ToolGrad: Efficient Tool-use Dataset Generation with Textual "Gradients"

Anonymous ACL submission

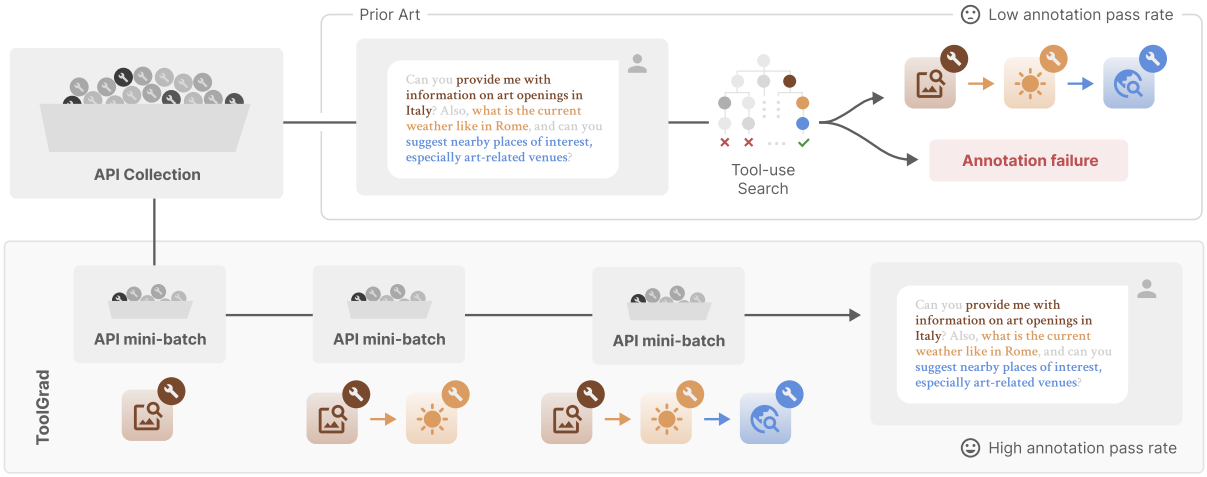

Figure 1: Prior art for tool-use dataset generation (top) starts with a user query, followed by an expensive, failure-prone tool search (*e.g.*, DFS). In contrast, *ToolGrad* (bottom) first generates successful tool-use chains, then annotates corresponding user queries, achieving superior efficiency and a 100% pass rate.

Abstract

Prior work synthesizes tool-use LLM datasets by first generating a user query, followed by complex tool-use annotations like DFS. This inherently leads to inevitable annotation failures and low efficiency in data generation. We introduce *ToolGrad*, an agentic framework that inverts this paradigm. *ToolGrad* first constructs valid tool-use chains through an iterative process guided by textual "gradients", and then synthesizes corresponding user queries. This "answer-first" approach led to *ToolGrad-5K*, a dataset generated with more complex tool use, lower cost, and 100% pass rate. Experiments show that models trained on *ToolGrad-5K* outperform those trained on expensive baseline datasets and proprietary LLMs, even on OOD benchmarks.

1 Introduction

Tool uses empower large language models (LLMs) by interfacing a parametric model with the external world through API calls. For instance, RAG (Lewis et al., 2020), an exemplary tool-use system, demonstrated its impact in reducing LLM hallucination and increasing AI response quality (Shuster et al., 2021). Further studies have extended the concept and use programs and database retrieval to enhance LLMs' math reasoning and fact-checking capabilities (Gao et al., 2023; Augenstein et al., 2024).

In practice, teaching LLM to use tools is non-trivial – its main challenge lies in the dataset. While prior work has collected large-scale API databases (Shen et al., 2023; Yan et al., 2024), we still lack a scalable method to create a pair of "user prompt" and "tool-use chain" for training. Since it is impractical to ask for human annotation at scale, prior work primarily used an agent to search a tool-use path with trial and error. Figure 1 (top) shows a representative annotation approach, which includes two steps: 1) generate a hypothetical user instruction from a sampled API pool, and 2) use a DFS agent to find its tool-use solution. This approach is inherently *inefficient* because its core concept is to distill valuable trajectory from a complex agent exploration for training an LLM. This implies that exploration must be expensive by nature. More importantly, the exploration has no guarantee of annotation success, causing a waste of agent resources.

As a result, such a tool-use dataset generation usually suffers from 1) a high agent cost and 2) a low pass rate.

To address this issue, this work explores an alternative solution paradigm, *i.e.*, we first generate a ground-truth tool-use chain and then annotate its corresponding user prompt. Intuitively, an explicit tool-use solution provides more unambiguous information than a prompt, making the annotation, from tool usage to the use query, much easier and requiring only one LLM step. At the same time, this new problem formulation introduces a new challenge: how can we effectively generate tool-use chains directly from a large-scale API database?

