ToolAlpaca: Generalized Tool Learning for Language Models with 3000 Simulated Cases

Anonymous ACL submission

Abstract

Enabling large language models to utilize real-world tools effectively is crucial for achieving embodied intelligence. Existing approaches to tool learning have either primarily relied on extremely large language models, such as GPT-4, to attain generalized tool-use abilities in a zero-shot manner, or utilized supervised learning to train limited scopes of tools on compact models. However, it remains uncertain whether smaller language models can achieve generalized tool-use abilities without tool-specific training. To address this question, this paper introduces ToolAlpaca, a novel framework designed to automatically generate a diverse tool-use corpus and learn generalized tool-use abilities on compact language models with minimal human intervention. Specifically, ToolAlpaca first automatically creates a highly diversified tool-use corpus by building a multi-agent simulation environment. The corpus contains 3.9k tool-use instances from more than 400 real-world tool APIs spanning 50 distinct categories. Subsequently, the constructed corpus is employed to fine-tune compact language models, resulting in two models, namely ToolAlpaca-7B and ToolAlpaca-13B, respectively. Finally, we evaluate the ability of these models to utilize previously unseen tools without specific training. Experimental results demonstrate that ToolAlpaca achieves effective generalized tool-use capabilities comparable to those of extremely large language models like GPT-3.5, demonstrating that learning generalized tool-use ability is feasible for compact language models.

1 Introduction

Embodied intelligence, the ability to meaningfully interact with the environment, stands as a core attribute of advanced cognitive systems and a crucial advancement in artificial intelligence. The ability to create and use tools has expanded human beings' physical capabilities to interact with environments and augmented cognitive functions. Such evolutionary milestone has not only broadened our range of physical actions, but also brought about transformative changes in our problem-solving abilities and innovative thinking. The pursuit of incorporating tool-use capabilities into artificial intelligence holds great significance in advancing the development of general intelligent systems.

Recent advancements in enhancing large language models (LLMs) such as GPT-4 (OpenAI, 2023) with tool-use abilities have made significant progress in this area. These models have shown their ability to effectively employ external tools through integrated plugins, thereby expanding their versatility and enhancing the precision and quality of their outputs. Unfortunately, due to a lack of understanding of how existing large language models acquire the general tool-use capability, currently compact language models still do not possess such general ability. Consequently, substantial research efforts are dedicated to fine-tuning smaller language models to acquire the capacity for tool usage (Komeili et al., 2022; Parisi et al., 2022; Schick et al., 2023) on a limited range of tools, which lacks the ability to generalize to unseen tools. This discrepancy between the generalized tool-use abilities of larger models and the more constrained capabilities of compact models presents an intriguing question: *Can these compact language models learn to generalize their tool-use abilities, thus enabling interaction with a broader spectrum of tools?*

In this paper, we explore whether it is feasible for compact language models to learn generalized tool-use abilities. Intuitively, previous studies have demonstrated the possibility of equipping compact language models with generalized instruction-following abilities by fine-tuning them on diversified instruction datasets (Taori et al., 2023; Zhou et al., 2023). Therefore, a promising strategy for equipping language models with generalized tool-use abilities involves fine-tuning them on a highly-diversified tool-use corpus. Unfortunately,

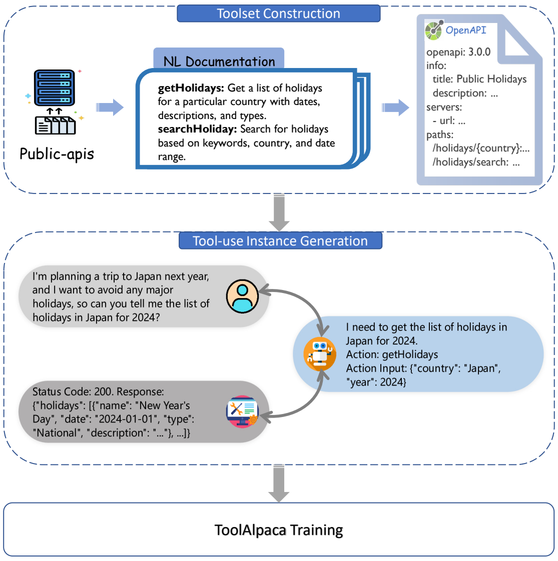

Figure 1: A high-level overview of ToolAlpaca, consisting of three components: (1)Toolset construction, where structured documentation for each tool is generated based on the brief introductions provided by public-apis. (2) Tool-use instance generation via multi-agent simulation. (3) ToolAlpaca model training, which involves fine-tuning language models on generated tool-use corpus to get ToolAlpaca.

such a corpus is currently unavailable, especially where API access is restricted due to confidentiality. This scarcity stems from several crucial factors. Firstly, in environments with limited API access, directly accessing APIs for data generation, like ToolLLM (Qin et al., 2023b), is challenging and poses privacy or security risks, significantly limiting language models' application in these scenarios. Secondly, real-world tool usage often involves complex interactions between the language model, users, and tools, greatly increasing the difficulty and effort needed to create a broad range of tool instances at scale. These factors substantially hinder the development of a diversified tool-use corpus for efficient language model training.

To this end, we propose a framework named ToolAlpaca, which is designed to automatically create a diverse and well-structured toolset for LLMs and generate multi-turn complex tool-use instances for generalized tool learning. The overall structure of ToolAlpaca is shown in Figure 1. Specifically, ToolAlpaca gathers a substantial amount of brief introductions of potentially valuable tools from the internet. It's important to note that there is no requirement for these tools' APIs to be accessible or for them to possess structured documentation directly usable by LLMs. This ensures our data generation approach remains applicable even when APIs are inaccessible or documentation is incomplete. Building on this foundation, ToolAlpaca employs the generative capacity of LLMs by taking the brief introduction of relevant tools as input and prompts the model to produce detailed, structured documentation for each tool. By employing this methodology, ToolAlpaca has collected more than 400 tool descriptions spanning 50 categories. Each tool is uniformly represented using a standardized documentation format. Subsequently, in order to acquire tool-use instances involving the aforementioned tools, we have designed a simulation environment aimed at emulating the multi-step interactions among language models, users, and tools. Specifically, we utilize LLMs to simulate the interactions between the model, users, and the APIs of the tools by leveraging LLMs to serve as different kinds of agents. In this way, our simulation environment can generate a substantial volume of tool-use instances without any manual intervention. Consequently, we have crafted an inclusive tool-use dataset that comprises 3.9k instances, effectively showcasing the practical application of over 400 distinct tools.

