a-447a-8f01-0d418fc4d341/030ddad7-5305-461c-ba86-27e1ca79d899`

parameters and is designed for cases where the provided APIs cannot successfully complete the original instruction after multiple rounds of API call attempts.

Depth First Search-based Decision Tree In our pilot studies, we find that conventional chain-of-thought (CoT) (Wei et al., 2023) or ReACT (Yao et al., 2022) has inherent limitations for decision making: (1) **error propagation**: a mistaken action may propagate the errors further and cause the model to be trapped in a faulty loop, such as continually calling an API in a wrong way or hallucinating APIs; (2) **limited exploration**: although the action space is infinite, CoT or ReACT only explores one possible path, leading to limited exploration of the whole action space. Hence even the most sophisticated GPT-4 often fails to find a valid answer, making solution path annotation difficult.

To this end, we propose to construct a decision tree to expand the search space and increase the possibility of finding a valid reasoning path. As depicted in Figure 4, our DFSDT allows the model to assess different reasoning paths and choose to either (1) proceed along a promising path or (2) abandon an existing node (such as a node with a failed API call) by calling the "Finish by Giving Up" function and expand a new node. During node expansion, to diversify the child nodes and expand the search space, we prompt ChatGPT with the information of the previously generated nodes and explicitly encourage the model to generate a distinct node.

We prefer depth-first search (DFS) instead of breadth-first search (BFS) because we aim to find a reasonable reasoning path rather than expanding the entire space. Using BFS will cost excessive OpenAI API calls before reaching a terminal node (the node with the action "Finish with Final Answer" or "Finish by Giving Up").

In practice, it is essential to balance effectiveness with costs (the number of OpenAI API calls). Therefore, we choose to perform pre-order traversal (a variant for DFS) for DFSDT, which is detailed in appendix A.1. Overall, this design achieves a similar performance as DFS while significantly reducing costs. Ultimately, we generate $12,657$ instruction-answer pairs, which are used to train ToolLLaMA in § 3.3. Although it is possible to construct more training instances, we find that $12,657$ instances already bring satisfying generalization performance.

3 EXPERIMENTS

In this section, we investigate the performance of ToolLLaMA. We start by introducing the evaluation metric for ToolLLaMA in § 3.1, then we evaluate the efficacy of API retriever and DFSDT in § 3.2 and then introduce the main experiments in § 3.3.

3.1 TOOLEVAL

Considering the API's temporal variability, it is infeasible to annotate a fixed ground-truth answer for each test instruction. Besides, we need to ensure that different models use the same version of APIs during evaluation. Considering that human evaluation can be time-consuming, we follow AlpacaEval (Li et al., 2023b) to develop an efficient machine evaluator **ToolEval**, which incorporates two evaluation metrics:

- **Pass Rate**: it calculates the proportion of successfully completing an instruction within limited OpenAI API calls (200 in this paper). The metric measures the executability of instructions for an LLM and can be seen as a basic requirement for ideal tool use. Since we require each model to finish the whole process with the "Finish with Final Answer" action, we treat the percentage of outputting this action in the last as the pass rate. We also define rules to inspect the content of each final answer and exclude those "false positives", such as final answers that contain "I'm sorry, based on the existing APIs, I cannot give a proper response for you".
- **Win Rate**: Since pass rate only measures whether an instruction can be completed, instead of how well it is completed, we additionally adopt another metric: win rate. It is measured by comparing two answers (action sequences and the final answer) for a given instruction. We pre-define a set of criteria for a better answer (see appendix B.2 for details), which are organized as prompts for ChatGPT evaluator. We provide the instruction and two answers to the evaluator and obtain

Instruction	API Retriever (ours)			BM25			Ada Embedding		
	NDCG1	NDCG3	NDCG5	NDCG1	NDCG3	NDCG5	NDCG1	NDCG3	NDCG5
Single-tool (I1)	84.20	89.59	89.65	18.37	17.97	19.65	57.52	54.90	58.83
Category (I2)	68.24	77.43	77.90	11.97	9.85	10.95	36.82	28.83	30.68
Collection (I3)	81.65	87.24	87.13	25.23	18.95	20.37	54.59	42.55	46.83
All	75.73	83.19	83.06	15.84	13.98	15.63	46.59	41.06	43.95

Table 2: We compare our API retriever with two baselines (BM25 and Ada Embeddings) on single-tool, intra-category multi-tool, intra-collection multi-tool instructions, and the whole data,respectively.

Method	Single-tool (I1)	Category (I2)	Collection (I3)	Average
ReACT	43.98	23.62	20.42	29.34
ReACT@N	50.80	36.14	32.87	39.94
DFSDT	**54.10**	**47.35**	**44.80**	**48.75**

Table 3: Pass rate of different answer generation methods for three types of instructions generated in § 2.2 based on ChatGPT. ReACT@N and DFS consume nearly the same OpenAI API calls per instruction.

its preference. We evaluate each answer pair multiple times to improve the reliability. Then we calculate the percentage of being preferred by the evaluator.

To validate the reliability of ChatGPT evaluator, we sample among three different methods (ChatGPT +ReACT, GPT4+ReACT, and ChatGPT +DFSDT) to obtain answer pairs for 600 test instructions. Then we engage humans to annotate human preference (win rate) for them (4 annotations for each answer pair, 2400 annotations in total). Our ChatGPT evaluator demonstrates a high correlation of **75.8%** with human annotators. We also obtain the agreement (see Appendix B.1) among different human annotators (**83.54%**), and the agreement between humans and our evaluator (**80.21%**). This result shows that our evaluator generates highly similar evaluation results to humans and can be viewed as a credible evaluator who simulates human preference. We also find that our automatic evaluator achieves lower variance (**3.47%**) than humans (**3.97%**) when annotating multiple times for the same answer pair. This indicates that our evaluator is more consistent than human counterparts.