In this work, we introduce ToolGrad, an agentic framework to chain APIs iteratively with mini-batches in a large database. Inspired by a standard ML optimization loop and TextGrad (Yuksekgonul et al., 2024), we design *ToolGrad* to boost textual "gradients" by chaining the best API in each iteration of the framework (Table 1), This is achieved by four modules that perform API proposal, execution, selection, and workflow update, respectively, which resemble the forward inference and backward propagation processes in ML. Using the framework, we created *ToolGrad-5K*, a tool-use dataset that contains 5k samples of user prompts with their corresponding tool calls and AI responses to the user. Compared to a baseline dataset, ToolBench (Qin et al., 2023), *ToolGrad-5K* features more complex tool-use data and was generated with lower cost and a 100% pass rate. We further demonstrate that small LLMs fine-tuned on *ToolGrad-5K* can output SoTA proprietary LLMs. More importantly, our OOD evaluation shows that our models perform comparably or even outperform the same models fine-tuned in distribution and propriety LLMs.

In summary, this work contributes 1) *ToolGrad*, an agentic framework for efficient data generation, 2) *ToolGrad-5K*, a tool-use dataset, and 3) the corresponding fine-tuned models, all of which will be open-sourced to support future research.

2 Related Work

2.1 Tool-use LLMs

Researchers have studied tool-use LLMs in various fields (Patil et al., 2023; Huang et al., 2024b). In NLP, tool-use LLMs have shown improved performance in QA (Zhuang et al., 2023), fact checking (Nakano et al., 2022; Augenstein et al., 2024; Peng et al., 2023) and mathematical reasoning (Gao et al., 2023; Das et al., 2024; Schick et al., 2023). The impact of tool-use LLMs extends beyond NLP, with notable applications in VQA (Gupta and Kembhavi, 2023; Surís et al., 2023), human-computer interaction (De La Torre et al., 2024; Zhou et al., 2024)), and graphic modeling (Huang et al., 2024a; Du et al., 2024).

Datasets play a critical role in advancing the tool-use capability of LLMs. Initial efforts focused on constructing API databases from various resources, such as Hugging Face APIs (Shen et al., 2023) and a community platform (Yan et al., 2024). Given the API databases, there are two primary approaches for creating tool-use datasets that connect user prompts with tool-use actions. The first group of work relies on human annotations (Zhuang et al., 2023; Tang et al., 2023), which is often expensive and difficult to scale up. Therefore, a large portion of work developed synthetic datasets (Yang et al., 2023; Wu et al., 2024). ToolBench (Qin et al., 2023), for example, employs LLMs to generate user queries based on an API database and then performs DFS to search its tool-use solution. τ-bench (Yao et al., 2024) synthesizes multi-turn user interactions with a multi-agent simulation.

This work follows the synthetic data approach and targets the efficiency issue in the data generation process. As we will show in experiments, *ToolGrad* can generate datasets with more complex tool usage with a lower cost and a 100% pass rate.

2.2 Multi-agent Data Optimization

LLMs demonstrated their ability to solve problems via simple prompts. This inspired researchers to create multi-agent collaborative systems for various applications (Park et al., 2023; Wang et al., 2023). For example, AgentCoder (Huang et al., 2023) improves LLM code generation by having a code generator and a verifier work collaboratively. MetaGPT (Hong et al., 2024) further simulates human collaboration in software development by simulating different roles like code writers and planners. Additionally, research shows that agents can self-improve by step-by-step self-criticism (Madaan et al., 2023). Copper (Bo et al., 2024) further formulates the self-refinement problem with RL, and trains an agent that performs better refinement.

Recent studies formulate LLM agents as operators in classical algorithms for data optimization in various downstream applications (Chen et al., 2025; Zhuge et al., 2024). For example, ProTeGi (Pryzant

et al., 2023) optimizes a prompt via LLM-based beam search, which iteratively evaluates, criticizes, and updates an initial prompt design. TextGrad further defined a unified framework for prompt optimization with textual "gradients", and demonstrated its application in a larger domain (Yuksekgonul et al., 2024).

We extend the concept of TextGrad (Yuksekgonul et al., 2024) into tool-use LLM dataset generation. Unlike TextGrad, which optimizes LLMs with better prompts, we aim to generate better datasets to teach LLMs tool usage.

3 Background: Prompt Optimization with Textual "Gradients"

We first review how prior work defines textual "gradients" for prompt engineering in an agentic framework. Note that textual "gradients" are not actual mathematical gradients for numerically optimizing objective functions in ML. Recent work (Yuksekgonul et al., 2024) generalizes the mathematical "gradient" concept into textual feedback from an LLM critic in an agentic framework, which guides LLMs to update a prompt.

Formally, given an LLM, $f(\cdot;\phi)$, instructed by a prompt ϕ, prompt optimization aims to iteratively refine an initial prompt ϕ_0 into an optimized version ϕ_T, so that ϕ_T can better instruct $\texttt{LLM}$ for the given downstream task. This is achieved from an agentic framework with textual "gradient" descents that resemble the standard ML optimization. In specific, given a batch of downstream task data, $\{(x_i, y_i)\}$, an agentic forward process is defined as $\hat{y_i} = f(x_i; \phi_t)$, where $\hat{y_i}$ is the LLM prediction on a given input x_i using prompt ϕ_t on the t_{th} iteration. The loss signal for the "gradient" descent, $\mathcal{L}$, is computed by an LLM agent that criticizes the prediction $\hat{y_i}$. For example, in article summarization, a critic may comment that a generated summary does not fully summarize the core concept for some reason. This results in some textual feedback on the summarization tasks, *i.e.*, the textual "gradients". Lastly, another LLM agent edits the prompt conditioned on the critic's feedback, *i.e.*, $\phi_{t+1} \leftarrow \texttt{LLM}(\phi_t, \mathcal{L})$.

