Seal-Tools: Self-Instruct Tool Learning Dataset for Agent Tuning and Detailed Benchmark

Anonymous ACL submission

Abstract

This paper presents a new tool learning dataset Seal-Tools, which contains self-instruct API-like tools. Seal-Tools not only offers a large number of tools, but also includes instances which demonstrate the practical application of tools. Seeking to generate data on a large scale while ensuring reliability, we propose a self-instruct method to generate tools and instances, allowing precise control over the process. Moreover, our Seal-Tools contains hard instances that call multiple tools to complete the job, among which some are nested tool callings. For precise and comprehensive evaluation, we use strict format control and design three metrics from different dimensions. Therefore, Seal-Tools can serve as a new benchmark to evaluate the tool-calling ability of LLMs. Finally, we evaluate several prevalent LLMs and our finetuned model on Seal-Tools. The results show that current systems are far from perfect. The code, data and experiment results will be available at https://github.com/ .

1 Introduction

Large Language Models (LLMs) have shown strong abilities in many tasks in recent years (Achiam et al., 2023, Wu et al., 2023). Many researchers attempt to use the LLMs as agents (Shen et al., 2023, Patil et al., 2023, Liang et al., 2023), which help users complete difficult tasks by using external tools or plugins. The agents serve as a bridge between the users and the tools. Therefore, it is particularly important to teach the LLMs how to understand and utilize tools correctly (Qin et al., 2023, Li et al., 2023). For this purpose, we need to prepare high-quality tool learning datasets for enhancing the capabilities of the LLMs as well as precise evaluation.

A tool learning dataset often includes **tool pool** which contains different kinds of tools, and **instances** which call the tools to complete tasks. In the previous studies, the researchers have built several datasets and achieved a certain of success (Hao et al., 2023, Li et al., 2023, Xu et al., 2023), but the datasets exhibit some shortcomings. Hao et al. (2023) and Li et al. (2023) craft tools by hand but the amount is limited. Xu et al. (2023) collects real-world APIs from Rapid API Hub[1] to construct instances but results are evaled by ChatGPT with simple metrics that causes inaccurate evaluation and costs money. The benchmark of Li et al. (2023) is coarse-grained, only considering text similarity during evaluation. And its training data is not open source. To summarize, a large-scale high-quality dataset is urgently needed for tuning the agents and performing the automatic precise evaluation.

In our preliminary experiments, we directly use the LLMs to generate the tools and the instances. It is easy to obtain a large-scale dataset. However, the model has limited context length and often outputs duplicate tools. The generated instances are mostly simple which can be solved with a quick glance. Moreover, it's hard to ensure the correctness of tool callings in each instance due to the LLM hallucination.

To overcome the above challenges, in this paper we propose a self-instruct method to utilize the LLMs to generate a new tool learning dataset, named as **Seal-Tools**. In our method, we first use a LLM to generate a set of fields which refer to different domains, and then tools (just like in Figure 1) are generated for each field. This simple strategy can well avoid the duplication and diversity problems. Given the tool pool, we further generate the instances which call single/multiple tools to resolve requests. We separate the generation into multi steps and set up several checking steps to greatly reduce errors caused by the LLM hallucination. We use JSON format to describe the tools and instances strictly. Besides, we suc-

[1] https://rapidapi.com/hub

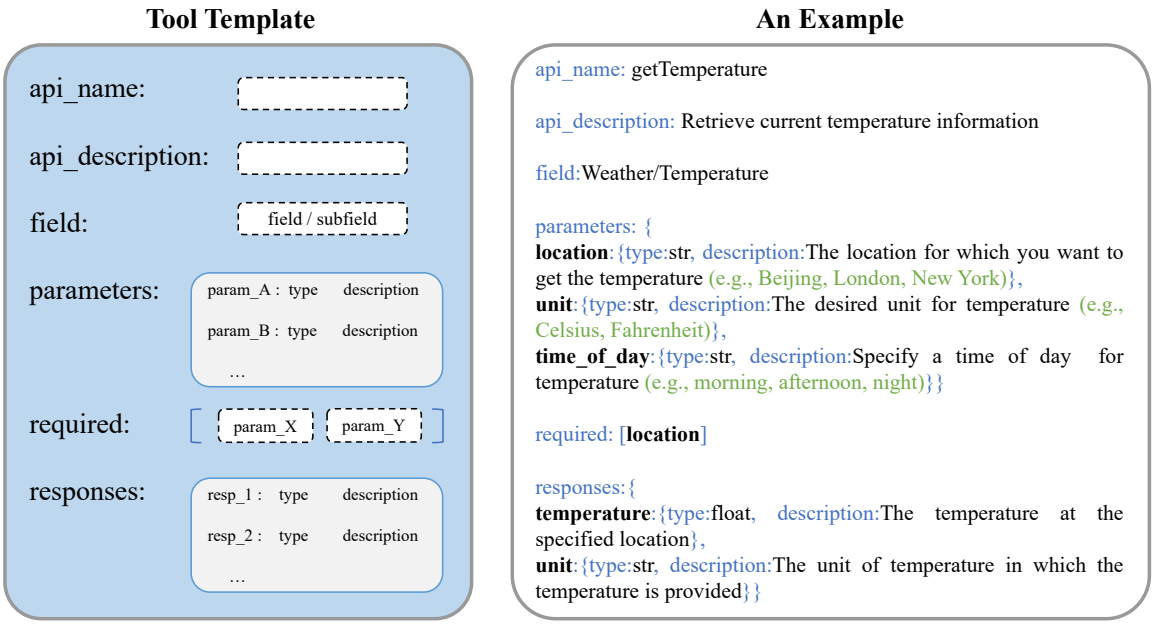

Figure 1: Tool template of Seal-Tools and the tool "getTemperature" as an example.

cessfully generate some instances with nested tool callings thanks to the well-designed construction method and calling template. These instances have extremely difficult queries to solve and are valuable for finetuning.

To make Seal-Tools a comprehensive benchmark, we design three evaluation dimensions for detailed metrics: Output Format, Tool Selection, and Tool-Parameter Filling-in. Since we post-process the LLM outputs into JSON format in Seal-Tools, the evaluation can be more automatic and precise compared to the previous ones.