3.2 Preliminary Experiments

API Retriever The API retriever aims to retrieve relevant APIs to an instruction. We follow Sentence-BERT (Reimers & Gurevych, 2019) to train a dense retriever based on BERT-BASE (Devlin et al., 2019). The model encodes the instruction and API document into two embeddings, respectively, and the relevance is determined by the similarity of these two embeddings. During training, we regard the relevant APIs of each instruction generated in § 2.2 as positive examples and also sample a few APIs as negative examples for contrastive learning. For baselines, we choose BM25 (Robertson et al., 2009) and OpenAI's text-embedding-ada-002 API[7]. We evaluate the retrieval performance using NDCG (Järvelin & Kekäläinen, 2002). We train and evaluate the model on single-tool instructions (I1), intra-category multi-tool instructions (I2), and intra-collection multi-tool instructions (I3), respectively. Besides, we also merge all the instructions (All) and conduct the training and evaluation.

As shown in Table 2, our API retriever consistently outperforms BM25 and Ada Embedding across different types of instructions. The high NDCG score indicates its efficacy in API retrieval. Additionally, the NDCG score of I1 is much higher than I2 and I3, which means single-tool instructions are relatively simpler for API retrieval than multi-tool counterparts.

Comparing DFSDT and ReACT Before the solution path annotation, we validate the superiority of DFSDT over ReACT. Based on ChatGPT, we compare DFSDT and ReACT for three types of instructions (1000 for each) generated in § 2.2. For evaluation, we choose **Pass Rate**. Since DFSDT consumes more OpenAI API calls than ReACT, for a fairer comparison, we also establish a "ReACT@N" baseline, which conducts multiple times of ReACT until the total costs reach the same level of DFSDT. Once a complete answer is found by ReACT@N, we deem it a success and include it in the Pass Rate calculation.

[7] https://openai.com/blog/new-and-improved-embedding-model

Model	I1-Inst.		I1-Tool		I1-Cat.		I2-Inst.		I2-Cat.		I3-Inst.		Average	
	Pass	Win	Pass	Win	Pass	Win	Pass	Win	Pass	Win	Pass	Win	Pass	Win
ChatGPT-ReACT	56.0	-	62.0	-	66.0	-	28.0	-	22.0	-	30.0	-	44.0	-
Vicuna (ReACT & DFSDT)	0.0	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0	-
Alpaca (ReACT & DFSDT)	0.0	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0	-	0.0	-
Text-Davinci-003-DFSDT	53.0	46.0	58.0	38.0	61.0	39.0	38.0	46.0	38.0	45.0	39.0	48.0	47.8	43.7
ChatGPT-DFSDT	**78.0**	**68.0**	**84.0**	**59.0**	**89.0**	**57.0**	**51.0**	**78.0**	**58.0**	77.0	**57.0**	**77.0**	**69.6**	**69.3**
ToolLLaMA-DFSDT	68.0	**68.0**	80.0	**59.0**	75.0	56.0	47.0	75.0	56.0	**80.0**	40.0	72.0	61.0	68.3

Table 4: Main experiments on the test set of ToolBench. Win rate is calculated by comparing each model with ChatGPT-ReACT. A win rate higher than 50% means the model performs better than ChatGPT-ReACT.

From Table 3, it can be observed that DFSDT significantly outperforms the two baselines in all scenarios, showing that DFSDT is a more efficient way that saves the costs for solution path annotation. We also find that the performance improvement of DFSDT is more evident for harder instructions (i.e., I2 and I3) than those simpler instructions (I1). This means that besides efficiency, DFSDT can solve those difficult, complex instructions that are unanswerable by the vanilla ReACT no matter how many times it is performed. Involving such "hard examples" in our dataset can fully elicit the tool-use capabilities for those complex scenarios.

3.3 Main Experiments

ToolLLaMA We fine-tune LLaMA 7B model using the instruction-answer pairs. The original LLaMA model is pre-trained with a sequence length of 2048, which is not enough under our setting since the tool response can be very long. To this end, we use positional interpolation (Chen et al., 2023) to extend the context length to 8192. We train the model in a multi-round conversation mode. For the training data format, we keep the input and output the same as those of ChatGPT. Since it is unclear how ChatGPT organizes the function call field, we just concatenate this information into the input as part of the prompt for ToolLLaMA.

Settings Ideally, by scaling the number and diversity of instructions and unique tools in the training data, ToolLLaMA is expected to generalize to new instructions and tools unseen during training. This is more practical since users can define customized APIs and expect ToolLLaMA to adapt accordingly. To this end, we strive to evaluate the **generalization ability** of our ToolLLaMA at three levels: (1) **Inst.**: **unseen instructions** for the same set of tools in the training data, (2) **Tool**: **unseen tools** that belong to the **same (seen) category** of the tools in the training data, and (3) **Cat.**: **unseen tools** that belong to a **different (unseen) category** of tools in the training data.

We perform experiments on three scenarios: single-tool instructions (I1), intra-category multi-tool instructions (I2), and intra-collection multi-tool instructions (I3). For I1 and I2, we randomly select six categories as the testing categories, leaving the remaining 43 categories as training data. For I1, we conduct the evaluation for the aforementioned three levels (I1-Inst., I1-Tool, and I1-Cat.); for I2, since the training instructions already involve different tools of the same category, we only perform level 1 and level 3 for the generalization evaluation (I2-Inst. and I2-Cat.); similarly, we only perform level 1 generalization for I3 (I3-Inst.) since it already covers instructions that involve various combinations of tools from different categories (the tools in a RapidAPI collection may come from different RapidAPI categories). For each test instruction, we feed the ground truth APIs $\mathbb{S}_{\mathrm{N}}^{\mathrm{sub}}$ to each model. This simulates the scenario where the user specifies the API set they prefer.

Baselines Since the original LLaMA checkpoint is not fine-tuned toward any downstream task, it cannot be leveraged to use tools directly. Instead, we choose two LLaMA variants that have been fine-tuned for general-purpose instruction tuning on dialogue data, i.e., Vicuna (Chiang et al., 2023) and Alpaca (Taori et al., 2023). Both models have shown strong instruction-following capabilities. We conduct sophisticated prompt engineering for both models to elicit the best of their tool-use abilities. We also choose the "teacher model" (ChatGPT) and Text-Davinci-003 as the baseline. We apply DFSDT to all these models and also apply ReACT to ChatGPT. When calculating the win rate, each model is compared with ChatGPT-ReACT.