4 *ToolGrad*

Instead of optimizing prompt engineering, *ToolGrad* aims to generate a dataset to teach LLMs tool-use capability. Table 1 summarizes the analogy and difference of *ToolGrad*, compared to TextGrad and ML. In practice, generating a tool-use dataset is more complicated than prompt refinement. Simultaneously updating the model and dataset is an intrinsically challenging analogy to bi-level optimization, as the dataset is used to fine-tune a model, *i.e.*, the internal optimization loop. Therefore, we leverage LLM feedback for the iterative dataset construction without training an LLM on the dataset in each step. To achieve such LLM feedback, *i.e.*, the textual "gradients", we devise four modules that resemble forward and backward propagation in each step.

4.1 Tool-use LLMs: Preliminary

We aim to generate a $\mathcal{D} = \{(q, \mathcal{W}, r)\}$ to finetune a tool-use LLM. q is a user query; $\mathcal{W}$ is an API workflow consisting of a collection of API-use chains: $\mathcal{W} := \{C_1, C_2, \ldots, C_n\}$; and r is the response to q conditioned on $\mathcal{W}$. A chain is defined as a sequence of API execution steps, $C := \texttt{API}_1 \rightarrow \cdots \rightarrow \texttt{API}_n$. An API execution step contains 1) an API id, 2) the input of this API request, and 3) the response from this API request.

An inference model trained on our dataset differs from the ReAct-based tool-use paradigm, *i.e.*, the default function calling method defined in the OpenAI SDK. With this dataset, the model is trained to predict all the tool uses in one shot, while ReAct agents predict one tool use in each LLM step. See more discussion in Appendix A.

4.2 *ToolGrad*: One Iteration Step

Figure 2 visualizes the pipeline of *ToolGrad* in each iteration, which contains four core steps: 1) propose top-k APIs to augment the existing API workflows given a mini-batch of APIs, 2) execute the selected APIs, generating API reports, 3) select the best API to augment the current workflow, and 4) update a workflow with the selected API.

API Proposer. The API proposer, $\texttt{LLM}_{pr}$, takes an API mini-batch as input ($\{\texttt{API}\}^{bs}$ with size bs) and output a list of selected APIs with its corresponding instruction on how to use the API for augmenting the current workflow $\mathcal{W}_t$:

$$\{(\texttt{API}_i, \texttt{inst}_i)\}_{i=0}^{i<m} = \texttt{LLM}_{pr}\left(\{\texttt{API}\}^{bs}; \mathcal{W}_t\right) \quad (1)$$

Parameter m is pre-specified to control the maximal number of API proposals in each step. Note that we prompt $\texttt{LLM}_{pr}$ with simple API configuration, and $\texttt{LLM}_{pr}$ cannot respond with a tool-calling request. This design distills the most valuable APIs

Table 1: An analogy of ToolGrad to conventional machine learning and TextGrad (Yuksekgonul et al., 2024). $\mathcal{D}$ is a tool-use LLM dataset, composed of many triplets of (query, API workflow, response), *i.e.*, $(q, \mathcal{W}, r)$.

	ML	TextGrad	ToolGrad
Model	$f_\theta(x)$	$f(x;\phi)$: prompted by ϕ	$f(x;\mathcal{D})$: fine-tuned on $\mathcal{D}$
Parameter	θ: weights	ϕ: prompt	$\mathcal{D} = \{q, \mathcal{W}, r\}$: dataset
Batches	$\{(x, y)\}$: (query, reply)	$\{(x, y)\}$: (query, reply)	$\{\texttt{API}\}$: a small API set
"Gradients"	$\nabla_\theta \mathcal{L}\big(f_\theta(x), y\big)$	$\texttt{LLM}$ ("criticize it")	$\texttt{LLM}$ ("select the best API")
Optimizer	$\theta_{t+1} \leftarrow \theta_t - \eta\, \nabla_\theta \mathcal{L}$	LLM updater	$\mathcal{W}_{t+1} \leftarrow \mathcal{W}_t.\texttt{add}(\texttt{API}_t)$