Contributions of this paper are listed as following:

- We propose a self-instruct method to use LLMs to generate tool learning datasets. Our method can generate various in-field tools and single/multiple-tool instances which are mostly reliable through quality control.
- A brand new tool learning dataset, named as **Seal-Tools**, is constructed for agent tuning. Compared with the previous datasets, Seal-Tools is relatively large and contains hard instances with nested tool callings. With the help of strict format control, we can perform precise evaluation automatically.
- For a comprehensive evaluation, we design 3 main metrics in different dimensions. We implement the mainstream agent system and finetune its foundation model with Seal-Tools. From the results, we find that the current systems show room for improvement, especially in nested calling.

2 Related Work

2.1 In-Context Learning

In-context learning (ICL) has become the paradigm for the use of LLMs. We add some examples or so-called demonstrations of the task in prompt. Models can finish what users want them to do very well through learning from demos. According to Dong et al. (2022), ICL presents for the first time in GPT3's technical report (Brown et al., 2020). As it's widely used, Hendel et al. (2023) explores the operational mechanisms behind it. ICL demos may play similar roles as continuous-value learnable token of prompt tuning. Gao et al. (2023) proves the importance of demo selection and studied how to select better demos with the help of retriever. Auto-ICL (Yang et al., 2023) attempts to let LLMs generate similar questions with answers. ICL then can be applied in scenarios without human supervision.

2.2 Tool Learning

Tool learning is actually a subtask of serving LLMs as agents in our opinion. The agent which supports for tool calling consists of the foundation model, retriever and tool pool. The foundation model is

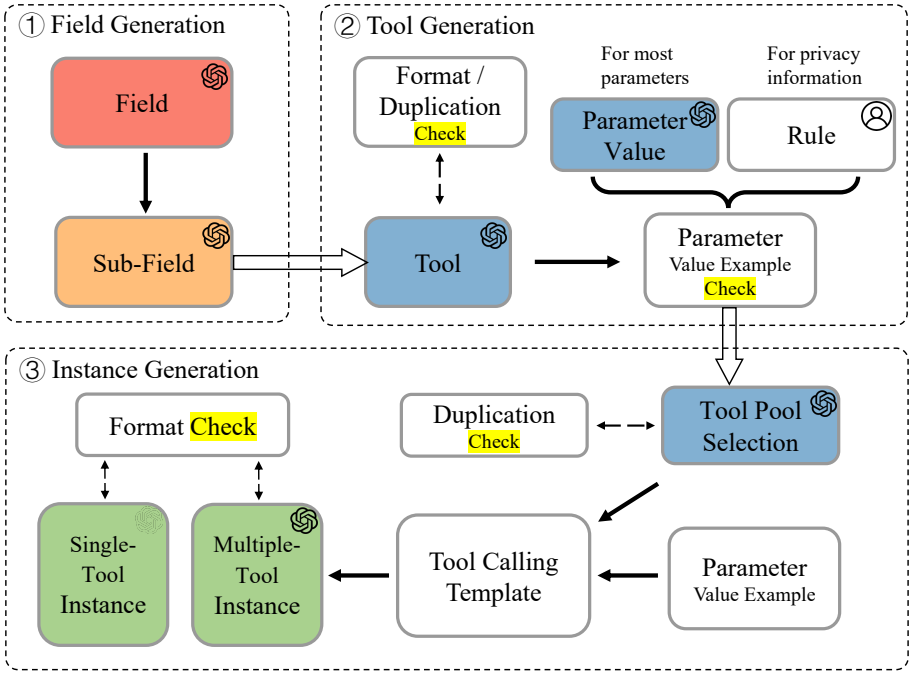

Figure 2: Flowchart of the dataset construction method.

the core component which decides how to reply to users and whether to call tools. The retriever is responsible for researching relevant tool information according to the user query. The tool pool is used to store and manage tool information. We look through a lot of related works and find that tool learning can be summarized into two categories (prompt-based and finetune-based) according to how LLMs learn to use tools.

a. Prompt-Based Tool Learning

Prompt-based agent has an external tool pool. The foundation model selects proper tools in prompt given by the retriever like ToolLLM (Qin et al., 2023) and API-Bank (Li et al., 2023). HuggingGPT (Shen et al., 2023) and Gorilla (Patil et al., 2023) serve models as tools for use. Prompt-based agent has more flexible and large-scale tool pools. We use this kind of framework in this paper to explore how LLMs can call a huge amount of tools accurately.

b. Finetune-Based Tool Learning

Finetune-based agent doesn't need an extra tool pool. The foundation model learns which tools it has and how to use them through finetuning. Gorilla (Patil et al., 2023) tests the zero-shot scenario (which means no retriever) since it's finetuned with relevant self-instruct dataset. ToolkenGPT (Hao et al., 2023) transforms tools into special tokens and adds them into the vocabulary. Finetune-based agent calls tools accurately and rapidly but needs to be finetuned in advance to remember those tools.

3 Method

In this section, we talk about how our dataset construction method works and what the quality of constructed result is. The highlight of the method is that it is convenient for everyone to try and does not require too much human involvement. We can put more efforts on designing the tool template and ICL prompts, LLMs will finish all other jobs. The amount of data generated can be controlled at will within LLM capability.

3.1 Dataset Construction

We adopt the self-instruct strategy to generate datasets with LLMs. But generative models have shortcomings like hallucination, context length limitations, etc. When constructing dataset with LLMs, we must make sure there are no logical errors and the answer matches the question because of the hallucination. Also avoiding too many duplicates when generating data on a large scale is a challenge due to context length limitations. Thus, to overcome the above challenges, we propose a novel dataset construction method. As shown in Figure 2, our solution contains three steps: Field Generation, Tool Generation and Instance Generation. In our paper, Field, Tool, and Instance are defined as follows:

Field: It describes the specific domain to which

the tool belongs. We categorize tools into specific layered fields (2 layers here, field and subfield) based on their use.

Tool: A tool is able to perform a specific job. It has name, description, input/output parameters and so on as shown in Figure 1 before.

Instance: It is a practical use case of the tool, containing user query and tool calling. There are two categories, **single-tool instance** and **multiple-tool instance**. Single-tool instance invokes only one tool while multiple-tool instance invokes multiple tools.

In brief, LLM crafts in-field tools according to pre-generated fields. Then LLM makes up some instances which can be solved by these tools.