To verify whether our corpus can empower compact language models with the generalized tool-use ability, we conduct experiments to train ToolAlpaca model on Vicuna (Chiang et al., 2023), a representative compact language model, and subsequently evaluate its performance on various unseen tools. Through machine evaluation with GPT-4, we find that ToolAlpaca can effectively use numerous unseen tools, ranging from real-world APIs to multi-modal tools, and it exhibits competitive performance with GPT-3.5. Furthermore, we investigate the effect of diversity. It is observed that even with the same number of instances, the model trained on more varied toolsets will achieve better performance. This underscores that diversity is a pivotal factor for ToolAlpaca to generalize tool learning with 3000 simulated cases.

In summary, the main contributions of this paper are:

- To the best of our knowledge, this paper is the first work that verifies the feasibility of equipping compact language models with generalized tool-use capacities, showing that they can be applied in real-world scenarios even when trained exclusively on simulated data.

- This paper presents ToolAlpaca, a simple framework for the automated generation of tool-use corpus, applicable in special scenarios such as restricted API access and the absence of structured documentation.

- We create a diverse tool-use corpus containing 3.9k tool-use instances from more than 400 tools across 50 distinct categories. It serves as a solid foundation for compact language models to acquire generalized tool-use ability.

2 Related Work

Tool Use The utilization of external tools in LLMs has emerged as a rapidly growing research area (Mialon et al., 2023; Qin et al., 2023a). Current approaches can be divided into two distinct categories. The first category leverages the capabilities of LLMs, prompting them to interact with various tools, ranging from highly specialized ones such as code interpreters (Gao et al., 2022; Chen et al., 2022), search engines (Yao et al., 2022), retrieval models (Khattab et al., 2023) and AI models (Shen et al., 2023; Lu et al., 2023), to more versatile toolsets (Qin et al., 2023a; Li et al., 2023; Song et al., 2023). Large language models have already demonstrated robust generalization capabilities in tool usage and enable to equip numerous unseen tools via prompting. In contrast, the second category concentrates on enhancing the tool-specific usage capabilities of compact language models through fine-tuning with datasets specifically designed for the specialized tools (Parisi et al., 2022; Schick et al., 2023; Xu et al., 2023). Concurrent with our work, GPT4Tools (Yang et al., 2023) fine-tuning compact models to incorporate multi-modal tools, which concentrates on a set of quite similar multi-modal tools. ToolLLM (Qin et al., 2023b) facilitates language models to master massive APIs. However, their data collection strategy requires the prior accumulation of massive authentic APIs, which requires manual efforts to obtain and verify. Despite their effectiveness, the domain of generalized tool-use abilities in compact language models remains largely unexplored upon the accomplishment of this paper. This study aims to bridge this research gap by automatically constructing a diverse dataset on tool utilization that encompasses various tool-use scenarios.

LLMs for Data Generation Many research studies have employed LLMs for data generation, focusing on various tasks such as question answering (Wang et al., 2021; Agrawal et al., 2022; Chen et al., 2023), semantic similarity predictions (Schick and Schütze, 2021), and instruction tuning (Honovich et al., 2022; Wang et al., 2023). Furthermore, in the context of tool use, several works (Schick et al., 2023; Patil et al., 2023; Yang et al., 2023) have already employed model-synthesized data to enhance specific tool-use capabilities. However, the generation of generalized tool-use data poses more significant challenges, as it involves extensive and diverse tools and more intricate multi-turn interactions.

3 Diversified Tool-use Corpus Generation via Multi-agent Simulation

In this section, we introduce ToolAlpaca, a multi-agent simulation framework designed to generate a diversified tool-use corpus with minimal human intervention. As shown in Figure 1, our framework consists of two stages:

1. Toolset Construction. This step aims to construct a collection of tools and represent them using a standardized format as *{name, introduction, description, function documentation, OpenAPI specification}*. Specifically, we initiate the process by sourcing tool names and introductions from the internet and then utilize LLMs to enrich them with structured documentation that delineates the functionality and usage of each tool. In this way, we can construct a diverse and structured toolset that closely resembles real-world scenarios.

2. Tool-use Instance Generation. Given the toolset, this step's objective is to generate tool-use instances within a simulation environment automatically. This environment is constructed with three distinct virtual agents, each embodied by a LLM: the user, the tool executor, and the assistant. Through the multi-turn interplay among these agents, we can generate tool-use instances that reflect real-world tool-use scenarios. Each tool-use instance consists of three elements: *{the user's instructions, the actions and corresponding tool outputs, final response}*.

3.1 Diverse Toolset Construction

This section describes how to construct a diverse toolset and represent them in a uniform format. The process initiates with the accumulation of an extensive API collection from the internet, reflecting

Public Holidays

Introduction: Data on national, regional, and religious holidays via API
Description: The Public Holidays API is a user-friendly interface that provides comprehensive information on national, regional, and religious holidays around the world.
The API's key features are:
1) Get a list of holidays for a particular country with dates, descriptions, and types.
2) Retrieve detailed information on a specific holiday, including its history, purpose, and traditions.
3) Obtain information on public holidays for a specific year, month, or day.
Function Documentation:
getHolidays: Get a list of holidays for a particular country with dates, descriptions, and types.
Parameters: {"country": "Required. String. The country for which holidays are to be retrieved.", "year": "Integer. The year for which holidays are to be retrieved."}
Output: A list of holidays with their dates, descriptions, and types for the specified country, year, month, and day.
searchHoliday: Search for holidays based on keywords, country, and date range.
getHolidayDetails: Retrieve detailed information on a specific holiday, including its history, purpose, and traditions.
OpenAPI Specification:
- GET /holidays/{country} - GET /holidays/{holidayId}/details - GET /holidays/search

Figure 2: An instance of a tool documentation, composed of five essential parts: *name, introduction, description, function documentation, OpenAPI specification*.

real-world tool usage scenarios. Given the rudimentary descriptions and lack of uniform representation in these APIs, we further leverage the generative capabilities of LLM to create comprehensive documentation for each tool. This documentation assists language models in understanding the functionality and usage of each tool. Subsequently, we adhere to OpenAPI standards to generate a uniform specification for each API, enabling automated computer invocation and facilitating subsequent tool execution simulation. In this way, each tool can be represented as a quintuple *{name, introduction, description, function documentation, OpenAPI specification}*. Figure 2 provides an example, where the name, description, and introduction offer basic information and the purpose of the public holiday tool, the function documentation provides the functionality, inputs and outputs of various functions (*getHolidays, searchHolidays, getHolidayDetails*) contained within the tool, and the OpenAPI Specification provides a more comprehensive and structured document. The detailed construction steps are elaborated as follows.