Main Results The results are placed in Table 4, from which we observe that: (1) ToolLLaMA significantly outperforms the conventional method for tool use, i.e., ChatGPT-ReACT, in both

Model	I1-Inst.		I1-Tool		I1-Cat.		I2-Inst.		I2-Cat.		I3-Inst.		Average	
	Pass	Win	Pass	Win	Pass	Win	Pass	Win	Pass	Win	Pass	Win	Pass	Win
ToolLLaMA	**68.0**	-	**80.0**	-	**75.0**	-	**47.0**	-	**56.0**	-	40.0	-	**61.0**	-
→API Retriever	62.0	**54.0**	62.0	39.0	72.0	**49.0**	45.0	**54.0**	55.0	**51.0**	**47.0**	52.0	57.2	**49.8**
→ReACT	19.0	21.0	21.0	13.0	24.0	16.0	6.0	9.0	14.0	9.0	6.0	10.0	15.0	13.0
→LoRA	51.0	34.0	63.0	**44.0**	61.0	39.0	38.0	38.0	42.0	42.0	45.0	**54.0**	50.0	41.8

Table 5: Additional analyses of ToolLLaMA: (1) replacing the ground truth APIs with those recommended by our API retriever, (2) degrading the reasoning method from DFSDT to ReACT, and (3) tuning LLaMA using LoRA instead of full-parameter fine-tuning. We compare each variant with the original ToolLLaMA for win rate.

pass rate and win rate, exhibiting superior generalization abilities. Besides, ToolLLaMA also performs better than Text-Dainci-003 when combined with DFSDT. (2) Although we conduct prompt engineering extensively, both Vicuna and Alpaca fail to pass any instructions, which means their instruction-following abilities do not extend to the tool-use scenarios. This underscores the deficiency of current instruction tuning methods, which largely focus on enhancing language skills. (2) In general, ToolLLaMA demonstrates competitive performance in all scenarios, achieving a pass rate slightly lower than ChatGPT+DFSDT. For the Win Rate, ToolLLaMA generally matches ChatGPT's capability and even surpasses the latter in the **I2-Cat.** setting. Overall, these results demonstrate that ToolBench can sufficiently elicit the tool-use capabilities within open-source LLMs and empower them to skillfully master even unseen APIs for various instructions.

Integrating API Retriever with ToolLLaMA In practice, users may not be able to manually recommend APIs from a large pool. To emulate this practical setting and test the efficiency of our API retriever, we replace the ground truth APIs fed to ToolLLaMA with the top 5 APIs recommended by our API retriever. As seen from Table 5, the use of our API retriever only slightly diminishes the pass rate when compared to the ground truth API set. The average win rate of ToolLLaMA is 49.8, which is a significant achievement given the vast pool of APIs from which our API retriever has to select. It provides robust evidence of the ability of our API retriever to retrieve relevant APIs. We also find that the win rate shows a slight increase in several scenarios when using the API retriever. This suggests that the ground truth API set contains many APIs that can be replaced by those with similar or even better functionalities, which are successfully identified by our API retriever from the large pool.

Comparing DFSDT with ReACT Since ReACT can be viewed as a degraded version of DFSDT, although ToolLLaMA is trained on data generated by DFSDT, the model can be used either through ReACT or DFSDT during inference. Here we directly compare the difference between these two decision-making strategies for ToolLLaMA and the results are shown in Table 5. It can be derived that DFSDT achieves a significantly higher pass rate and is more preferred across all the scenarios. This comparison underscores the superiority of DFSDT over ReACT, demonstrating its higher performance in decision-making tasks. Besides, compared with the results in Table 3, we find that the improvements brought by DFSDT over ReACT is more evident for ToolLLaMA than ChatGPT, this demonstrates that expanding the search space is extremely essential for those LLMs with inferior reasoning capabilities. This finding shows the potential utility of applying DFSDT to small-scale models in practice.

ToolLLaMA with Better Parameter Efficiency In previous experiments, we fine-tune all the parameters of LLaMA to obtain ToolLLaMA. To improve the parameter efficiency, we further apply a representative parameter-efficient tuning (Ding et al., 2022) method LoRA (Hu et al., 2021) and investigate how the performance is affected. The results in Table 5 indicate that improved parameter efficiency is achieved with a trade-off in performance (e.g., 11% in pass rate). We expect future attempts to design more sophisticated methods that could achieve parameter efficiency without sacrificing performance.

4 Related Work

Tool Learning Recent studies have shed light on the burgeoning capabilities of LLMs in mastering tools and making decisions within complex environments (Qin et al., 2023b; Vemprala et al., 2023; Nakano et al., 2021; Qin et al., 2023a; Shen et al., 2023; Wu et al., 2023; Schick et al., 2023; Hao et al., 2023; Qian et al., 2023; Song et al., 2023). Gaining access to external tools endows LLMs with real-time factual knowledge (Yang et al., 2023), multimodal functionalities (Gupta & Kembhavi, 2023), and specialized skills in vertical domains (Jin et al., 2023). However, open-source LLMs still lag far behind SOTA LLMs in tool use, and how tool-use ability is acquired by SOTA LLMs remains unclear. In this paper, we aim to bridge this gap and fathom the underlying mechanism.