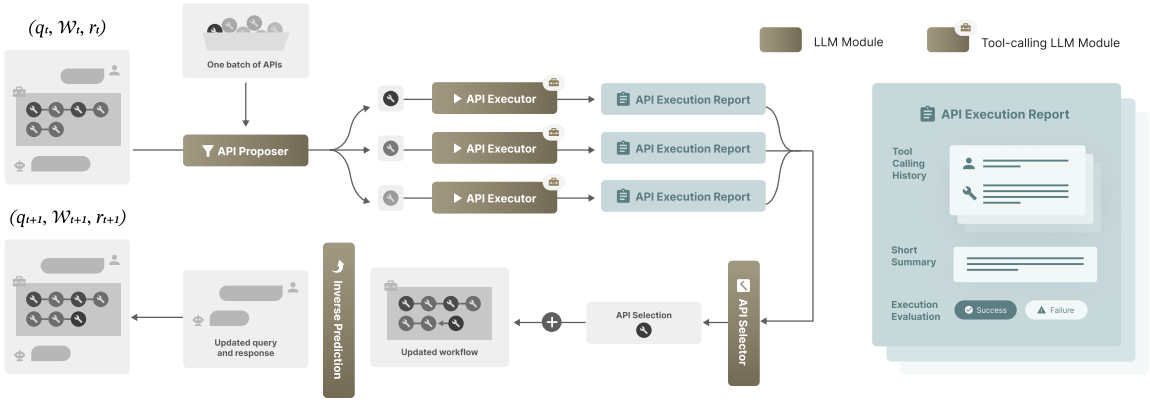

Figure 2: *ToolGrad* Framework. Each iteration starts with $(q_t, \mathcal{W}_t, r_t)$ and a mini-batch of APIs. An API Proposer first predicts up to m APIs, and then m API Executors perform tool calls and return execution reports. An API Selector finds the most valuable API to chain $\mathcal{W}_t \rightarrow \mathcal{W}_{t+1}$. Lastly, an LLM updater is used to predict q_{t+1}, r_{t+1}.

for use in subsequent requests, thereby improving overall system efficiency. Our intuition is that 1) most of the APIs in a randomly sampled batch are irrelevant to the current workflow, and 2) providing simple API configurations is sufficient for an LLM to decide which APIs are worth further in-depth execution. Therefore, m must be much smaller than bs to achieve such efficiency in practice.

API Executor. The API proposals are then sent to m API executors, $\{\texttt{LLM}_{ex}^T\}^m$. $\texttt{LLM}^T$ is denoted as a tool-calling LLM agent that can return tool-calling requests, as opposed to $\texttt{LLM}$, which returns standard responses to the user. $\texttt{LLM}_{ex}^T$ takes an API proposal $(\texttt{API}_i, \texttt{inst}_i)$ as input and return a report,

$$\text{rep}_i = \texttt{LLM}_{ex}^T\left(\texttt{API}_i, \texttt{inst}_i\right). \tag{2}$$

The report contains the following information: 1) a full record of the API request history and 2) a boolean variable showing whether the execution is successful. This is the most expensive step in the *ToolGrad* framework because each selected API is paired with an LLM agent for parallel execution. This verifies the necessity of our API proposer step, which performs filtering, in one LLM step, to avoid redundant API calls.

API Selector. Given a set of execution reports $\{\text{rep}_i\}$, we design an API selector, $\texttt{LLM}_{sel}$, to choose the best API that can augment the current workflow $\mathcal{W}_t$.

$$\begin{aligned} j &= \arg\max_i V\left(\{\text{rep}_i\}^m, \mathcal{W}_t\right) \\ &\sim \texttt{LLM}_{sel}\left(\{\text{rep}_i\}^m, \mathcal{W}_t\right), \end{aligned} \tag{3}$$

where $V(\cdot,\cdot)$ is a hypothetical value function. In practice, instead of defining V and perform $\arg\max V(\cdot,\cdot)$, we use an LLM as its proxy. Intuitively, $\arg\max V(\cdot,\cdot)$ is a process that chooses the most valuable API from the reports, and we hypothesize that an LLM can achieve this task conditioned on the API execution reports, $\{\text{rep}_i\}^m$, and the current workflow, $\mathcal{W}_t$. In addition, we instruct $\texttt{LLM}_{sel}$ to specify which chain $C_k \in \mathcal{W}_t$ the selected API $(\texttt{API}_j)$ augments – or to create a new chain if necessary. Therefore, the following equation shows the API selector step at *ToolGrad*:

$$\begin{aligned} j, k &= \texttt{LLM}_{sel}\left(\{\text{rep}_i\}^m, \mathcal{W}_t\right), \\ &\text{where } \begin{cases} j \text{ is the selected } \texttt{API} \text{ id for } \texttt{API}_j, \\ k \text{ is the chain id for } C_k. \end{cases} \end{aligned} \tag{4}$$

The API selection is the core step that performs the "gradient" computation in our optimization loop (Table 1). As opposed to the LLM critic step that uses textual feedback as "gradient", our API selector chooses a discrete API to augment $\mathcal{W}_t$ as "gradients" of data generation in *ToolGrad*.