Tool Collection. Various tools are commonly utilized by human beings, typically manifested in the form of web-based APIs. To facilitate the utilization and discovery of these APIs, a plethora of repositories exist on the Internet, aggregating a vast collection of practical and commonly used APIs. Consequently, this step leverages the representative API repository, public-apis [1], as our target toolset. This repository encompasses over 1400 APIs spanning more than 50 distinct categories. From this, we collect the name and introduction of each tool.

[1] https://github.com/public-apis/public-apis

Documentation Generation. To enhance the language model's comprehension of tools' functionalities and usage, this step employs LLMs to automatically generate documentation for each tool, including its description and function documentation. Specifically, given the tool's name and introduction, we leverage the powerful generative capabilities of LLMs, such as ChatGPT, and construct corresponding prompts to generate the description and function documentation for each tool. As illustrated in Figure 2, for the tool description, we expand the introduction to provide a general overview of the tool's purpose and functionality, enabling the language model to understand the appropriate task scenarios for using the tool. For the function documentation, we prompt LLMs to generate more specific functions within the scope described in the description, including their respective input and output parameters. It ensures that the LLM comprehends how to use the tool's different functionalities. By employing this approach, we lay a solid foundation for subsequent user instruction generation and the creation of the simulated tool-use environment.

OpenAPI Specification Generation. Finally, we prompt LLM to generate a more formalized and structured representation for each tool in the form of OpenAPI Specification. OpenAPI Specification (OAS) defines a standard and language-agnostic interface for describing APIs, including information about endpoints, expected input/output types, and possible error responses. OAS provides consistency and readability for both humans and machines, making it an ideal choice for our tool-use corpus. This comprehensive documentation serves as the foundation for simulating tool execution in the subsequent stages of our framework.

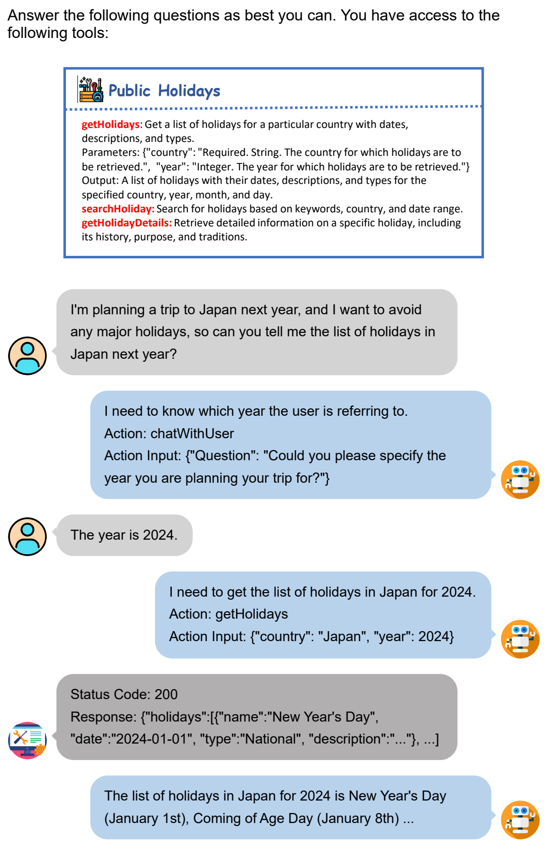

Figure 3: An illustration of the tool-use instance generation process within the simulation environment. The user agent initiates the sequence by providing an instruction. The assistant agent then interprets the instruction and engages in a multi-turn interaction with the user and the tool executor until a suitable response is generated.

In this way, we construct a diverse, uniformly represented toolset, which provides a solid foundation for the multi-agent simulation environment building and further tool-use corpus generation.

3.2 Automatic Tool-use Instances Generation

Given the toolset, this section describes how to automatically construct a tool-use corpus, so that language models can be trained to acquire generalized tool-use ability. Specifically, as depicted in Figure 3, each tool-use instance can be represent as a triple *{Instruction, Actions, Response}*:

- **Instruction:** A user query that requires tool assistance for resolution. *"... so can you tell me the list of holidays in Japan next year?"* serves as an instruction in our example.

- **Actions:** The process of resolving an instruction may involve executing multiple actions in a specific order. Following React (Yao et al., 2022), each action is represented by a tuple that includes the thought, the function name, the input parameters, and the corresponding tool response. For example, as shown in Figure 3, the tuple *("I need to get the list of holidays in Japan for 2024.", "getHolidays", {"country": "Japan", "year": 2024}, "Status Code: 200 Response:...")* represents an action.

- **Response:** This refers to the model's conclusive response after the multi-turn interaction, integrating the tool responses to provide a comprehensive solution to the user instruction. For instance, the response in our example is: *"The list of holidays in Japan for 2024 is ..."*.

However, constructing a diverse and authentic tool-use dataset is a challenging task. Firstly, the wide variety within our toolset makes it impracticable to manually draft instructions for each tool. Given the vast array of tools, spanning from recreational to professional domains, and the fact that the construction of instructions relies on understanding the functionality and potential use cases of the tools, the burden of manual annotation becomes overwhelming. Secondly, tool usage in real-world scenarios often involves a multi-round iterative process of trial and error, making the automated construction of tool-use instances that reflect real situations highly challenging.

To this end, we design a simulation environment to generate a tool-use corpus, encompassing three virtual agents: the user, the assistant, and the tool executor. Tool-use instances are generated through the interplay among these agents. Specifically, each agent is simulated by a large language model with a specific prompt. The distinct roles of each agent are detailed as follows:

- **User Agent** is designed to mimic the tool user, with its functionalities encompassing: (1) drafting task instructions for the current tool based on its function documentation; (2) responding to the assistant's queries based on the current interaction context, providing essential information that might be missing from the initial instruction. For each functionality, we construct corresponding prompt templates to guide LLMs to generate appropriate outputs. Moreover, to ensure diversity in task instructions, we have employed various prompts to generate instructions of different formats, including commands, questions, and others. Leveraging the large model's proficiency across virtually all domains, this approach enables the generation of high-quality and diversified instructions. This effectively addresses the previously mentioned issues with manual annotation.

- **Assistant Agent** is designed to simulate an assistant with tool utilization capabilities. It receives instructions from the user agent and determines the subsequent actions. This involves choosing the appropriate tools and functions, generating commands for the tool executor, and summarizing the interaction to generate the final response. As shown in Figure 3, following ReAct (Yao et al., 2022), we employ a (thought, action, observation) format template to guide LLM in accomplishing these tasks.