We construct dataset Seal-Tools by ChatGPT. Due to funding constraints, the dataset scale has not yet reached the upper limit of our construction method and model capacity.

3.1.1 Field Generation

This section is about how to generate various fields. We've tried skipping this step and going straight to tool generation. We find the tools generated repeat frequently in that way. To enhance tool diversity, we use field information as an anchor. Large models are asked to generate tools corresponding to a segmented field. We set hierarchical fields to ensure that the functional classification of tools is sufficiently granular. The number of tools generated has steadily increased.

First take an field example as the initial demonstration to fill in the prompt. LLM generates field set with the insturction "Please generate a field list in the format of a python list." through ICL. After that, subfields for all fields in field set are generated with the instruction "Please generate a subfield list in the format of a python list for the __?___ field." in the same way. Finally we generate 2 levels of fields, including 146 fields with 5,860 subfields.. As shown in Figure 3, field "Science" has multiple subfields.

3.1.2 Tool Generation

Initially, we think this step would be simple and we just have to generate a lot of tools. In the next step, We try to generate instance with only tool information but it works badly. LLM tends to fill in api callings with general and repetitive concepts which are in low quality and these values may be not mentioned in the query. We later find that it would be effective to generate examples of parameter values

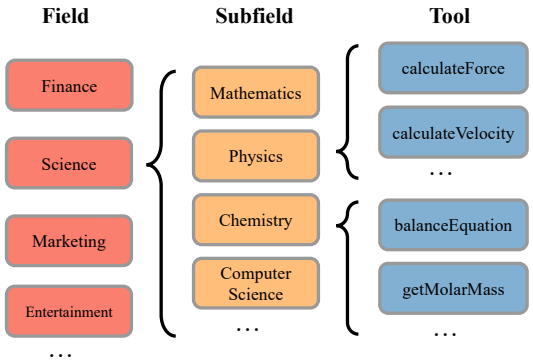

Figure 3: Some examples of generated fields/subfields and tools.

at the same time as generating the tool.

When generating parameter values, We find LLM tends to refuse to output entities in real world. That's probably a restriction added for security and privacy. So we ask LLM to "make up" some proper entities as parameter values. The experiment shows that this kind of prompt works well. But there may be potential privacy leaks from LLMs.

Then we figure out a suitable method for tool generation. At first, we design the tool template and make up a tool example to initialize the tool pool. LLM generates tools in the given subfield with the instruction "Please generate some APIs according to the given field/sub-field. An API is a function with input parameters and output responses." through ICL. When writing the prompt, it's important to add requirements of generating examples of values in parameter description like "(e.g. , ___, ___)". After checking the format of generated new tools, we put them into the tool pool if not repetitive. When LLM doesn't output new tools in one subfield for several times, we switch to next subfield. After generating with all subfields, we examined whether required parameters of each tool have example values. If not, we collect them in categories and generate examples in batch through ICL. For some parameters of sensitive user information (phone number, email address, etc.) , we generate values using rules. In conclusion, we get the tool set and parameter values now. Finally we generate 4,076 tools. Every tool belongs to a subfield just like Figure 3.

3.1.3 Instance Generation

The instance contains two parts: the user query and tool callings (how to invoke tools like Figure 4) . LLMs can make up different styles of queries

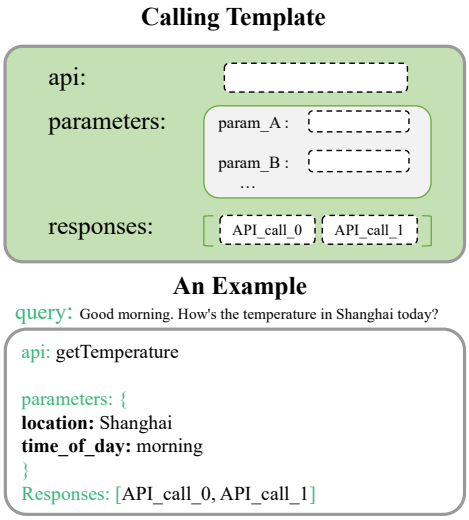

Figure 4: An instance template for single-tool calling.

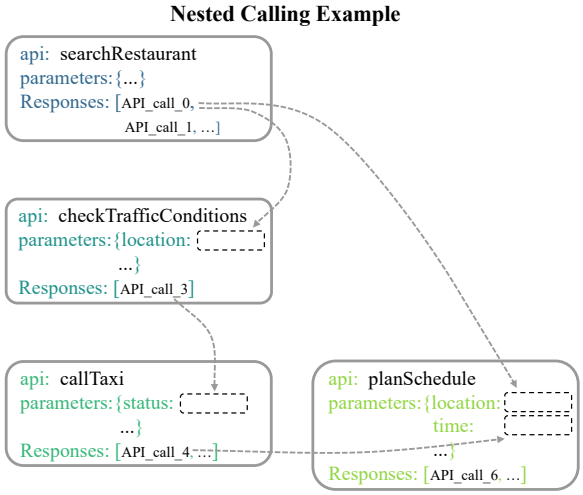

Figure 5: An example for multiple-tool instance with nested calling.

with given parameter values. For better finetuning effects, we choose to generate queries in a brief style.

For single-tool instances, we choose the tool and pick up parameter values to fill in the prompt. It is successful to generate the single-tool invoking instance by ICL. Finally we generate 4076 single-tool instances.

For multiple-tool instances, the prompt in single-tool instance generation is hard to get good output. The tool callings generated through ICL are mostly confusing in format. Since it might be difficult for LLMs nowadays, we try to simplify it into two steps and transform the question answering task into blank filling task. At the first step, LLM should choose several tools which can be combined into one query from huge amount candidate tools in prompt as shown in Figure 2. Then we generate the tool invoking template according to the chosen tools. At the second step, LLM only needs to fill in parameter values with given examples and generate the relevant query. Finally we generate 10k multiple-tool instances.

The innovation of this step is that we could generate some multiple-tool instances with nested calling (simply called **nested instances**) in this way. In the nested instance, Figure 5 for example, the response value of the previous invoked tool could be parameter value of the next invoked tool. All tools invoked can form a directed acyclic graph instead of flow line. This is closer to complex real-world application scenarios and makes our evaluations more effective.

3.2 Dataset Analysis

We compare Seal-Tools with several common tool learning datasets in Table 1: ToolBench[2], API-Bank[3] and ToolAlpaca[4]. The result shows that our dataset is competitive among all datasets.