Workflow Updater. j and k from the API executor provide clear information on 1) which API from the mini-batch the workflow updater should use and 2) where (at which chain) the updater should append the API to. Therefore, the workflow updating process can be clearly defined as follow without using LLMs.

$$\mathcal{W}_{t+1} \leftarrow \mathcal{W}_t.\texttt{add}(\texttt{API}_j, C_k) \tag{5}$$

On the other hand, once $\mathcal{W}_t$ is updated to $\mathcal{W}_{t+1}$, we should also update (q_t, r_t) to maintain the coherence of the sample triple $(q, \mathcal{W}, r)$. Therefore, in the workflow updating step, we perform the following LLM step:

$$q_{t+1}, r_{t+1} = \texttt{LLM}_{\textit{updater}}(\mathcal{W}_{t+1}) \tag{6}$$

Intuitively, this step resembles summarization tasks that convert detailed texts (*i.e.*, a tool-use workflow) to ambiguous messages (*i.e.*, a user query and its response). This inverse prediction process is much more straightforward than the standard forward pass that explores answers with a given user query: $\mathcal{W}, r = \texttt{LLM}_{\textit{DFS}}(q)$, where $\texttt{LLM}_{\textit{DFS}}$ is an agent using DFS (Qin et al., 2023).

4.3 Sampling Negative APIs

Given the $(q, \mathcal{W}, r)$ with the ground-truth tool uses, we post-process it by sampling negative tools. The objective is to simulate a real-world use scenario where an agent can access more APIs than necessary. Prompting the LLM with every API configuration is impractical given our API database's size (8k). Therefore, we aim to simulate a benchmark for an RAG-like agent, in which the agent first samples top-p APIs based on the text-embedding similarity and then prompts an LLM with the p APIs only. Formally, given a ground truth set $\{\mathcal{W}\}^n$ of n positive APIs, we sample the top-$(p-n)$ APIs most similar to these positives as our negative samples.

4.4 Generation Configuration

In this work, we choose the number of API proposals as $m = 3$, and the API batch size $bs = 50$. Each generation loop takes 10 iteration steps, which we observed is sufficient to generate complex API-use workflows. We chose $p = 20$ when sampling negative APIs and *gpt-4.1-mini* as our LLM for data generation.

5 Experiments

We conducted three experiments to demonstrate the efficiency of the *ToolGrad* framework in various aspects. This includes 1) the high-quality but low-cost dataset generation, 2) high performance of models trained on *ToolGrad-5K*, both in distribution and OOD.

5.1 Efficiency of Generating *ToolGrad-5K*

Using *ToolGrad*, we collected *ToolGrad-5K*, a dataset containing 5K triplets of $(q, \mathcal{W}, r)$. This experiment presents details of our generation and demonstrates the efficiency of our generation.

API Library. We used the API library provided by ToolBench (Qin et al., 2023), which contains approximately 16K APIs. We found some API names and their corresponding configuration are not well annotated (*e.g.*, APIs named as "test_v5", "test_for_test", etc.), which negatively affects our generation. Therefore, we used *gpt-4o-mini* to filter these APIs with low-quality annotations. 8,691 APIs remain in the API library for *ToolGrad*.

Baselines. We chose (Qin et al., 2023)'s DFS-based data generation approach as our baseline. *ToolGrad-5K* shares the same API databases with ToolBench but differs in the data generation framework. They chose the query-first generation, followed by the DFS answer annotation. This helps us control many factors and leaves the data generation framework as an independent factor for fair comparison in the experiment.

Metrics. We incorporated four metrics to measure the data generation efficiency of a given framework. Two metrics are used to evaluate the data generation quality: 1) *Pass rate* and 2) *the number of ground-truth tool uses*. The pass rate determines whether the triplet $(q, \mathcal{W}, r)$ can be successfully generated. The number ground-truth tool uses $n = ||\mathcal{W}^n||$. The remaining two metrics show the cost of data generation: 1) *# of LLM calls* and 2) *# of tool calls*.

Results. We evaluate the data generation efficiency of the *ToolGrad* framework by considering both 1) the generation quality (*i.e.*, generation pass rate and the trace of generated tool-use chains) and 2) the generation cost (*i.e.*, the number of LLM steps and tool calls). Table 2 summarizes the re-

Table 2: Generation efficiency comparison between DFS (Qin et al., 2023) and *ToolGrad*. *: We only count the trace of passed annotations. The overall discounted value will be $3.3 \times 63.8\% = 2.1$.

	DFS	*ToolGrad*
Pass rate (%) ↑	63.8	**100.0**
# of gt tool uses ↑	3.3*	**6.1**
LLM cost ↓	64.5	**45.9**
Tool cost ↓	34.3	**<30.0**

sults compared to the baseline method introduced in ToolBench (Qin et al., 2023). ToolGrad achieves a perfect 100.0% pass rate – a significant improvement from 63.8% for DFS, while producing more complex chains (an average of 6.1 ground-truth tool uses vs. 3.3 for DFS). More importantly, *ToolGrad* cuts down on the generation cost: LLM invocations drop from 64.5 to 45.9, and tool-use steps fall from 34.3 to below 30.0. Note that we did not explicitly track the number of tool-use steps, but we can prove its maximal number is 30, since *ToolGrad* has 3 tool-use LLM modules per iteration and uses 10 iterations in total. The results demonstrate the high efficiency of *ToolGrad* for data generation: it generates more complex tool-use chains with higher pass rate and lower cost.

5.2 *ToolGrad-5K* Improves LLM Tool Usage

We then aim to understand the effectiveness of *ToolGrad-5K* to teach LLMs' tool-use capability.