- **Tool Executor Agent** is constructed to emulate the execution of tools, receiving requests from the assistant agent and generating responses based on the tool's predefined functionalities. Specifically, after conducting format and parameter checks on the assistant's requests, these requests are converted into network request formats. Then the tool executor prompts LLM with the tool's OpenAPI specification and the assistant's requests to generate simulated execution results. Leveraging LLMs' robust simulation and generation capabilities, we mitigate the intricacies involved in constructing actual API calls. This method is notably flexible, making it suitable for a wide range of scenarios, including those with restricted API usage. Its accuracy and effectiveness have been empirically validated, as demonstrated in the following section.

Given the above agents, tool-use cases are generated through multiple rounds of interaction between them. Initially, the user agent generates instructions based on the tool information. Subsequently, the assistant agent selects an appropriate action and its corresponding input and awaits simulation execution and response from the tool executor. This iterative procedure of action selection and tool response collection continues until the assistant agent deems it has gathered sufficient information to respond to the user's instructions. Through this multi-agent interaction, we can simulate realistic tool-use scenarios and generate comprehensive and diversified tool-use instances.

4 ToolAlpaca Corpus

4.1 Construction Details

Leveraging the aforementioned multi-agent simulation framework, we have constructed the ToolAlpaca corpus. Specifically, the process begins with randomly selecting 500 APIs from the public-apis repository. Subsequently, we utilize ChatGPT to generate more comprehensive documentation, resulting in a varied and well-structured toolset. Within our simulation environment, we leverage ChatGPT to serve as the agents to generate diversified tool-use instances.

As a result, we automatically construct an extensive and diversified tool-use corpus. As shown in Table 1, it encompasses 426 distinctive tools from 50 categories, totaling 3938 instances. In the following sections, we will analyze the diversity and quality of our corpus.

statistics	
# of Tool Categories	50
# of Tools	426
# of Instance	3,938
# of single function call	2,512
# of multiple function calls	1,426
avg. functions per tool	4.85
avg. steps	1.66
avg. instruction length	23.42
avg. output length	36.19

Table 1: Statistics of ToolAlpaca corpus.

4.2 Diversity

Diversity is pivotal for large models to acquire generalized capabilities and adapt to unseen scenarios (Wang et al., 2023). ToolAlpaca corpus demonstrates diversity in two aspects:

- **Toolset.** As outlined in Table 1, our toolset demonstrates diversity in multiple aspects: (1) The toolset encompasses 50 categories of tools, ranging from common categories, such as jobs and news, to specialized categories like blockchain and finance. (2) Each tool provides an average of five functions, highlighting the comprehensiveness of its capabilities. (3) The range of function inputs varies from simple to complex scenarios, including arrays and objects, further enhancing the richness and complexity of our toolset.

- **Instances.** The instances within the ToolAlpaca corpus demonstrate diversity in terms of instruction, function calls, and error handling.

Specifically, we employ a variety of prompts during instruction generation to stimulate the language model in producing diverse instructions. Additionally, our dataset contains about 1.5k instances that require multiple function invocations, further underscoring the comprehensiveness of our dataset. Furthermore, our data adequately reflects the potential errors that may be encountered in authentic tool usage scenarios, encompassing instances that involve various types of errors, such as invalid actions and incorrect parameters.

4.3 Quality

To evaluate the quality of ToolAlpaca corpus, we randomly sample 100 instances and engage a human annotator for assessment. The evaluation tests the solvability of the instructions generated by the user agent, the precision of the output from the tool executor agent, and the accuracy of the assistant agent's actions and responses. As illustrated in Table 2, we observe that the metrics for assessing the capabilities of the three agents all exceed 80%. This substantiates that each agent is proficient in their respective roles, demonstrating the reliability of data constructed based on simulation and affirming the decent quality of our dataset.

Quality	Yes%
solvability of instructions	88%
effectiveness of Tool agent's response	92%
accuracy of action sequences and final output	80%

Table 2: Data quality review for ToolAlpaca corpus.

5 Experiment

In this section, we investigate whether a set of simulated data can empower compact language models to acquire generalized tool-use capabilities. To verify this, we conduct zero-shot experiments on various tools that have not appeared in the training set, ranging from simulated tools, real-world tools, to out-of-dataset multi-modal tools. Furthermore, we investigate how the diversity of the toolset impacts the generalized tool-use ability of LLMs.

5.1 Experimental Settings

Training We fine-tune Vicuna models (Vicuna-7B and Vicuna-13B) on ToolAlpaca corpus. The fine-tuning process consists of three epochs, with a batch size of 128 and a learning rate of 2e-5.

Evaluation To measure the generalized tool-use ability of the language model, we create an evaluation dataset through our data generation framework and manually annotate the data. This evaluation dataset consists of two subsets: (1) a simulated subset that includes 10 simulated tools and 100 instances, which were not part of the training toolset; (2) a real-world subset comprising 11 real-world APIs and 114 cases from various domains, designed to assess the divergence between our simulated data and real-world data.

To evaluate the models, we utilize GPT-4 for machine evaluation across all experiments, with an additional manual evaluation conducted specifically for the simulated subset. We prompt GPT-4 with the tool documentation and the standard answer from the human annotator and expect it to evaluate the performance in the following aspects:

- **Procedure:** This metric evaluates the model's proficiency in accurately selecting suitable actions, utilizing correct parameters, and avoiding redundant actions.
- **Response:** This criterion measures whether the final response can satisfy the instruction.
- **Overall:** This metric evaluates the whole process, requiring the correctness of procedure and response.

5.2 Results

Effectiveness of ToolAlpaca corpus. Table 3 presents the main results from the simulated set, evidencing that fine-tuning on ToolAlpaca corpus can foster generalized tool learning for compact models. Without fine-tuning on our corpus, Vicuna models demonstrate constrained tool-use capabilities, with the human accept rate of 16 and 25, respectively. These statistics emphasize the existing compact models' insufficiency in achieving the generalized tool-use capacity like larger models. Nevertheless, our ToolAlpaca models attain 73 (+57) and 75 (+50) accept rates, respectively. ToolAlpaca-13B even achieves comparable performance to GPT-3.5. This evidences the feasibility of instilling generalized tool-use capabilities into compact language models by only training on 3000 simulated instances generated by our framework.