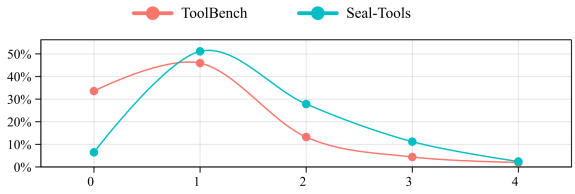

Figure 6: Required parameters amount of tools comparison between **Seal-Tools** and **ToolBench**.

With regard to tools, we propose the first open-source batch generation method with self-instruct strategy. The quality of tools generated may be not inferior to real-world tools collected by ToolBench. As Figure 6 shows, we find nearly 34% tools in ToolBench have no required parameters. In Seal-Tools, the amount is only 6%. Models have to fill in more parameters in each instances just like the avg. params amount show in Table 1.

As for instances, the scale of Seal-Tools is very large, second only to ToolBench. It also contains

[2]**ToolBench** lists data in Table 1 of the original paper. We count number of APIs as tools in our settings.

[3]**API-Bank** has evaluation data implemented by human and self-instruct training data. We get information of APIs and single/multiple callings in Table 2 of the original paper. The training data seems to be not publicly available.

[4]**ToolAlpaca** lists data in Table 1 of the original paper. We count number of functions as tools.

		Seal-Tools (ours)	ToolBench (Xu et al., 2023)	API-Bank (Li et al., 2023)	ToolAlpaca (Tang et al., 2023)
Tools	**Source**	self-instruct	real-world	self-instruct + annotation	real-world + annotation
	Amount	4,076	16,464	2,211	2,386
	Avg. params (required)	1.551	1.013	unknown	N/A†
Instances	**Source**	self-instruct	self-instruct	self-instruct + annotation	self-instruct
	Amount	14,076	126,486	4,125	3,938
	├Multiple-tool callings	10,000	≈85,330	615	1,426
	└Nested-tool callings	586	N/A♣	unknown	N/A♣
	Cross-field tool callings	✔	✔	unknown	✘
Benchmark	**Metric**	Acc, P/R/F1	Pass Rate, Win Rate	Acc, Rouge	N/A
	Tool parameter	✔	✘	✔	N/A
	Deterministic evaluation	✔	✘	✔	N/A
Total open-source?		✔	✔	✘	✔
Extensible (fully self-instruct) ?		✔	✘	✘	✘

Table 1: Comparison of several Tool Learning datasets. † Formatting confusion. ♣ Multi-step.

more hard instances like cross-field callings and nested callings which test LLM's ability to think logically. Most datasets ask LLMs to call one tool in one response. It often needs multiple steps to handle difficult problems. While Seal-Tools asks LLMs to give above multiple-tool callings in one turn chat instead of step-by-step by other datasets. It is much more difficult to solve and closer to real-world scenarios. We think it helps improve agent execution efficiency.

Seal-Tools also provides very detailed evaluations. We post-process the output of LLMs into JSON format for evaluating tool selection and parameter filling-in. But ToolBench and API-Bank do not process it. ToolBench could only get pass rate and win rate given by ChatGPT. It is not deterministic since ChatGPT has its own preferences and limitations. API-Bank calculates the rouge score. ToolAlpaca doesn't provide evaluation. When evaluating agents, we could calculate the correctness of every selected tool and its parameters. Model capabilities such as understanding tools, generating parameters, etc. are presented more clearly.

Moreover, it can be extended with our current most automated construction method. Seal-Tools can be used in more places without worrying about dataset scale. We just need to provide the universal tool template and ICL prompts to generate much more data. Other method can only generate more instances. Both tools and instances can be generated much more within the capability of LLMs. Furthermore, tools and instances are generated together in a complete self-consistent generative process in our method so they can be more harmonized with each other.

4 Experiment

4.1 Evaluation Metric

There are currently no standardized evaluation metrics for tool learning task. For evaluating in detail, we design 3 main eval metrics: Format ACC, Tool P/R/F1 and Parameter P/R/F1. Specific calculations can be found in Appendix C.

Format ACC measures the format accuracy of model output. The foundation model should follow instructions in the prompt so that tools can be invoked well.

Tool P/R/F1 measures the tool selection ability of foundation model. The specific calculation is similar to P/R/F1 metrics for the information extraction task. Please refer to the appendix for details, the same below.

Parameter P/R/F1 measures the tool parameter filling-in ability. We use these metrics to hierarchically examine the capability of the foundation model to understand and use tools.

4.2 Main Result

4.2.1 Overall

Here is the main evaluation section of the paper. We hope that this experiment simulates the performance of agent in a real environment. As mentioned in Section 2.2, when a prompt-based agent system receives a user query, it uses the retriever to search for relevant tools firstly. Then the foundation model decides whether to use candidate tools according to the query and generate a reply. So we eval the performance of different retrievers and select the best retriever DPR to add to the system. More details about the selection of retriever are in Appendix D.

Model	Format ACC	Tool			Parameter		
		P	R	F1	P	R	F1
ChatGPT *(gpt-3.5-turbo-0613)*	96.16	83.20	74.73	78.74	68.63	66.85	67.73
GPT4 *(gpt-4-0613)*	97.12	90.02	74.71	81.65	80.52	67.57	73.48
LLaMA2 *7B*	40.55	47.91	26.74	34.33	33.52	20.43	25.39
LLaMA2-Chat *7B*	78.73	62.10	53.91	57.72	44.92	43.24	44.06
Vicuna *7B-v1.5*	70.83	67.33	49.81	57.26	49.11	42.26	45.43
Mistral *7B-Instruct-v0.2*	77.03	76.84	59.65	67.16	64.81	50.25	56.61
ToolLLaMA2 *7B-v2*	13.44	19.35	0.96	1.84	18.98	0.84	1.61
Ours *(finetuned on LLaMA2-7B)*							
w/ BM25	95.57	79.67	74.79	77.15	73.51	**70.76**	72.11
w/ DPR	**95.86**	**82.81**	**77.84**	**80.25**	**75.95**	70.23	**72.98**

Table 2: Overall result. All LLMs use DPR retriever as default.