Instruction Tuning and Data Augmentation Instruction tuning enhances LLMs in understanding human instructions and generating proper responses (Wei et al., 2021; Bach et al., 2022; Mishra et al., 2022). Since manually annotating instruction tuning data is time-consuming, self-instruct (Wang et al., 2022) proposes to generate high-quality data from SOTA LLMs, which facilitates a recent trend of data curation for multi-turn dialogue (Taori et al., 2023; Chiang et al., 2023; Xu et al., 2023a; Penedo et al., 2023; Ding et al., 2023). However, compared with the dialogue, tool learning is inherently more challenging given the vast diversity of APIs and the complexity of multi-tool instructions. As a result, even the most sophisticated GPT-4 often fails to produce correct responses for tool use, making data collection difficult. Hence, existing tool-learning dataset (Li et al., 2023a; Patil et al., 2023; Tang et al., 2023; Xu et al., 2023b) and their construction methods are still in their infancy and cannot effectively address real human needs as mentioned in § 1. Instead, our ToolBench is designed for practical scenarios and improves the previous pipeline for tool-learning data construction. Besides, we target making LLMs generalize to diverse tool-use scenarios instead of focusing on a particular type of tool (Nakano et al., 2021), which is less practical.

Prompting LLMs for Decision Making Prompting facilitates LLMs to decompose high-level tasks into sub-tasks (Huang et al., 2022a) and generate grounded plans (Ahn et al., 2022; Huang et al., 2022b). ReACT (Yao et al., 2022) proposes to better integrate reasoning with acting by allowing LLMs to give a proper reason for an action and incorporating environmental feedback for reasoning. However, these studies do not incorporate a mechanism for decision retraction, which becomes problematic as an initial error can propagate along an action sequence, leading to a cascade of subsequent errors. Recently, Reflexion (Shinn et al., 2023) tries to eliminate this issue by asking LLMs to reflect on previous failures to correct its decision making. Instead, our DFSDT extends Reflexion to a more general method by allowing LLMs to assess different reasoning paths and select the most promising one. It should be noted our DFSDT shares a similar idea to the recent tree-of-thought (ToT) reasoning (Yao et al., 2023). However, our DFSDT targets addressing general decision-making problems where the decision space is *infinite*, compared to ToT's simple tasks that can be easily addressed by brute-force search, such as Game of 24 and Crosswords. The distinct target determines the significant difference in the implementation details. Notably, our method is designed for diverse decision-making tasks, while ToT is tailored specifically for its selected task set.

5 Conclusion

This work introduces how to elicit the tool-use capabilities within LLMs. We present an instruction tuning dataset, ToolBench, which covers 16k+ real-world APIs and various practical use-case scenarios including both single-tool and multi-tool tasks. The construction of ToolBench purely uses ChatGPT and requires minimal human supervision. Moreover, we propose DFSDT to reinforce the planning and reasoning ability of LLMs, enabling them to navigate through reasoning paths strategically. By fine-tuning LLaMA on ToolBench, the obtained model ToolLLaMA matches the performance of ChatGPT and exhibits remarkable generalization ability to unseen APIs. Besides, we develop a neural API retriever to recommend relevant APIs for each instruction. The retriever can be integrated with ToolLLaMA as a more automated tool-use pipeline. We also introduce an automatic evaluator **ToolEval** for efficient evaluation of tool learning. In general, this work paves the way for future research in the intersection of instruction tuning and tool use for LLMs.

ACKNOWLEDGEMENTS

The contributions are listed as follows: (1) API collection: Shihao Liang, Sihan Zhao, Kunlun Zhu, Yujia Qin; (2) instruction generation: Lan Yan, Kunlun Zhu, Shihao Liang, Yujia Qin; (3) solution path annotation: Yining Ye, Shihao Liang, Runchu Tian, Yujia Qin, Xin Cong; (4) model implementation: Shihao Liang, Yujia Qin, Kunlun Zhu, Yifan Wu; (5) system demonstration: Xiangru Tang, Bill Qian. Yujia Qin led the project, designed the methodology and experiments, and wrote the paper. Yankai Lin, Mark Gerstein, Dahai Li, Zhiyuan Liu, Maosong Sun, and Jie Zhou advised the project. Yankai Lin, Xin Cong, Ruobing Xie proofread the whole paper. All authors participated in the discussion.

The authors would like to thank Yifan Wu, Si Sun, Zheni Zeng, Chen Zhang, Yu Gu, Chenfei Yuan, Junxi Yan, Shizuo Tian, Mingxi Yan, and Jason Phang for their valuable feedback, discussion, and participation in this project.

REFERENCES

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

Stephen Bach, Victor Sanh, Zheng Xin Yong, Albert Webson, Colin Raffel, Nihal V Nayak, Abheesht Sharma, Taewoon Kim, M Saiful Bari, Thibault Févry, et al. Promptsource: An integrated development environment and repository for natural language prompts. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pp. 93–104, 2022.

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

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023. URL `https://lmsys.org/blog/2023-03-30-vicuna/`.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL `https://aclanthology.org/N19-1423`.

Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. Delta tuning: A comprehensive study of parameter efficient methods for pre-trained language models. *arXiv preprint arXiv:2203.06904*, 2022.

Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Zhi Zheng, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. Enhancing chat language models by scaling high-quality instructional conversations. *arXiv preprint arXiv:2305.14233*, 2023.

Tanmay Gupta and Aniruddha Kembhavi. Visual programming: Compositional visual reasoning without training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14953–14962, 2023.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. *arXiv preprint arXiv:2305.11554*, 2023.

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

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato (eds.), *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pp. 9118–9147. PMLR, 2022a.

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

Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

Qiao Jin, Yifan Yang, Qingyu Chen, and Zhiyong Lu. Genegpt: Augmenting large language models with domain tools for improved access to biomedical information. *ArXiv*, 2023.

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

Xuechen Li, Tianyi Zhang, Yann Dubois, Rohan Taori, Ishaan Gulrajani, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Alpacaeval: An automatic evaluator of instruction-following models. `https://github.com/tatsu-lab/alpaca_eval`, 2023b.

Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. Cross-task generalization via natural language crowdsourcing instructions. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3470–3487, 2022.

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

OpenAI. OpenAI: Introducing ChatGPT, 2022. URL `https://openai.com/blog/chatgpt`.

OpenAI. Gpt-4 technical report, 2023.

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

Guilherme Penedo, Quentin Malartic, Daniel Hesslow, Ruxandra Cojocaru, Alessandro Cappelli, Hamza Alobeidli, Baptiste Pannier, Ebtesam Almazrouei, and Julien Launay. The refinedweb dataset for falcon llm: outperforming curated corpora with web data, and web data only. *arXiv preprint arXiv:2306.01116*, 2023.