Generalization on real-world tools. The effectiveness of our corpus is further validated through real-world APIs, demonstrating that simulation

Model	Simulated Tools				Real-world APIs		
	Procedure	Response	Overall	Human	Procedure	Response	Overall
GPT-3.5 (OpenAI, 2022)	77.0	85.0	75.0	79.0	75.4	80.7	72.8
Vicuna-7B (Chiang et al., 2023)	19.0	21.0	17.0	16.0	7.9	11.4	7.9
Vicuna-13B (Chiang et al., 2023)	17.0	31.0	16.0	25.0	13.2	16.7	12.3
ToolLLM (Qin et al., 2023b)	-	-	-	-	46.5	50.0	43.9
ToolAlpaca-7B	63.0	69.0	60.0	73.0	63.2	57.9	55.3
ToolAlpaca-13B	70.0	73.0	70.0	75.0	66.7	67.5	61.4

Table 3: Evaluation results on unseen simulated tools and real-world APIs. We can observe that after training on our corpus, ToolAlpaca's performance significantly surpasses that of the Vicuna model, reaching comparable performance with GPT-3.5.

serves as an exceptionally efficient data collection method. Table 3 exhibits the performance of ToolAlpaca on the real-world test set, where it achieves an overall accuracy of 55.3 and 61.4, respectively, significantly surpassing the performance of Vicuna models and. This suggests that training on simulated data can indeed adapt to real-world tool usage scenarios. We attribute this to the current LLMs' robust simulation capabilities, which provide compelling evidence for future simulation-based data construction. Furthermore, ToolAlpaca even surpass ToolLLM (Qin et al., 2023b), which is trained on more than 100k instances. Further analysis reveals that ToolAlpaca is more adept at handling repeated API calls, such as "give me 5 jokes".

Model	SR_t	SR_{act}	SR_{args}	SR
GPT-3.5	99.5	99.5	91.5	91.5
Vicuna-13B	84.4	43.7	46.7	26.2
GPT4Tools	98.2	97.0	92.2	90.6
ToolAlpaca-13B*	-	95.5	85.3	83.7

Table 4: Evaluation results on unseen tools from GPT4Tools. Metrics: successful rate of thought, action, arguments, and the entire instance. *: As our dataset does not include data not involving tool use, we exclude 50 out of 652 test cases that do not involve tool usage.

Moreover, to evaluate ToolAlpaca's generalization on out-of-dataset scenarios, we conduct experiments on GPT4Tools(Yang et al., 2023) test set, which encompasses 8 multi-modal tools. As shown in Table 4, ToolAlpaca, trained on merely 3.9k cases, gets 83.7 success rate on out-of-dataset evaluation, which is close to GPT4Tools, trained on 71k instances constructed with the same process. This observation indicates that LLMs can invoke out-of-dataset tools after training on ToolAlpaca corpus. We speculate that the performance may be attributed to the diversity of instances and toolset, and we delve into it in the subsequent experiment.

Impact of diversity. The diversity of the dataset is crucial for the generalization of tool learning. To investigate this, we maintain the number of instances and construct datasets on 10, 40, 100, and 400 tools, respectively. Subsequently, we fine-tune Vicuna-13B on these datasets with the same experimental settings and utilize GPT-4 to evaluate the validation set. As shown in Figure 4, as the diversity of the toolset increases, the performance on the validation set gradually improves. Specifically, training with a dataset of 10 different tools resulted in mere 51 overall accept rate. In contrast, when the variety of tools increases to 400 and keeps the number of instances, the performance escalates to 70. This finding highlights the significant role of toolset diversity in generalizing tool learning. This provides valuable insight for the construction of datasets for generalized ability learning.

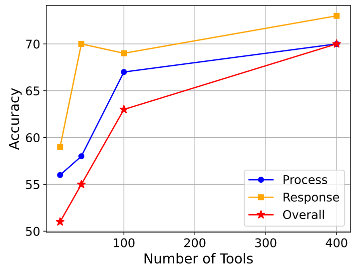

Figure 4: Performance variation with the increasing toolset diversity.

6 Conclusion

In this paper, we introduce ToolAlpaca, an automated framework designed to improve the generalized tool-use capability of language models. Specifically, we first create a comprehensive corpus spanning a broad range of tools with various usage instances. Subsequently, this corpus serves as the basis for fine-tuning compact language models, leading to the generation of the ToolAlpaca models. Experimental results indicate that ToolAlpaca performs comparably to GPT-3.5 in generalized tool-use scenarios. This finding not only substantiates the potential of our data generation framework but also highlights the feasibility of mastering generalized tool use in compact-size models.

Limitations

ToolAlpaca can automatically generate a diversified tool use corpus to empower compact language models with generalized tool-use capabilities. In this paper, we construct 3.9k simulated instances to verify the effectiveness of our framework. However, the variation in model performance on larger scale simulated data still requires exploration. Additionally, the tool usage in the dataset focuses on different functions within the same tool type. In future work, we will explore the construction of instances that involve calling multiple different types of tools through simulation generation.

References

Priyanka Agrawal, Chris Alberti, Fantine Huot, Joshua Maynez, Ji Ma, Sebastian Ruder, Kuzman Ganchev, Dipanjan Das, and Mirella Lapata. 2022. Qameleon: Multilingual qa with only 5 examples.

Mingda Chen, Xilun Chen, and Wen tau Yih. 2023. Efficient open domain multi-hop question answering with few-shot data synthesis.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks.

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. 2023. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2022. PAL: Program-aided Language Models. *ArXiv preprint*, abs/2211.10435.

Or Honovich, Thomas Scialom, Omer Levy, and Timo Schick. 2022. Unnatural instructions: Tuning language models with (almost) no human labor.

Omar Khattab, Keshav Santhanam, Xiang Lisa Li, David Hall, Percy Liang, Christopher Potts, and Matei Zaharia. 2023. Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive nlp.

Mojtaba Komeili, Kurt Shuster, and Jason Weston. 2022. Internet-augmented dialogue generation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8460–8478, Dublin, Ireland. Association for Computational Linguistics.

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

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2023. Chameleon: Plug-and-play compositional reasoning with large language models.

Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. 2023. Augmented language models: a survey.

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

OpenAI. 2023. Gpt-4 technical report.

Aaron Parisi, Yao Zhao, and Noah Fiedel. 2022. TALM: tool augmented language models. *ArXiv preprint*, abs/2205.12255.

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

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, Yi Ren Fung, Yusheng Su, Huadong Wang, Cheng Qian, Runchu Tian, Kunlun Zhu, Shihao Liang, Xingyu Shen, Bokai Xu, Zhen Zhang, Yining Ye, Bowen Li, Ziwei Tang, Jing Yi, Yuzhang Zhu, Zhenning Dai, Lan Yan, Xin Cong, Yaxi Lu, Weilin Zhao, Yuxiang Huang, Junxi Yan, Xu Han, Xian Sun, Dahai Li, Jason Phang, Cheng Yang, Tongshuang Wu, Heng Ji, Zhiyuan Liu, and Maosong Sun. 2023a. Tool learning with foundation models.

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

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

Timo Schick and Hinrich Schütze. 2021. Generating datasets with pretrained language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6943–6951, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

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

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

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

Shuohang Wang, Yang Liu, Yichong Xu, Chenguang Zhu, and Michael Zeng. 2021. Want to reduce labeling cost? GPT-3 can help. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 4195–4205, Punta Cana, Dominican Republic. Association for Computational Linguistics.