We finetune LLaMA2-7B[5] with Seal-Tools. The tools in prompt are given mainly by retriever DPR. We add the missing gold tools to the prompt in the train split. When evaluating, DPR gives the top-5 relevant tools.

Table 2 represents the main result of tool learning task. Open-source LLMs perform similarly. They all have great potential for making further progress. Searchers can try to add more tool learning datasets in the pre-training phase.

It is also reasonable that ToolLLaMA which is finetuned with ToolBench has a bad performance. Single dataset fine-tuning severely impacts LLM's understanding of other instructions.

After finetuning with Seal-Tools, our model can output much more correct tool callings than the base model. Tool F1 increases 45.92% and Parameter F1 increases 47.59% as shown in Table 2. It even outperforms ChatGPT and is slightly inferior to GPT4. The score proves that Seal-Tools is efficient in finetuning.

We discuss about various instances in Section 4.2.2 and Section 4.2.3 below. The difficulty of different instances is shown in Figure 7 roughly for reference (Higher in the bar graph means simpler).

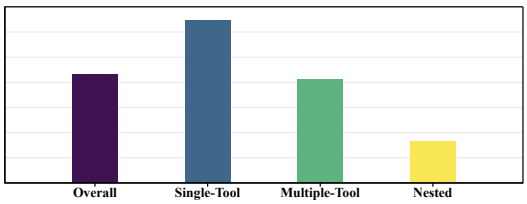

Figure 7: Simplicity level of different kinds of instances. Visualized using Mistral's Parameter F1 metric.

[5] https://huggingface.co/meta-llama/Llama-2-7b-hf

4.2.2 Single/Multiple-Tool Instances

We count for single/multiple-tool instances in Table 3 for more details. For most models, they do better in single-tool instances than multi-tool instances. Calling a single tool is easier than calling multiple tools, which is to be expected. While LLaMA2 and Vicuna are exceptions. Since this looks a little strange, we check their outputs and find that they tend to call more tools when dealing with single-tool instances. Maybe LLaMA2 uses some low-quality corpus during pre-training or it is not well trained. LLaMA2 tends to invoke more tools and doesn't know how to make trade-offs.

Our model outperforms GPT4 in single-tool instances but falls slightly behind in multiple-tool instances. How to further improve the performance of multiple-tool calling is the focus of our future research.

4.2.3 Nested Instances

We collect all 586 nested instances in Seal-Tools. The performance of different models is listed in Table 4. It shows that nested instances are the most difficult for models to solve. LLaMA2-Chat and Mistral behaves poorly compared to other data.

For our model, since 501 of these data have been used in the train split, We calculate the final scores separately. Although the model has seen them during the finetuning process, its Parameter F1 is only 77.12%, not too high. For unseen data, it performs 5.11% worse than multiple-tool instances. But it is still better than raw model LLaMA2 which confirms the effect for finetuning. In summary, our dataset Seal-Tools is of high quality with these hard instances.

Model	Single-Tool							Multi-Tool						
	Format ACC	Tool			Parameter			Format Acc	Tool			Parameter		
		P	R	F1	P	R	F1		P	R	F1	P	R	F1
ChatGPT	98.98	88.01	94.90	91.33	74.28	83.94	78.82	95.38	82.70	73.01	77.55	68.11	65.49	66.77
GPT4	98.64	88.16	96.26	92.03	82.00	85.16	83.55	96.70	90.24	72.86	80.62	80.37	66.17	72.58
LLaMA2	44.22	25.83	42.18	32.04	15.93	28.66	20.48	39.53	54.52	25.42	34.68	38.43	19.78	26.11
LLaMA2-Chat	85.37	40.27	81.63	53.93	26.54	63.21	37.38	76.89	67.02	51.54	58.27	49.03	41.64	45.03
Vicuna	76.53	47.65	72.45	57.49	33.79	59.76	43.17	69.25	71.13	47.88	57.23	51.85	40.87	45.71
Mistral	86.73	72.99	86.39	79.13	66.14	68.29	67.20	74.34	77.36	57.36	65.88	64.67	48.81	55.63
ToolLLaMA	21.77	12.50	2.72	4.47	11.94	1.63	2.86	11.13	22.95	0.81	1.57	21.05	0.78	1.50
Ours														
w/ BM25	**98.30**	91.81	91.50	91.65	84.31	85.16	84.73	94.81	78.57	73.36	75.87	72.61	**69.61**	71.08
w/ DPR	**98.30**	**93.13**	**92.18**	**92.65**	**85.54**	**85.37**	**85.45**	**95.19**	**81.88**	**76.61**	**79.16**	**75.12**	69.02	**71.94**

Table 3: Results for single-tool / multiple-tool instances.

Model	Format ACC	Tool			Parameter		
		P	R	F1	P	R	F1
LLaMA2-Chat	79.86	73.04	58.39	64.90	37.23	34.66	35.90
Mistral	68.43	84.16	57.67	68.44	52.00	36.94	43.20
Ours	96.76	89.64	85.82	87.69	77.32	74.15	75.70
├ has seen (501)	96.41	91.03	86.61	88.76	78.88	75.43	77.12
└ still unseen (85)	98.82	81.71	81.08	81.40	67.66	66.02	66.83

Table 4: Result for nested instances.

4.3 Extended Result

4.3.1 Evaluation of Parameter Filling-In

We focus on testing LLMs' parameter filling-in ability for multiple-tool instances in this subsection. Prompts for finetuning LLaMA2 and evalutation only contain gold tools. The results are listed in Table 5.

For tool selection, since only gold tools are given in prompt, Tool P for each model should ideally reach 100%. Our model works as expected, but LLaMA2 and ChatGPT do not due to the in-context hallucination. For parameter filling in, as shown in Table 5, the finetuned model performs very well even through only the results of multiple-tool instances are counted. Considering all the above facts,, Seal-tools is helpful for improving the ability of parameter filling-in for the LLM. However, how to finetune the LLM to make it more robust remains to be investigated.

Model	Format ACC	Tool			Parameter		
		P	R	F1	P	R	F1
LLaMA2-chat	82.74	99.86	80.72	89.27	72.37	70.38	71.36
ChatGPT	94.06	99.97	92.82	96.26	80.94	85.22	83.02
Ours	**98.87**	**100.00**	**98.84**	**99.41**	**94.26**	**93.65**	**93.95**

Table 5: Result of parameter filling-in. Only needed tools are given in the prompt.