Model Training. We used 90% of *ToolGrad-5K* for training and the rest for testing. We trained the models with three epochs, using three Gemma-3 models (Team, 2025) (1B, 4B, and 12B). We use SFTTrainer on HuggingFace, and more detailed training configurations will be released with our code. We name the fine-tuned models ToolGrad-1B, ToolGrad-4B, and ToolGrad-12B, respectively. Appendix C shows the GPU budgets required to train the models in this paper.

Baselines. We compare our models with SoTA general-purpose models. We chose three proprietary LLMs (gpt-4.1, gemini-2.5-flash[1], and claude-3.7-sonnet) and two open-sourced models (deepseek-v3 and Llama-4-Maverick) as our baseline models. We disabled reasoning for those models but additionally studied the effect of reasoning by benchmarking two reasoning models that support tool use. We chose to compare o4-mini and gemini-2.5-flash with their "corresponding" base model[2] (gpt-4.1-mini, and gemini-2.5-flash).

[1]"gemini-2.5-pro" does not support disabled thinking.

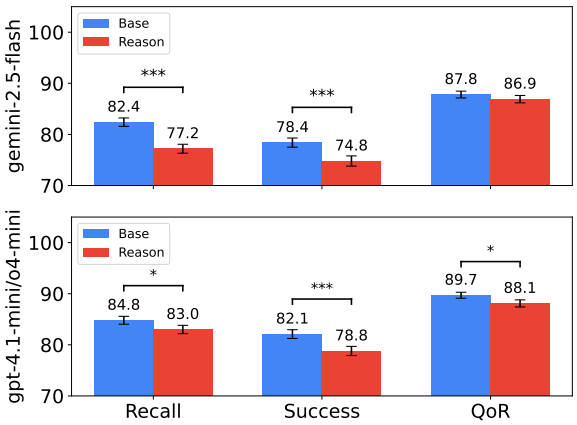

Figure 3: Comparison of base and reasoning Gemini / GPT models on *ToolGrad-5K*. The error bar represents the standard error. $*$ and $***$ denote as $p < .05$, $p < .001$ in the paired t-test, respectively.

Metrics. We consider two metrics in these experiments: 1) tool recall, 2) success rate, and 3) quality of response (QoR), all scaled from 0 to 100. The tool recall evaluates whether the agent can retrieve the correct tool given a user query. The success rate further examined how many of them are called successfully. Formally, the success rate is defined as the number of recalled APIs that receive a successful response divided by the total number of ground-truth APIs. Therfore, the success rate is always lower than or equal to the tool recall by definition. The QoR further implicitly evaluates whether the successful call provides valuable contexts for an LLM to formulate a sense-making response. The score is rated by an LLM judge ("gpt-4.1"), which is prompted with 1) a user query, 2) a tool-use trace, and 3) a textual response to the user query. We use a unified response writer ("gpt-4.1-mini") in this study to eliminate the bias introduced by the different LLMs' "writing" skills.

Results. Table 3 shows that our models, even the 1B model, outperform all baseline models in all metrics. The tool recall of our models reaches ∼99%, demonstrating that the fine-tuned models can always retrieve the correct tool(s) to call while the baseline LLMs can only retrieve 80∼85%. Furthermore, our models can receive a successful response from $> 95\%$ of ground-truth APIs, while all baselines can only achieve ∼80%. Our models

[2]Since these are proprietary models, we are not able to officially pair the reasoning model with its base model. We chose this mapping based on the names and release dates.

Table 3: LLM Benchmark on *ToolGrad-5K*. **The best score** is highlighted in each metric across all models.

	ToolGrad 1B	ToolGrad 4B	ToolGrad 12B	gpt-4.1	gemini-2.5 flash	claude-3.7 sonnet	deepseek v3	llama-4 maverick
Tool recall	98.8	99.3	**99.6**	84.1	82.4	84.9	83.9	83.4
Success rate	95.5	96.4	**96.8**	78.6	78.4	79.6	79.4	80.6
QoR	93.7	95.3	**95.8**	87.2	87.8	88.3	87.8	87.9

Table 4: OOD Experiment setups. Our models (standard) are evaluated OOD, and baselines (ReAct & DFS) are evaluated in distribution.

Base model	Llama-3.1/3.2 (1B, 3B, 8B)	
Train set	ToolGrad	ToolBench
Size	5k	197k
Eval set	ToolBench	ToolBench
Framework	Standard	ReAct DFS

also show dominant performance on QoR. This result demonstrates that *ToolGrad-5K* can effectively teach LLMs' tool-use capability – a small LLM learned on the dataset can significantly outperform large LLMs.

Figure 3 shows the performance comparison between a base model and its reasoning model. Surprisingly, the base model consistently outperforms its reasoning model (*e.g.*, tool recall: $+5.2\%$, success rate: $+3.6\%$, QoR: $+0.9\%$ for "gemini-2.5-flash"). A paired t-test of the data shows a significant difference between the base and reasoning model's performance in two metrics of the Gemini model and all metrics for the GPT model. We further investigated related benchmarks and found similar surprising results in BFCL (Yan et al., 2024), where 1) "gpt-4o" outperforms "o1" (score: +2.63), and 2) "gemini-2.0-flash" outperforms "gemini-2.0-flash-thinking" (score: +1.31) when using prompt engineering. While it is out of scope to investigate the tool-use capability of reasoning models further, we hope our findings and dataset can inspire future exploration of this issue.