Cheng Qian, Chi Han, Yi R Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. Creator: Disentangling abstract and concrete reasonings of large language models through tool creation. *arXiv preprint arXiv:2305.14318*, 2023.

Yujia Qin, Zihan Cai, Dian Jin, Lan Yan, Shihao Liang, Kunlun Zhu, Yankai Lin, Xu Han, Ning Ding, Huadong Wang, et al. Webcpm: Interactive web search for chinese long-form question answering. *arXiv preprint arXiv:2305.06849*, 2023a.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, et al. Tool learning with foundation models. *arXiv preprint arXiv:2304.08354*, 2023b.

Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

Stephen Robertson, Hugo Zaragoza, et al. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389, 2009.

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

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface, 2023.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023.

Yifan Song, Weimin Xiong, Dawei Zhu, Cheng Li, Ke Wang, Ye Tian, and Sujian Li. Restgpt: Connecting large language models with real-world applications via restful apis. *arXiv preprint arXiv:2306.06624*, 2023.

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

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

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Sai Vemprala, Rogerio Bonatti, Arthur Bucker, and Ashish Kapoor. Chatgpt for robotics: Design principles and model abilities. Technical Report MSR-TR-2023-8, Microsoft, February 2023.

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

Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

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

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. Wizardlm: Empowering large language models to follow complex instructions, 2023a.

Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. On the tool manipulation capability of open-source large language models. *arXiv preprint arXiv:2305.16504*, 2023b.

Linyao Yang, Hongyang Chen, Zhao Li, Xiao Ding, and Xindong Wu. Chatgpt is not enough: Enhancing large language models with knowledge graphs for fact-aware language modeling. *arXiv preprint arXiv:2306.11489*, 2023.

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

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*, 2023.

Method	Single-tool (I1)	Category (I2)	Collection (I3)
Average word count	42.82	47.53	46.66
Average relevant API count	2.23	2.64	2.90
Average tool count	1.00	2.17	2.22
Average category count	1.00	1.00	2.10

Table 6: Average counts for different metrics across three instruction groups.

APPENDIX

A IMPLEMENTATION DETAILS

A.1 DETAILS FOR DFSDT

Classical DFS algorithms generate multiple child nodes at each step, then sort all the child nodes, and select the highest-scoring node for expansion. After greedily expanding to the terminal node, DFS backtracks gradually to explore nearby nodes, expanding the search space. Throughout the algorithm, the most resource-intensive part is the sorting process of child nodes. If we use an LLM to evaluate two nodes at a time, it requires approximately $O(n \log n)$ complexity of OpenAI API calls, where n is the number of child nodes.

In fact, we find empirically that in most cases, the nodes ranked highest are often the node generated at first. Therefore, we skip the sorting process of child nodes and reversed the order of generating child nodes. Ultimately, we choose a pre-order traversal for the tree search. The comparison between the classical DFS and our DFSDT is shown in appendix A.1. This design has the following advantages:

- If the model does not retract an action (e.g., for the case of simple instructions), then DFSDT degrades to CoT, which makes it as efficient as CoT.
- After the algorithm finishes, the nodes explored by this method are almost the same as those found by a classical DFS search. Hence, it can also handle complex instructions that only DFS can solve.

During the process of data generation, we found that DFSDT consumes approximately 4 times the search resources compared to CoT.

Algorithm 1: Normal DFS Algorithm with Sort function

Input: Tree Width W, max tree depth L, root node N
Output: answer node Ans
if N.*depth* $\geq L$ **then**
  return None
candidates $\leftarrow \emptyset$
for *int* $i = 0; i < W$ **do**
  generate new child node new node M from N
  M.father $= N$
  candidates.append(M)
candidates.sort() `// best first`
for *cont in candidates* **do**
  if *cont.finish* **then**
    return cont
  output = DFS($W, L + 1$, cont)
  if *output.finish* **then**
    return output
return None

Algorithm 2: ToolBench DFS Algorithm

Input: Tree Width W, max tree depth L, root node N
Output: answer node Ans
if N.*depth* $\geq L$ **then**
  return None
candidates $\leftarrow \emptyset$
for *int* $i = 0; i < W$ **do**
  generate new child node new node M from N, that is different from all candidates
  M.father $= N$
  if *M.finish* **then**
    return M
  output = DFS($W, L + 1, M$)
  if *output.finish* **then**
    return output
  candidates.append(M)
return None

A.2 PROMPTS FOR INSTRUCTION GENERATION

Below we list the detailed prompt for instruction generation, which consists of four parts: task introduction, in-context learning examples, sampled API list, and other requirements.

Task Introduction of Single-tool Instructions:
You will be provided with a tool, its description, all of the tool's available API functions, the descriptions of these API functions, and the parameters required for each API function. Your task involves creating 10 varied, innovative, and detailed user queries that employ multiple API functions of a tool. For instance, if the tool 'climate news' has three API calls - 'get_all_climate_change_news', 'look_up_climate_today', and 'historical_climate', your query should articulate something akin to: first, determine today's weather, then verify how often it rains in Ohio in September, and finally, find news about climate change to help me understand whether the climate will change anytime soon. This query exemplifies how to utilize all API calls of 'climate news'. A query that only uses one API call will not be accepted. Additionally, you must incorporate the input parameters required for each API call. To achieve this, generate random information for required parameters such as IP address, location, coordinates, etc. For instance, don't merely say 'an address', provide the exact road and district names. Don't just mention 'a product', specify wearables, milk, a blue blanket, a pan, etc. Don't refer to 'my company', invent a company name instead. The first seven of the ten queries should be very specific. Each single query should combine all API call usages in different ways and include the necessary parameters. Note that you shouldn't ask 'which API to use', rather, simply state your needs that can be addressed by these APIs. You should also avoid asking for the input parameters required by the API call, but instead directly provide the parameter in your query. The final three queries should be complex and lengthy, describing a complicated scenario where all the API calls can be utilized to provide assistance within a single query. You should first think about possible related API combinations, then give your query. Related_apis are apis that can be used for a give query; those related apis have to strictly come from the provided api names. For each query, there should be multiple related_apis; for different queries, overlap of related apis should be as little as possible. Deliver your response in this format: [Query1: ....., 'related_apis':[api1, api2, api3...],Query2: ....., 'related_apis':[api4, api5, api6...],Query3: ....., 'related_apis':[api1, api7, api9...], ...]