4.3.2 Error Analysis

In Figure 8, we count the types of errors made by our model with retriever DPR in Section 4.2. Understanding where mistakes are made allows us to continually strive for further improvement.

For tool selection, how to retrieve all needed tools is most urgent. The limitations of existing retriever training methods are mentioned in Appendix D. Hallucination also needs to be noted, since models may generate tools that are not in the retrieval results.

For parameter filling in, most of the errors are that models do not extract the correct keywords from queries. LLM omits the required parameters for 7% and overfills with unmentioned parameters for 9%. Besides, 14% of the errors are due to models not understanding the query requirements or not converting to the parameter request format.

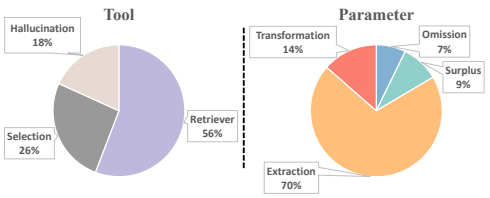

Figure 8: Error in tool selection and parameter filling-in.

5 Conclusion

In this paper, we present a novel construction method for building dataset Seal-Tools which includes a set of tools and instances. In our method, we carefully control the quality of auto-generated data, increasing reliability and diversity. In Seal-Tools, hard instances include nested tool callings and cross-field callings, which have seldomly been investigated in previous studies.. We further design evaluation metrics from three dimensions. Experimental results show that current agent systems still have room for improvement. We believe that Seal-Tools can serve as a new benchmark and boost the research on tool learning with LLMs.

Limitations

One possible limitation of our work is the format of tools. We refer to previous work and use JSON format to store and call tools. We cannot prove whether LLMs are good at handling data in JSON format. The final evaluation results may be affected to some extent. We need to find a more reasonable data format or try to avoid its impact on the tool learning task in the future.

Another limitation is that our data is generated by LLMs so that we cannot control the number of high quality hard instances. We think instances with nested tool callings is of extremely high quality. However, it accounts for a relatively small portion of our generated instances with multiple-tool callings.

Besides, we talk about the extensibility of SealTools with our self-instrcut dataset construction method. However, there is no discussion of the upper limit of its construction. As the LLM continues to output, the tools or instances generated will be more easily duplicated. Due to funding limitations, we are unable to do larger scale construction experiments on ChatGPT and GPT4. However, there is no downward trend in the unit response validity of ChatGPT in our trial which can be seen in Appendix B.

References

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

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Bill Byrne, Karthik Krishnamoorthi, Chinnadhurai Sankar, Arvind Neelakantan, Ben Goodrich, Daniel Duckworth, Semih Yavuz, Amit Dubey, Kyu-Young Kim, and Andy Cedilnik. 2019. Taskmaster-1: Toward a realistic and diverse dialog dataset. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4516–4525, Hong Kong, China. Association for Computational Linguistics.

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

Lingyu Gao, Aditi Chaudhary, Krishna Srinivasan, Kazuma Hashimoto, Karthik Raman, and Michael Bendersky. 2023. Ambiguity-aware in-context learning with large language models. *arXiv preprint arXiv:2309.07900*.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. 2023. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings. *arXiv e-prints*, pages arXiv–2305.

Roee Hendel, Mor Geva, and Amir Globerson. 2023. In-context learning creates task vectors. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 9318–9333, Singapore. Association for Computational Linguistics.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online. Association for Computational Linguistics.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. API-bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3102–3116, Singapore. Association for Computational Linguistics.

Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, et al. 2023. Taskmatrix. ai: Completing tasks by connecting foundation models with millions of apis. *arXiv e-prints*, pages arXiv–2303.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv e-prints*, pages arXiv–2305.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv e-prints*, pages arXiv–2307.

Abhinav Rastogi, Xiaoxue Zang, Srinivas Sunkara, Raghav Gupta, and Pranav Khaitan. 2020. Towards scalable multi-domain conversational agents: The schema-guided dialogue dataset. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 8689–8696.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. *arXiv e-prints*, pages arXiv–2303.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *arXiv e-prints*, pages arXiv–2306.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv e-prints*, pages arXiv–2307.

Tianyu Wu, Shizhu He, Jingping Liu, Siqi Sun, Kang Liu, Qing-Long Han, and Yang Tang. 2023. A brief overview of chatgpt: The history, status quo and potential future development. *IEEE/CAA Journal of Automatica Sinica*, 10(5):1122–1136.

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. *arXiv e-prints*, pages arXiv–2305.

Jinghan Yang, Shuming Ma, and Furu Wei. 2023. Auto-icl: In-context learning without human supervision. *arXiv preprint arXiv:2311.09263*.

Xiaoxue Zang, Abhinav Rastogi, Srinivas Sunkara, Raghav Gupta, Jianguo Zhang, and Jindong Chen. 2020. MultiWOZ 2.2 : A dialogue dataset with additional annotation corrections and state tracking baselines. In *Proceedings of the 2nd Workshop on Natural Language Processing for Conversational AI*, pages 109–117, Online. Association for Computational Linguistics.

A Prompts of Dataset Construction

A.1 Field Generation

Generating Initial Field

Please generate a field list in the format of a python list. Try to cover all areas.
Tips:
1. The field should be coarse-grained.
2. The job is really important. Please finish it perfectly with your full effort.
For example:
field_list = [
"{}",
"{}",
]

Generating Sub-Field

Please generate a subfield list in the format of a python list for the "{}" field.
Tips:
1. The subfield should be fine-grained.
2. The subfield list is used to classify tasks to the specified subfield.
3. The job is really important. Please finish it perfectly with your full effort.

A.2 Tool Generation

Generating In-Field Tool

Please generate some APIs according to the given field/sub-field. An API is a function with input parameters and output responses. It's like a tool to help with all kinds of fields. The generated APIs should be related to the field. This task is really important for human beings, so please finish it with your best effort.

For example:
field:"{}"
sub-field:"{}"
{}

Tips:
1. Generate enough parameters in "parameters" list. Parameters in the "required" list are definitely needed each time; only core parameters should be selected from "parameters" list.
2. The format of the "field" key is "field/sub-field".
3. Your answer should be in JSON format， the format of your answer should be strictly consistent with the example.
4. Make sure descriptions of parameters end with examples of values in the format of "(e.g., value1, value2, value3, ...)". You can just make up some.
5. The "type" key in lists of "parameters" and "responses" should be selected from ["str", "int", "float", "bool"].