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

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

Rui Yang, Lin Song, Yanwei Li, Sijie Zhao, Yixiao Ge, Xiu Li, and Ying Shan. 2023. Gpt4tools: Teaching large language model to use tools via self-instruction.

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*, abs/2210.03629.

Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, Susan Zhang, Gargi Ghosh, Mike Lewis, Luke Zettlemoyer, and Omer Levy. 2023. Lima: Less is more for alignment.

A Implementation Details

In this section, we show the details of prompt templates in ToolAlpaca. Figure 5, Figure 6, and Figure 7 delineate the prompts employed during toolset construction. Figure 8 and Figure 9 illustrate the corresponding prompts for user agent's two responsibility, generating user instructions and providing missing information. The prompts designed for to the assistant agent and the tool executor agent are detailed in Figure 10 and Figure 11. *"${...}"* within the prompts are placeholders, will be replaced by real variables during the generation process.

B Experiment Details

B.1 Hyperparameters

The fine-tuning configuration for ToolAlpaca is recorded in Table 5.

Hyperparameters	Value
optimizer	AdamW
learning rate	2e-5
weight decay	0.0
warmup ratio	0.03
lr scheduler type	cosine
num train epochs	3
batch size	128
max length	2048

Table 5: The fine-tuning configuration for ToolAlpaca.

B.2 Evaluation Dataset Details

To evaluate the generalized tool-use ability, we construct the evaluation dataset via our framework, which consists two subsets: a simulated subset with 10 simulated tools and 100 instances, a real-world subset with 11 real-world APIs and 114 instances. The toolset used in the evaluation datset is detailed in Table 6.

B.3 Evaluation Prompt

Following the evaluation method used by Vicuna (Chiang et al., 2023), we use GPT-4 as our evaluator. The evaluation prompt is shown in Figure 12.

B.4 Case Study

Through training on a set of diverse simulated tool-use instances, ToolAlpaca can equip various tools, even real-world APIs, some selected cases are shown in Figure 13, Figure 14 and Figure 15.

Name	Category	Introduction
Simulated Tools		
Axolotl	Animals	Collection of axolotl pictures and facts
AniAPI	Anime	Anime discovery, streaming & syncing with trackers
AbuseIPDB	Anti-Malware	IP/domain/URL reputation
Améthyste	Art & Design	Generate images for Discord users
Auth0	Authentication & Authorization	Easy to implement, adaptable authentication and authorization platform
Abstract Public Holidays	Calendar	Data on national, regional, and religious holidays via API
1Forge	Currency Exchange	Forex currency market data
A Bíblia Digital	Books	Do not worry about managing the multiple versions of the Bible
Apache Superset	Business	API to manage your BI dashboards and data sources on Superset
Lob.com	Data Validation	US Address Verification
Real-world APIs		
Nager.Date	Calendar	Public holidays for more than 90 countries
airportsapi	Transportation	Get name and website-URL for airports by ICAO code
AviationAPI	Transportation	FAA Aeronautical Charts and Publications, Airport Information, and Airport Weather
chucknorris.io	Entertainment	JSON API for hand curated Chuck Norris jokes
Random Useless Facts	Entertainment	Get useless, but true facts
apilayer weatherstack	Weather	Real-Time & Historical World Weather Data API
Free Dictionary	Dictionaries	Definitions, phonetics, pronounciations, parts of speech, examples, synonyms
WolframAlpha	Machine Learning	Provides specific answers to questions using data and algorithms
Fruityvice	Food & Drink	Data about all kinds of fruit
Cataas	Animals	Cat as a service (cats pictures and gifs)
CurrencyBeacon	Currency Exchange	Real-time and historical currency rates JSON API

Table 6: Tools used in our evaluation dataset.

Toolset Construction - Description Prompt

```
I will provide the API's name, link, and brief introduction. You need
to generate a detailed description for the API.

Guidelines:
1. Write a general overview of the API's purpose and functionality.
2. List and briefly describe all features provided by the API,
ensuring each feature has a clear and distinct purpose with low
coupling between them.
3. Use clear, concise language and avoid jargon, keeping the
description under 300 tokens in length.

<API>
Name: AdoptAPet
Link: https://www.adoptapet.com/public/apis/pet_list.html
Introduction: Resource to help get pets adopted
Description: The Adopt-a-Pet.com API (Application Programming
Interface) is a series of tools that allows partners to use
Adopt-a-Pet.com's pet adoption features and pet data in other
applications. It provides the following features: 1) Retrieve the
list of available pets for the shelter or rescue. 2) Retrieve the
details for a specific pet.
</API>

<API>
Name: ${name}
Link: ${link}
Introduction: ${introduction}
Description:
```

Figure 5: Description generation prompt.

Toolset Construction - Function Documentation Prompt

```
You are given the name, link, and description of an API. Your task is
to create a comprehensive introduction for this API.

Guidelines:
1. For each function of the API, detail its purpose, input
requirements, and output results.
2. For function input, present it in JSON format. Each key should be
the input parameter's name, and its value should be a string
indicating whether it's required or not, its type, and a brief
description, such as "Required/Optional. Integer. {some
description}".
3. Do not design functions that return excessive data, such as
'getAllXxx'. If such a function is necessary, incorporate input
parameters to limit, filter, or paginate the results.
4. Limit the number of functions generated. Only generate functions
based on the API Description. Do not create unnecessary functions
that overcomplicate the API.
5. If any API function requires fields that are not directly
accessible to the users (like IDs, internal codes, etc.) as inputs,
there must be corresponding methods for users to retrieve these
values, such as through 'search' or 'list' functions.
6. Output with the following format:
    {index}. Name: {function name, follow the camel case naming
    convention.}
    Description: {function short description}
    Input: {function input, presented as a single line without any
    formatting}
    Output: {function output, describe all the information that this
    function will return}

Begin!
Name: ${name}
Link: ${link}
Description: ${description}
Functions:
```

Figure 6: Function documentation generation prompt.