Task Introduction of Multi-tool Instructions:
You will be provided with several tools, tool descriptions, all of each tool's available API functions, the descriptions of these API functions, and the parameters required for each API function. Your task involves creating 10 varied, innovative, and detailed user queries that employ API functions of multiple tools. For instance, given three tools 'nba_news', 'cat-facts', and 'hotels': 'nba_news' has API functions 'Get individual NBA source news' and 'Get all NBA news', 'cat-facts' has API functions 'Get all facts about cats' and 'Get a random fact about cats', 'hotels' has API functions 'properties/get-details (Deprecated)', 'properties/list (Deprecated)' and 'locations/v3/search'. Your query should articulate something akin to: 'I want to name my newborn cat after Kobe and host a party to celebrate its birth. Get me some cat facts and NBA news to gather inspirations for the cat name. Also, find a proper hotel around my house in Houston Downtown for the party.' This query exemplifies how to utilize API calls of all the given tools. A query that uses API calls of only one tool will not be accepted. Additionally, you must incorporate the input parameters required for each API call. To achieve this, generate random information for required parameters such as IP address, location, coordinates, etc. For instance, don't merely say 'an address', provide the exact road and district names. Don't just mention 'a product', specify wearables, milk, a blue blanket, a pan, etc. Don't refer to 'my company', invent a company name instead. The first seven of the ten queries should be very specific. Each single query should combine API calls of different tools in various ways and include the necessary parameters. Note that you shouldn't ask 'which API to use', rather, simply state your needs that can be addressed by these APIs. You should also avoid asking for the input parameters required by the API call, but instead directly provide the parameters in your query. The final three queries should be complex and lengthy, describing a complicated scenario where all the provided API calls can be utilized to provide assistance within a single query. You should first think about possible related API combinations, then give your query. Related APIs are APIs that can be used for a given query; those related APIs have to strictly come from the provided API names. For each query, there should be multiple related APIs; for different

queries, overlap of related APIs should be as little as possible. Deliver your response in this format: [Query1: ....., 'related_apis':[[¡tool name¿, ¡api name¿], [¡tool name¿, ¡api name¿], [¡tool name¿, ¡api name¿]...],Query2: ....., 'related_apis':[[¡tool name¿, ¡api name¿], [¡tool name¿, ¡api name¿], [¡tool name¿, ¡api name¿]...],Query3: ....., 'related_apis':[[¡tool name¿, ¡api name¿], [¡tool name¿, ¡api name¿], [¡tool name¿, ¡api name¿]...], ...]

In-context Seed Examples. In the following, we show one single-tool instruction seed example and one multi-tool instruction seed example.

For example, with tool ASCII Art, the given api_names are 'figlet', 'list figlet styles', 'cowsay', 'list_cowsay_styles', 'matheq'.
Some sample queries and related_apis would be:
"Query": "Need to create an ASCII art representation of a mathematical equation. The equation is 'y = mx + c', where m and c are constants. Help me generate the ASCII art for this equation. Also please generate an ASCII art representation of the text 'Newton's Second Law of Motion'.", "related_apis": ['figlet', 'list figlet styles', 'matheq']
"Query": "Working on a research paper on cows and need to include ASCII art representations of various cows. Can you first retrieve available ASCII art styles for cows? Then, can you generate ASCII art for cows like the Jersey, Holstein, and Guernsey? Finally, I want the cow to say 'Moo!' in the ASCII art.", "related_apis": ['figlet', 'list figlet styles', 'cowsay', 'list_cowsay_styles']
"Query": "I'm writing a blog post on ASCII art and need to include some examples. Can you generate ASCII art for the following strings: 'ASCII', 'art', and 'gallery'? You can first retrieve available figlet styles and then generate ASCII art for the strings using the styles.", "related_apis": ['figlet', 'list figlet styles']
"Query": "Greetings! I'm putting together a quirky slideshow about our furry friends and need your help to sprinkle some ASCII art goodness. Could you kindly fetch me the catalog of ASCII art styles available for animals? Also, I'm particularly keen on featuring ASCII art for creatures like pandas, cows, elephants, and penguins. And if they could say something cute like 'Hello!' or 'Hugs!' in the ASCII art, that would be purr-fect!", "related_apis": ['figlet', 'list figlet styles', 'cowsay', 'list_cowsay_styles']

For example, with tool ['Entrepreneur Mindset Collection', 'Random Words', 'thedigitalnewsfeederapi', 'Chemical Elements'], the given api_names are (tool 'Entrepreneur Mindset Collection')'Random Quote in JSON format', (tool 'Random Words')'Get multiple random words', (tool 'Random Words')'Get a random word', (tool 'thedigitalnewsfeederapi')'getting specific cricket articles', (tool 'thedigitalnewsfeederapi')'Getting Cricket Articles', (tool 'thedigitalnewsfeederapi')'getting specific news articles', (tool 'thedigitalnewsfeederapi')'Getting News Articles', (tool 'thedigitalnewsfeederapi')'getting all news articles', (tool 'Chemical Elements')'Get All Chemical Elements'.
Some sample queries and related_apis would be:
"Query": "For my best friend's surprise birthday party, I require inspiration for party games and decorations. Kindly suggest some random words that can serve as themes for the party. Furthermore, I'm interested in gathering news articles about the latest party trends to ensure a modern celebration. Also, I would appreciate details about the local hotels in my area for accommodation options. Your assistance is greatly appreciated.", "related_apis": [['Random Words', 'Get multiple random words'], ['thedigitalnewsfeederapi', 'Getting News Articles'], ['thedigitalnewsfeederapi', 'Getting all news articles']]
"Query": "In the midst of organizing a team-building event for my esteemed company, I eagerly seek your valued input for invigorating activities. Might I kindly request a collection of random quotes that encapsulate the essence of teamwork and motivation? Additionally, I am keen on exploring news articles that showcase triumphant team-building events, as they serve as a wellspring of inspiration.", "related_apis": [['Entrepreneur Mindset Collection', 'Random Quote in JSON format'], ['thedigitalnewsfeederapi', 'Getting News Articles']] "Query": "I need specific cricket articles that discuss the health benefits of sports for my research paper on exercise. I also want to know which chemical elements are associated with exercising, like increased iron (Fe) and its impact on bone marrow.", "related_apis": [['thedigitalnewsfeederapi', 'getting specific cricket articles'], ['Chemical Elements', 'Get All Chemical Elements']]
"Query": "I'm starting a new business venture and I need to make a speech announcing the new