5.3 OOD Evaluation on ToolBench

We further evaluate the models trained on *ToolGrad-5K* using an out-of-distribution (OOD) benchmark, ToolBench (Qin et al., 2023). We will show that our models achieve better performance, as well as lower inference cost, compared to those trained on ToolBench (*i.e.*, the in-distribution evaluation of the baseline models).

Setups. Table 4 shows our configuration for model training and inference in experiments. We use Llama-3 series models (Grattafiori et al., 2024) (*i.e.*, llama-3.2-1B, llama-3.2-3B and llama-3.1-8B) as base models because the ToolBench source code is more compatible with Llama compared to Gemma. For each base LLM, we have two training setups: 1) fine-tuned on *ToolGrad-5K*, which learns to use tools with standard LLM prompts and inference framework. 2) fine-tuned on ToolBench, which learns to use tools with ReAct/DFS frameworks. All models are evaluated on ToolBench-I3, the most universal and challenging test set in ToolBench. This implies that we perform OOD evaluation on our model while our baseline models are evaluated in distribution. We test the *ToolGrad* models with the standard framework, the ToolBench models with both ReAct and DFS, respectively. As a result, for each base LLM condition, we report three performance values (*i.e.*, "standard" from ToolGrad, "ReAct", and "DFS" from ToolBench).

It is important to note the inference framework is another moving factor in addition to the fine-tuning datasets. This is mainly because the two datasets we compare are deeply coupled with the inference frameworks. This difference gives our baseline methods advantages, as intuitively illustrated in Figure 4. We conduct additional experiments to verify that the more complicated and costly inference framework ReAct / DFS is an advantage for performance: GPT models (gpt-4.1-nano, gpt-4.1-mini, gpt-4.1) in three different inference frameworks achieve DFS > ReAct > Standard in performance.

Metrics. We report metrics in two dimensions that cover performance and costs. Performance-wise, we use QoR, as both tool recall and success rate are infeasible to compute here because ToolBench does not contain ground-truth tool labels. As shown in Table 3, the model ranking follows similar orders in both QoR and "success rate". This result shows that our prompt design for the LLM judges

Table 5: Performance & Cost Results on ToolBench. "# LLMs" and "# Tools" are denoted as the number of LLM steps and tool-calling requests, respectively. In each column, we highlight bold **the best value** with underline the second best value. We also highlighted scores evaluated on the OOD dataset.

	Evaluation Score↑						Inference Cost↓	
	Llama 3.2-1B	Llama 3.2-3B	Llama 3.1-8B	gpt-4.1 nano	gpt-4.1 mini	gpt-4.1	# LLMs	# Tools
Standard (Ours)	**23.95**	**23.00**	**26.34**	26.14	25.69	26.36	**1.00**	**2.69**
ReAct	17.92	19.81	21.30	30.76	33.74	34.42	6.60	3.71
DFS	22.04	21.97	23.94	**31.33**	**37.10**	**38.56**	30.78	19.01

can provide sensing-making evaluation scores in the evaluation. Regarding the cost, we report the number of LLM steps and tool-use requests during the model inference.

Results. Table 5 shows the results on performance and cost. Firstly, the inference cost shows that DFS is the most expensive framework, followed by ReAct, and our framework is the cheapest to run. This result is coherent with the performance results of GPT models, where the most expensive framework achieves the best performance. This verifies that our experiment design favors our baseline conditions (the ToolBench ReAct and DFS models). On the other hand, the results of the fine-tuned Llama-3 are "counter-intuitive". Despite the framework disadvantage, the standard model trained on *ToolGrad-5K* consistently outperforms the ReAct and DFS condition model. Additionally, comparing all scores in the "standard" row, we find that a fine-tuned 8B model can achieve comparable performance, on an OOD benchmark, with the gpt-4.1 model, and better than gpt-4.1-mini and gpt-4.1-nano. The results reinforce that *ToolGrad-5K* can better teach the LLM tool usage even on OOD benchmark.

5.4 Discussion

Contaminated Data with Unsolvable Queries. The performance of gpt-4.1 shows a large drop from our benchmark (Table 3) to ToolBench (Table 5). Its main reason is that many queries generated by ToolBench are not solvable. As shown in Figure 1, ToolBench proposes a user query from an API set, and the proposed query cannot guarantee its feasibility. The Toolbench training set includes every sample, regardless of whether a DFS succeeds or fails. This results in a contaminated training set where a trained LLM needs to imitate low-value experience replays. In comparison, the query generated from our framework, *ToolGrad*, is grounded on a verified answer, and can guarantee the resolvability.