Toolset Construction - OpenAPI Specification Prompt

```
Please generate API documentation that conforms to the OpenAPI
Specification for the provided API, following these guidelines:
1. Name the API with the 'title' field in the 'info' section, and
include a 'version' and 'description' field to describe the API's
purpose and functionality succinctly.
2. Exclude the 'tags' field in the specification.
3. For each function:
  - Design an endpoint, adhering to its definition and input/output
  requirements.
  - Use the function's name in the 'operationId' field. Decompose the
  description of the function into appropriate fields.
  - For the endpoint's input, provide additional details in the
  'parameters' section to complement the function's input
  requirements. For instance, use 'enum' to specify valid parameter
  values.
  - Generate a detailed model for each endpoint's response, including
  status codes and structured return values. This should base on the
  function's output description, detailing each field whenever
  possible.
  - If an endpoint's input includes fields unknown to the user, like
  IDs, these fields must be included in the responses of relevant
  'search', 'list', or similar endpoints.
4. Include a 'description' field for each input parameter and
'requestBody' in the operation object to explain their purpose and
usage.
5. Ensure the OpenAPI Specification is comprehensive, capturing all
functions mentioned in the API Introduction.
6. For parameters/schemas with a 'type' of 'object', you must include
their properties in the specification.

Name: ${name}
Link: ${link}
Description: ${description}
Functions: ${functions}
OpenAPI Spec(Format with JSON, indent=1):
```

Figure 7: Openapi specification generation prompt.

User Agent - Instruction Prompt

```
Imagine that you are a user who wants to utilize the features
provided by various APIs in your daily life. Your task is to come up
with realistic scenarios for using these APIs and express them as
natural language instructions, as if you were asking a friend or
assistant for help.

Please follow these guidelines:
1. The instructions should be 1 to 2 sentences long. Use a mix of
interrogative sentences, first-person statements, imperative
sentences, and other structures that convey a request. Aim for
diversity in your instructions.
2. Do not mention the API's name in your instructions.
3. Your instructions should only involve the features provided by
these APIs. The instructions that need multiple times of API call is
better.
4. Generate 10 diverse instructions.
5. Use specific nouns and real-world examples from various domains,
such as entertainment, sports, or technology. Avoid using any form of
placeholder or generic phrases, such as "this xxx", "a xxx" or "a
specific xxx", and provide concrete details instead.
6. Try not to repeat the verb for each instruction to maximize
diversity.
7. Ensure diversity in language by combining questions with
imperative statements and other structures that convey a request.

<API>
Name: ${name}
Description: ${description}
API Functions: ${functions}
</API>

Based on the API provided above, generate 10 natural language
instructions with specific examples and diverse language, following
the guidelines.
```

Figure 8: User agent prompt 1 for instruction generation.

User Agent - Additional Information Prompt

```
As a user, you ask the AI assistant some questions, but the assistant
believes you have missed crucial information. Please respond to the
AI assistant's inquiries with specific and direct answers using text
or formatted text. Avoid using placeholders. If a file is necessary,
please provide the contents of the file in your response. your
response should startswith "[User]:".
${interaction_history}
```

Figure 9: User agent prompt 2 for providing missing information.

Assistant Agent

```
Your task is to answer the user's question using available tools. The
user cannot see or use the tools themselves, nor can they know the
process of your tool usage. Provide all necessary information in the
"Final Answer" field. Do not make up any information. If required
parameters are missing, use the "getDetails" tool to ask the user for
them.
You have access to the following tools:

${tool_list}

Use the following format:
Question: the input question you must answer
Thought: you should always think about what to do
Action: the action to take, should be one of [${tool_names}].
Action Input: the input to the action, must be in JSON format. All of
the action input must be realistic and from the user.
Observation: the result of the action
... (this Thought/Action/Action Input/Observation can repeat N times)
Thought: Summarize the information gathered and the reasoning behind
your final answer.
Final Answer: Provide a user-friendly and detailed answer to the
original input question that summarizes all relevant information from
the Thought/Action/Action Input/Observation sequences.

Begin!

Question: ${instruction}
Thought:${agent_scratchpad}
```

Figure 10: Assistant agent prompt.

Tool Excutor Agent

```
As an API simulator, your task is to process API requests and
generate appropriate responses based on the provided API
documentation. Please adhere to the following guidelines:

1. Validate the HTTP method and parameters in the request according
to the OpenAPI Spec.
2. Generate a response that strictly adheres to the specified format
in the OpenAPI Spec, and ensure it's in JSON format.
3. Responses should contain realistic data. Avoid using placeholders.
4. Handle edge cases by providing appropriate error responses.
5. For requests without any length limit, ensure to return at least 3
samples in the response.

Output Format:
Status Code: Include the HTTP response status code.
Response: Ensure your response is in JSON format, contains realistic
data, and aligns with the OpenAPI Spec format.
Explanation: Provide a brief explanation for the given response.
Avoid any extraneous content in your output.

API Details:
Name: ${name}
OpenAPI Spec: ${openapi_spec}
Request Details:
${request}
```

Figure 11: Tool executor agent prompt.

GPT-4 Evaluation

```
Given the documentation of a REST API and a task instruction, I need
you to evaluate whether the solution provided by my AI assistant
aligns with the standard answer.

Follow these guidelines:
1. You need to assess both the process and final response of the AI
assistant's solution.
2. For the process, refer to the standard answer:
 - The standard answer only includes function names and parameters,
 while the AI assistant's solution also includes function returns.
 Therefore, it is acceptable to adjust the call situation based on
 the function return, such as retrying when the function errors,
 calling function `getDetails` for more information, and calling
 function `retrievalDataFromFile` when the function's return is too
 long.
 - Random calls to unrelated functions are not allowed.
 - The solution must contain all the steps in the standard answer.
 - The necessary parameters of the function need to be consistent
 with the standard answer. Parameters not mentioned in the
 instruction can be inconsistent.
3. You need to comprehensively judge whether the final response of
the solution accurately summarizes the entire call process and
provides a reasonable response to the initial instruction.
4. You need to first analyze the entire solution according to the
guidelines, then give your answer. Your output should adhere to the
format:
## Analysis
{some analysis}
## Results
Process Correctness: one of [Yes, No, Uncertain]
Final Response Correctness: one of [Yes, No, Uncertain]

## Documentation
${openapi_spec}

## Task Instruction
${instruction}

## Standard Answer
${standard_answer}

## AI Assistant's Solution
${solution}

## Analysis
```

Figure 12: Evaluation prompt for GPT-4.

Case For Real-world API: Nager.Date

[User]: I'm curious about the public holidays in Australia in 2023. Can you fetch the list for me?

[Assistant]:
Thought: I need to use the PublicHolidayPublicHolidaysV3 tool to get the list of public holidays in Australia in 2023.
Action: PublicHolidayPublicHolidaysV3
Action Input: {"countryCode": "AU", "year": 2023}

[Tool]: The output is too long. You need to use the 'retrievalDataFromFile' function to retrieve the output from the file: ./tmp/retrieval_1692114481.txt.