dawn. Provide me some quotes and words for me to start with. I would like to gather news articles about successful entrepreneurs for inspiration.", "related_apis": [['Entrepreneur Mindset Collection', 'Random Quote in JSON format'], ['Random Words', 'Get multiple random words'], ['thedigitalnewsfeederapi', 'getting specific news articles']]
These are only examples to show you how to write the query. Do not use APIs listed in the above examples, but rather, use the ones listed below in the INPUT.

Sampled API List (An example)


```
{
    "tool_description": "EntreAPI Faker is used to dynamically
       create mock, demo, test and sample data for your
       application",
    "name": "EntreAPI Faker",
    "api_list": [
        {
            "name": "Longitute",
            "url": "https://entreapi-faker.p.rapidapi.com/address/
               longitude",
            "description": "Generate a random longitude.",
            "method": "GET",
            "required_parameters": [],
            "optional_parameters": [
                {
                    "name": "max",
                    "type": "NUMBER",
                    "description": "Maximum value for latitude.",
                    "default": ""
                },
                {
                    "name": "min",
                    "type": "NUMBER",
                    "description": "Minimum value for latitude.",
                    "default": ""
                },
                {
                    "name": "precision",
                    "type": "NUMBER",
                    "description": "Precision for latitude.",
                    "default": ""
                }
            ],
            "tool_name": "EntreAPI Faker",
            "category_name": "Data"
        },
        {
            "name": "Boolean",
            "url": "https://entreapi-faker.p.rapidapi.com/datatype
               /boolean",
            "description": "Randomly generate a boolean value.",
            "method": "GET",
            "required_parameters": [],
            "optional_parameters": [],
            "tool_name": "EntreAPI Faker",
            "category_name": "Data"
        },
        {
            "name": "Past",
```

```
        "url": "https://entreapi-faker.p.rapidapi.com/date/
           past",
        "description": "Randomly generate a date value in the
           past.",
        "method": "GET",
        "required_parameters": [],
        "optional_parameters": [
            {
                "name": "refDate",
                "type": "STRING",
                "description": "Starting reference date",
                "default": ""
            },
            {
                "name": "years",
                "type": "NUMBER",
                "description": "Number of years for the range
                   of dates.",
                "default": ""
            }
        ],
        "tool_name": "EntreAPI Faker",
        "category_name": "Data"
    },
    {
        "name": "Image Url",
        "url": "https://entreapi-faker.p.rapidapi.com/image/
           imageUrl",
        "description": "Randomly generate an image URL.",
        "method": "GET",
        "required_parameters": [],
        "optional_parameters": [
            {
                "name": "width",
                "type": "NUMBER",
                "description": "Width of the image. Default is
                    640.",
                "default": ""
            },
            {
                "name": "height",
                "type": "NUMBER",
                "description": "Height of the image. Default
                   is 480.",
                "default": ""
            },
            {
                "name": "useRandomize",
                "type": "BOOLEAN",
                "description": "Add a random number parameter
                   to the returned URL.",
                "default": ""
            },
            {
                "name": "category",
                "type": "STRING",
                "description": "The category for the image.
                   Can be one: abstract, animal, avatar,
                   business, cats, city, fashion, food,
```

```
                    nature, nightlife, people, sports,
                    technics, transport",
                "default": ""
            }
        ],
        "tool_name": "EntreAPI Faker",
        "category_name": "Data"
    },
    {
        "name": "Sentence",
        "url": "https://entreapi-faker.p.rapidapi.com/lorem/
            sentence",
        "description": "Randomly generate a sentence of Lorem
            Ipsum.",
        "method": "GET",
        "required_parameters": [],
        "optional_parameters": [
            {
                "name": "wordCount",
                "type": "NUMBER",
                "description": "Number of words in the
                    sentence.",
                "default": ""
            }
        ],
        "tool_name": "EntreAPI Faker",
        "category_name": "Data"
    },
    {
        "name": "Gender",
        "url": "https://entreapi-faker.p.rapidapi.com/name/
            gender",
        "description": "Randomly select a gender.",
        "method": "GET",
        "required_parameters": [],
        "optional_parameters": [
            {
                "name": "useBinary",
                "type": "BOOLEAN",
                "description": "Use binary genders only.",
                "default": ""
            }
        ],
        "tool_name": "EntreAPI Faker",
        "category_name": "Data"
    },
    {
        "name": "Prefix",
        "url": "https://entreapi-faker.p.rapidapi.com/name/
            prefix",
        "description": "Randomly generate a prefix (e.g., Mr.,
             Mrs., etc.)",
        "method": "GET",
        "required_parameters": [],
        "optional_parameters": [
            {
                "name": "gender",
                "type": "STRING",
                "description": "Optional gender.",
```

```
                "default": ""
            }
        ],
        "tool_name": "EntreAPI Faker",
        "category_name": "Data"
    },
    {
        "name": "Array Element",
        "url": "https://entreapi-faker.p.rapidapi.com/random/
           arrayElement",
        "description": "Randomly select an array element.",
        "method": "GET",
        "required_parameters": [],
        "optional_parameters": [
            {
                "name": "array",
                "type": "ARRAY",
                "description": "The list of elements to choose
                    from. Default is [\"a\", \"b\", \"c\"].",
                "default": ""
            }
        ],
        "tool_name": "EntreAPI Faker",
        "category_name": "Data"
    },
    {
        "name": "Number Value",
        "url": "https://entreapi-faker.p.rapidapi.com/random/
           number",
        "description": "Randomly generate a number value.",
        "method": "GET",
        "required_parameters": [],
        "optional_parameters": [
            {
                "name": "min",
                "type": "NUMBER",
                "description": "Minimum value.",
                "default": ""
            },
            {
                "name": "max",
                "type": "NUMBER",
                "description": "Maximum value.",
                "default": ""
            },
            {
                "name": "precision",
                "type": "NUMBER",
                "description": "Precision of the number.",
                "default": ""
            }
        ],
        "tool_name": "EntreAPI Faker",
        "category_name": "Data"
    },
    {
        "name": "URL",
        "url": "https://entreapi-faker.p.rapidapi.com/internet
           /url",
```

```
            "description": "Randomly generate a URL.",
            "method": "GET",
            "required_parameters": [],
            "optional_parameters": [],
            "tool_name": "EntreAPI Faker",
            "category_name": "Data"
        }
    ]
}
```