Now generate some APIs like above as many as possible.

field:"{}"
sub-field:"{}"
{{ }}

A.3 Instance Generation

Generating Single-Tool Instance

Please generate the task description. The task requires calling API to finish. Make sure the task description coherent and natural. Please don't mention API in task description, API calling should be obtained by logical derivation.

For example:
function calling =
{{"api":"translate", "parameters":{{"text":"Hello world", "source_language":"English", "target_language":"Japanese"}}}}
Task description =
[Tell me how to speak "Hello world" in Japanese.]

function calling =
{{"api":"book_meeting", "parameters":{{"meeting_title":"academic research", "meeting_date":"2023-09-10", "meeting_time":"3:00 p.m."}}}}
Task description =
[Book a meeting for "academic research" on September 10, 2023, at 3:00 p.m.]

Now finish the following content in the format of the example above.

function calling =
{}
Task description =
[ ]

Tool combination

Here is a list of APIs. Please select and combine parts of given_apis to create a specific and complex task.
Tips:
1. Ensure that the selected APIs have a strong association and a logical relationship to each other.
2. You don't need to follow the original order of the APIs, but the chronological order of execution.

For example:
input:
given_apis = [{'getWeatherForecast': 'Retrieve weather forecast information'}}, {{'calculateBMI': 'Calculate Body Mass Index (BMI) based on height and weight'}}, {{'translateText': 'Translate text from one language to another'}}, {{'generateQRCode': 'Generate a QR code for a given text or URL'}}, {{'getHotelDetails': 'Retrieve detailed information about a hotel'}}, {{'getAirQualityIndex': 'Retrieve the air quality index (AQI) information for a specific location'}}, {{'searchRestaurant': 'Search for a restaurant based on various criteria'}}, {{'checkTrafficConditions': 'Retrieve current traffic conditions information'}}, {{'searchHotels': 'Search for hotels based on various criteria'}}, {{'reserveRentalCar': 'Reserve a rental car for a specific location and time'}}, {{'checkFlightAvailability': 'Check the availability of flights for a specified route and date'}}, {{'getArticleDetails': 'Retrieve details of an article by providing its identifier'}}, {{'cancelHotelReservation': 'Cancel a hotel reservation'}}, {{'callTaxi': 'Request a taxi service for transportation'}}]
output:
selected_apis = ['searchHotels', 'getHotelDetails', 'cancelHotelReservation']
task_description = ['Find the reserved hotel and obtain its information in order to cancel the reservation due to a schedule change.']

Please return the chosen list of APIs in the format of a Python list named 'selected_apis' and generate a paragraph describing the task, as shown in the upper example. Don't mention any API in the 'task_description'.

input:
given_apis = {}
output:

Generating Multiple-Tool Instance

Please use APIs in api_list to create a specific and complex task. First, fill in the blanks with parameter values in api_calling. Then, write the task description based on the api calling.
Tips for improved_api_calling generation:
1. Borrow from the parameter description or make up some specific and niche entities in reality as parameter values, without using the word "example".
2. Whenever possible, use the responses("API_call_" + serial number) of previous APIs as parameter values.
3. For different parameters, try to set the same value to combine APIs together and make the task more consistent.
Tips for task_description generation:

1. Make sure that all parameter values in the improved_api_calling list are mentioned in the task_description except the "API_call_" + serial number or "API".

For example:
input:
api_list = [{{"api_name": "searchRestaurant", "api_description": "Search for a restaurant based on various criteria", "parameters": {{"cuisine": {{"type": "str", "description": "The type of cuisine you prefer"}}, "price_range": {{"type": "str", "description": "The price range you're looking for"}}, "rating": {{"type": "float", "description": "The minimum rating you want for the restaurant"}}, "open_now": {{"type": "bool", "description": "Specify if you want to find restaurants that are currently open (true or false)"}}}}, "required": [], "responses": {{"location": {{"type": "str", "description": "The location of the enquired restaurant"}}}}}}, {{"api_name": "checkTrafficConditions", "api_description": "Retrieve current traffic conditions information", "parameters": {{"location": {{"type": "str", "description": "The location for which you want to check traffic conditions"}}, "time_of_day": {{"type": "str", "description": "Specify the time of day for checking traffic conditions"}}, "traffic_source": {{"type": "str", "description": "Specify the source of traffic information"}}, "include_incidents": {{"type": "bool", "description": "Include information about traffic incidents and accidents"}}}}, "required": ["location"], "responses": {{"traffic_level": {{"type": "str", "description": "The traffic level at the specified location"}}, "estimated_travel_time": {{"type": "int", "description": "The estimated travel time in minutes based on current traffic conditions"}}, "average_speed": {{"type": "int", "description": "The average speed of traffic in miles per hour (mph)"}}, "incidents": {{"type": "str", "description": "Information about any traffic incidents or accidents (if included in the request)"}}}}}}, {{"api_name": "callTaxi", "api_description": "Request a taxi service for transportation", "parameters": {{"pickup_location": {{"type": "str", "description": "The location where you want to be picked up"}}, "destination": {{"type": "str", "description": "The destination address where you want to go"}}, "passenger_count": {{"type": "int", "description": "The number of passengers"}}, "ride_type": {{"type": "str", "description": "The type of ride you prefer"}}, "special_requests": {{"type": "str", "description": "Any special requests or instructions for the driver"}}}}, "required": ["pickup_location", "destination"], "responses": {{"status": {{"type": "str", "description": "The status of the taxi request"}}, "driver_name": {{"type": "str", "description": "The name of the assigned taxi driver (if available)"}}, "estimated_arrival_time": {{"type": "str", "description": "The estimated time of arrival of the taxi"}}}}}}]
origin_api_calling = [{{"api": "searchRestaurant", "parameters": {{"cuisine": ___}}, "responses": ["API_call_0"]}}, {{"api": "checkTrafficConditions", "parameters": {{"location": ___, "time_of_day": ___}}, "responses": ["API_call_1", "API_call_2", "API_call_3", "API_call_4"]}}, {{"api": "callTaxi", "parameters": {{"pickup_location": ___, "destination": ___}}, "responses": ["API_call_5", "API_call_6", "API_call_7"]}}]
output:
improved_api_calling = [{{"api": "searchRestaurant", "parameters": {{"cuisine": "Italian"}}, "responses": ["API_call_0"]}}, {{"api": "checkTrafficConditions", "parameters": {{"location": "API_call_0", "time_of_day": "afternoon"}}, "responses": ["API_call_1", "API_call_2", "API_call_3", "API_call_4"]}}, {{"api": "callTaxi", "parameters": {{"pickup_location": "Nanjing Road", "destination": "API_call_0"}}, "responses": ["API_call_5", "API_call_6", "API_call_7"]}}]
task_description = ["Please help me to plan a convenient and enjoyable dinner outing. Find a nearby Italian restaurant with good reviews, then check the current traffic conditions from Nanjing Road to the restaurant"s location. If the traffic is favorable, you can reserve a rental car at 5:00 p.m. for the evening."]