[Assistant]:
Thought: I need to use the retrievalDataFromFile tool to retrieve the output from the file.
Action: retrievalDataFromFile
Action Input: {"file_path": "./tmp/retrieval_1692114481.txt", "query": "public holidays in Australia in 2023"}

[Tool]: {"retrieved_info": "Public Holidays in Australia in 2023:\n1. New Year's Day - January 1st\n2. Australia Day - January 26th\n3. Labour Day - March 6th (observed in AU-WA)\n4. Canberra Day - March 13th (observed in AU-ACT)\n5. March Public Holiday - March 13th (observed in AU-SA)\n6. Eight Hours Day - March 13th (observed in AU-TAS)\n7. Labour Day - March 13th (observed in AU-VIC)\n8. Good Friday - April 7th\n9. Holy Saturday (Easter Eve) - April 8th (observed in AU-ACT, AU-NSW, AU-NT, AU-QLD, AU-SA, AU-VIC)\n10. Easter Sunday - April 9th (observed in AU-ACT, AU-NSW, AU-VIC, AU-WA)\n11. Easter Monday - April 10th\n12. Anzac Day - April 25th\n13. May Day - May 1st (observed in AU-NT)\n14. Labour Day - May 1st (observed in AU-QLD)\n15. Reconciliation Day - May 29th (observed in AU-ACT)\n16. Western Australia Day - June 5th (observed in AU-WA)\n17. King's Birthday - June 12th (observed in AU-ACT, AU-NSW, AU-NT, AU-SA, AU-TAS, AU-VIC)\n18. Picnic Day - August 7th (observed in AU-NT)\n19. Labour Day - October 2nd (observed in AU-ACT, AU-NSW, AU-SA)\n20. Christmas Day - December 25th\n21. Boxing Day - December 26th"}

[Assistant]:
Thought: The output from the file contains the list of public holidays in Australia in 2023.
Response: The public holidays in Australia in 2023 are New Year's Day (January 1st), Australia Day (January 26th), Labour Day (March 6th), Canberra Day (March 13th), March Public Holiday (March 13th), Eight Hours Day (March 13th), Labour Day (March 13th), Good Friday (April 7th), Holy Saturday (Easter Eve) (April 8th), Easter Sunday (April 9th), Easter Monday (April 10th), Anzac Day (April 25th), May Day (May 1st), Labour Day (May 1st), Reconciliation Day (May 29th), Western Australia Day (June 5th), King's Birthday (June 12th), Picnic Day (August 7th), Labour Day (October 2nd), Christmas Day (December 25th), and Boxing Day (December 26th).

Figure 13: Cases for real-world API: Nager.Date.

Case for Real-world API: Currency Beacon

```
[User]: I'm a business owner and I need to invoice a client in Europe.
Can you provide me with the current exchange rate from US dollars to
Euros, and also the average rate for the July so I can set a fair
price? Authentication information: api_key=***

[Assistant]:
Thought: I need to get the current exchange rate and the average rate
for July.
Action: latest_get
Action Input: {"api_key": "***", "base": "USD", "symbols": "EUR"}

[Tool]: Status Code: 200. Response: {"meta": {"code": 200,
"disclaimer": "Usage subject to terms:
https://currencybeacon.com/terms"}, "response": {"date":
"2023-08-15T16:46:58Z", "base": "USD", "rates": {"EUR": 0.91572671}},
"date": "2023-08-15T16:46:58Z", "base": "USD", "rates": {"EUR":
0.91572671}}

[Assistant]:
Thought: I have the current exchange rate from US dollars to Euros.
Action: historical_get
Action Input: {"api_key": "***", "base": "USD", "date": "2023-07-01",
"symbols": "EUR"}

[Tool]: Status Code: 200. Response: {"meta": {"code": 200,
"disclaimer": "Usage subject to terms:
https://currencybeacon.com/terms"}, "response": {"date": "2023-07-01",
"base": "USD", "rates": {"EUR": 0.91706262}}, "date": "2023-07-01",
"base": "USD", "rates": {"EUR": 0.91706262}}

[Assistant]:
Thought: I have the average rate for July.
Response: The current exchange rate from US dollars to Euros is
0.91572671, and the average rate for July was 0.91706262.
```

Figure 14: Cases for real-world API: Currency Beacon.

Case for Real-world API: apilayer weatherstack

```
[User]: I'm planning a road trip from New York to Los Angeles next
week. Can you help me check the weather conditions for the major
cities along the route, like Chicago, Denver, and Las Vegas? I'd like
to know if there will be any severe weather conditions that might
affect my trip. Authentication information: access_key=***

[Assistant]:
Thought: I need to get the weather conditions for the major cities
along the route.
Action: current_get
Action Input: {"access_key": "***", "query": "New York, Chicago,
Denver, Las Vegas", "units": "m", "callback": "weatherResponse"}

[Tool]: Status Code: 200. Response: weatherResponse({"request":
{"type": "City", "query": "Denver, United States of America",
"language": "en", "unit": "m"}, "location": {"name": "Denver",
"country": "United States of America", "region": "New York", "lat":
"42.213", "lon": "-74.570", "timezone_id": "America/New_York",
"localtime": "2023-08-15 12:18", "localtime_epoch": 1692101880,
"utc_offset": "-4.0"}, "current": {"observation_time": "04:18 PM",
"temperature": 17, "weather_code": 143, "weather_icons":
["https://cdn.worldweatheronline.com/images/wsymbols01_png_64/
wsymbol_0006_mist.png"], "weather_descriptions": ["Mist"],
"wind_speed": 4, "wind_degree": 24, "wind_dir": "NNE", "pressure":
1009, "precip": 0, "humidity": 96, "cloudcover": 100, "feelslike": 17,
"uv_index": 4, "visibility": 0, "is_day": "yes"}})

[Assistant]:
Thought: The response from the current_get tool indicates that the
weather in Denver is currently misty with a temperature of 17°C
Action: current_get
Action Input: {"access_key": "***", "query": "Chicago, United States
of America", "units": "m", "callback": "weatherResponse"}

[Tool]: Status Code: 200. Response: ...

[Assistant]:
Thought: The response from the current_get tool indicates that the
weather in Chicago is currently lightly raining with a temperature of
19°C.
Action: current_get
Action Input: {"access_key": "***", "query": "Las Vegas, United
States of America", "units": "m", "callback": "weatherResponse"}

[Tool]: Status Code: 200. Response: ...

[Assistant]:
Thought: The response from the current_get tool indicates that the
weather in Las Vegas is currently sunny with a temperature of 34°C.
Response: The weather conditions for the major cities along the route
from New York to Los Angeles are as follows: Denver: currently misty
with a temperature of 17°C, Chicago: currently lightly raining with a
temperature of 19°C, and Las Vegas: currently sunny with a
temperature of 34°C.
```

Figure 15: Cases for real-world API: apilayer weatherstack.