Other Requirements:
Please produce ten queries in line with the given requirements and inputs. These ten queries should display a diverse range of sentence structures: some queries should be in the form of imperative sentences, others declarative, and yet others interrogative. Equally, they should encompass a variety of tones, with some being polite, others straightforward. Ensure they vary in length and contain a wide range of subjects: myself, my friends, family, and company. Aim to include a number of engaging queries as long as they relate to API calls. Keep in mind that for each query, invoking just one API won't suffice; each query should call upon two to five APIs. However, try to avoid explicitly specifying which API to employ in the query. Each query should consist of a minimum of thirty words.

A.3 Prompts for Solution Path Annotation

We use the following prompt when annotating the answer. And we use diversity_user_prompt when generating a new child node, showing the information of previous child nodes. For all the tasks, we additionally provide a "Finish" function to represent the end of a trail.

```
-----------------------------------------------------------
answer_annotation_system_prompt:
You are the Tool-GPT, capable of utilizing numerous tools and
   functions to complete the given task.
1.First, I will provide you with the task description, and your
   task will commence.
2.At each step, you need to analyze the current status and
   determine the next course of action by executing a function
   call.
3.Following the call, you will receive the result, transitioning
   you to a new state. Subsequently, you will analyze your
   current status, make decisions about the next steps, and
   repeat this process.
4.After several iterations of thought and function calls, you will
    ultimately complete the task and provide your final answer.
Remember:
1.The state changes are irreversible, and you cannot return to a
   previous state.
2.Keep your thoughts concise, limiting them to a maximum of five
   sentences.
3.You can make multiple attempts. If you plan to try different
   conditions continuously, perform one condition per try.
4.If you believe you have gathered enough information, call the
   function "Finish: give_answer" to provide your answer for the
   task.
5.If you feel unable to handle the task from this step, call the
   function "Finish: give_up_and_restart".
Let's Begin!
Task description: {task_description}
-----------------------------------------------------------
diversity_user_prompt:
```

```
This is not the first time you try this task, all previous trails
   failed.
Before you generate your thought for this state, I will first show
    you your previous actions for this state, and then you must
   generate actions that is different from all of them. Here are
   some previous actions candidates:
{previous_candidate}
Remember you are now in the intermediate state of a trail, you
   will first analyze the now state and previous action
   candidates, then make actions that is different from all the
   previous.
-----------------------------------------------------------
Finish_function_description:
{
    "name": "Finish",
    "description": "If you believe that you have obtained a result
        that can answer the task, please call this function to
       provide the final answer. Alternatively, if you recognize
       that you are unable to proceed with the task in the
       current state, call this function to restart. Remember:
       you must ALWAYS call this function at the end of your
       attempt, and the only part that will be shown to the user
       is the final answer, so it should contain sufficient
       information.",
    "parameters": {
        "type": "object",
        "properties": {
            "return_type": {
                "type": "string",
                "enum": ["give_answer","give_up_and_restart"],
            },
            "final_answer": {
                "type": "string",
                "description": "The final answer you want to give
                   the user. You should have this field if \"
                   return_type\"==\"give_answer\"",
            }
        },
        "required": ["return_type"],
    }
}
-----------------------------------------------------------
```

B AUTOMATIC EVALUATOR

B.1 DETAILS OF PERFORMANCE METRICS FOR AUTOMATIC EVALUATOR

Here we describe details of computing metrics used to assert automatic evaluators.

Human Agreement To estimate the agreement between the automatic evaluator and humans, we first sample 4 win rate results for each query in human annotations data set from the automatic evaluator. Then we check each sample whether agrees with the major human preference on 600 queries, giving scores of one if agree and zero if disagree. If there are multiple major preference, we give scores of $\frac{1}{n}$ (where n is the count of major preference) if one major preference of the evaluator agree with humans and zero if not. We finally average the averaged scores of the automatic evaluator samples for all query to get the human agreement.

B.2 WIN RATE RULES

We build rules to better evaluate the capacity of utilizing tools for methods, nor the performance of solving particular task. Based on this principle, we have the following rules:

1. If both answers give the none empty ‘final answer‘, check whether the given ‘final answer‘ solves the given query.
   (a) If both answers solve the query, choose one with smaller ‘total steps‘. If ‘total steps‘ are same, choose one answer with better ‘final answer‘ quality.
   (b) If one answer solve while the other not, chose the answer that solve query.
   (c) If both answers failed, check the ‘answer details‘ to choose one with considering following preference:
      i. Check ‘response‘ and prefer more successful tool calling.
      ii. Check ‘name‘ and prefer using more various tool usage.
      iii. Crefer smaller ‘total steps‘.
2. If one give none empty ‘final answer‘ while other not, choose the one give ‘final answer‘.
3. If both failed to give none empty ‘final answer‘, turn to 1.(c) to choose one with better ‘answer details‘.