Please complete the following content in the provided format above. You only need to return the "improved_api_calling" list and the "task_description".

input:
api_list = {}
origin_api_calling = {}
output:

B Generated Tool Amount per Time

In the whole process, The amount of generated tools holds up like in Figure 9. Benefits from the pre-generated field information, tools can be generated on a larger scale. Otherwise we may get only around 100 tools by ChatGPT.

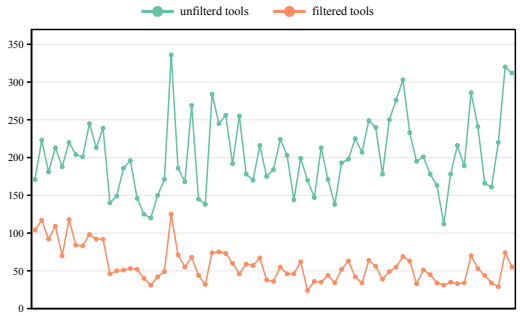

Figure 9: Generated Tool amount per 200 times. Duplicate tools are filtered in real-time.

C Detailed Formulae for Evaluation Metrics

$$Format\ ACC = \frac{amount_{correct\ format}}{amount_{all}}$$

$$Tool\ P = \frac{amount_{correct\ tools}}{amount_{predict\ tools}}$$

$$Tool\ R = \frac{amount_{correct\ tools}}{amount_{gold\ tools}}$$

$$Tool\ F1 = \frac{2 \cdot Tool\ P \cdot Tool\ R}{Tool\ P + Tool\ R}$$

$$Parameter\ P = \frac{amount_{correct\ tools}}{amount_{predict\ tools}}$$

$$Parameter\ R = \frac{amount_{correct\ tools}}{amount_{gold\ tools}}$$

$$Parameter\ F1 = \frac{2 \cdot Parameter\ P \cdot Parameter\ R}{Parameter\ P + Parameter\ R}$$

D Retriever Comparison

Two classical retrievers are tested, the discrete retriever BM25 and the dense retriever DPR. There is something special when training DPR. Retrievers like DPR are used in open-domain QA task before. Generally speaking, a open-domain question may have multiple answers but it's okay to answer only one of them. However in the tool learning task, the retriever is asked to find out all needed tools. Even one missing tools means that the reply can't be generated properly. The loss function in training step is different from that in traditional settings. We try to use contrastive loss but the result is terrible. The gold tools of one query is trained one by one and the retriever seems to only remember the last tool seen. We use the ranking loss finally and get significantly improved result. There should be better prescription remained to be studied.

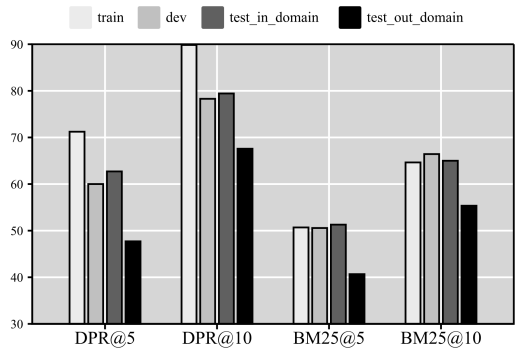

Figure 10: Retriever: DPR v.s. BM25.

As the result in Figure 10 shows, DPR performs better than BM25 which means the retrieval task involves semantic understanding. We can't find out all relevant tools by simple keywords matching. We use DPR as the retriever in agent system finally. How to optimize the retriever deserves further research.

E Generalization Verification

Model	Tool_ACC	Param_Selection_F1	Param_Fill-in_F1
LLaMA2-Chat	**55.49/26.78**/13.49/25.05	**48.80**/35.55/17.76/29.83	27.79/24.02/16.51/23.61
Ours	32.56/26.58/**27.75/31.60**	42.87/**40.20/42.85/44.02**	**32.30/31.76/35.93/33.01**

Table 6: Experiments on Benchmark which is transformed from TOD datasets.

We construct a small tool learning benchmark for initial validation before. In the early stages of exploring the tool learning task, we think it's a good idea to get useful data from other tasks since there is a serious lack of it. This dataset is transformed from task-oriented dialogues (TOD) datasets SGD (Rastogi et al., 2020), Multwoz2.2 (Zang et al., 2020) and Taskmaster (Byrne et al., 2019). The dataset contains 20 tools and 519 instances.

We design 4 styles of prompts (vanilla, role-play, task description and imitaton) to eval through ICL in Table 6:

Vanilla: It's the basic instruction containing a brief introduction to the task.

Role-play: We let LLM play the role of an NLP technologist who is well versed in all kinds of tasks. The Prompt focuses on describing the powerful natural language understanding capabilities of the technologist and does not describe the task specifics.

Task description: In such prompts, we inform LLMs in detail how to fulfill the corresponding tasks. We explain the key concepts involved in detail and describe how to perform them step by step.

Imitation: Given LLM's strong in-context learning or few-shot learning capabilities, this prompt focuses on guiding LLM to learn how to complete the task from the examples given.

It's somewhat unexpected that the model finetuned in Section 4.3.1 performs better than the chat model. Our Seal-Tools might be helpful in instruction tuning to enhance the overall capabilities of LLMs. But when we test the finetuned model in Section 4.2, it perfroms badly. We might get out some conclusions: the model learns how to fill in parameters when given all needed tools in prompt; the model learns how to select tools and what tools are used for when given the retrieved tool list.