Data Filtering vs. Inverse Prediction. One common approach to eliminate such contamination is to filter out failure samples (Du et al., 2024). This approach is also suboptimal – It limits an agent's learning space to problems that a teacher agent can solve. As a result, a student cannot outperform a teacher model (*e.g.*, ToolLlama fails to beat GPT-4 (Qin et al., 2023)). In contrast, *ToolGrad* shows potential in bootstrapping an agent intelligence with our unique design of answer-first data generation. For example, Table 3 shows that "gpt-4.1" can only call 78.6% of the APIs successfully on the queries in *ToolGrad-5K*, generated by "gpt-4.1-mini". We further showed that even 1B model trained on "gpt-4.1-mini"-generated data can outperform "gpt-4.1".

Reasoning Agent Framework. While Table 5 shows that our models outperform ToolLlama with the reasoning agent framework, the GPT-series models still show superior performance on ReAct / DFS frameworks, which also aligns with the recent study (Lu et al., 2025). This indicates our dataset quality outweighs the framework's disadvantage. Meanwhile, we encourage future work to extend *ToolGrad* to formulate a training set for teaching reasoning agents tool usage.

6 Conclusion

This work introduces *ToolGrad*, an agentic framework for efficient tool-use dataset generation. Our core concept is to first generate tool-use answers with textual "gradients", followed by query generation. We further contribute *ToolGrad-5K*, a dataset containing complex tool usage, but was generated with a lower cost and 100% pass rate. Experiments show that models trained on *ToolGrad-5K* outperform those on expensive baseline datasets and proprietary LLMs, even on the OOD benchmark.

Limitations

Our models are limited in inferencing with the ReAct/DFS framework because our fine-tuning dataset does not contain reasoning examples. Additionally, this work focuses on demonstrating the usage of our dataset with supervised fine-tuning. Recent exploration highlights the benefit of teaching LLM tool usage with reinforcement learning (RL) (Qian et al., 2025). The value of our generated datasets for RL is underexplored.

References

Isabelle Augenstein, Timothy Baldwin, Meeyoung Cha, Tanmoy Chakraborty, Giovanni Luca Ciampaglia, David Corney, Renee DiResta, Emilio Ferrara, Scott Hale, Alon Halevy, and 1 others. 2024. Factuality challenges in the era of large language models and opportunities for fact-checking. *Nature Machine Intelligence*, 6(8):852–863.

Xiaohe Bo, Zeyu Zhang, Quanyu Dai, Xueyang Feng, Lei Wang, Rui Li, Xu Chen, and Ji-Rong Wen. 2024. Reflective multi-agent collaboration based on large language models. *Advances in Neural Information Processing Systems*, 37:138595–138631.

Minghui Chen, Ruinan Jin, Wenlong Deng, Yuanyuan Chen, Zhi Huang, Han Yu, and Xiaoxiao Li. 2025. Can textual gradient work in federated learning? *arXiv preprint arXiv:2502.19980*.

Debrup Das, Debopriyo Banerjee, Somak Aditya, and Ashish Kulkarni. 2024. Mathsensei: A tool-augmented large language model for mathematical reasoning. *Preprint*, arXiv:2402.17231.

Fernanda De La Torre, Cathy Mengying Fang, Han Huang, Andrzej Banburski-Fahey, Judith Amores Fernandez, and Jaron Lanier. 2024. LLMR: Real-Time Prompting of Interactive Worlds Using Large Language Models. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, CHI '24, New York, NY, USA. Association for Computing Machinery.

Yuhao Du, Shunian Chen, Wenbo Zan, Peizhao Li, Mingxuan Wang, Dingjie Song, Bo Li, Yan Hu, and Benyou Wang. 2024. BlenderLLM: Training Large Language Models for Computer-Aided Design With Self-Improvement. *Preprint*, arXiv:2412.14203.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Tanmay Gupta and Aniruddha Kembhavi. 2023. Visual Programming: Compositional Visual Reasoning Without Training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 14953–14962.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. Metagpt: Meta programming for a multi-agent collaborative framework. *Preprint*, arXiv:2308.00352.

Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Ian Huang, Guandao Yang, and Leonidas Guibas. 2024a. BlenderAlchemy: Editing 3D Graphics With Vision-Language Models. *ArXiv Preprint ArXiv:2404.17672*.

Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Zhenqiang Gong, and Lichao Sun. 2024b. MetaTool Benchmark for Large Language Models: Deciding Whether to Use Tools and Which to Use. In *The Twelfth International Conference on Learning Representations*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc.

Pan Lu, Bowen Chen, Sheng Liu, Rahul Thapa, Joseph Boen, and James Zou. 2025. Octotools: An agentic framework with extensible tools for complex reasoning. *Preprint*, arXiv:2502.11271.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. 2022. Webgpt: Browser-assisted question-answering with human feedback. *Preprint*, arXiv:2112.09332.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large Language Model Connected With Massive APIs. *Preprint*, arXiv:2305.15334.

Baolin Peng, Michel Galley, Pengcheng He, Hao Cheng, Yujia Xie, Yu Hu, Qiuyuan Huang, Lars Liden, Zhou Yu, Weizhu Chen, and 1 others. 2023. Check your facts and try again: Improving large language models with external knowledge and automated feedback. *arXiv preprint arXiv:2302.12813*.

Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. 2023. Automatic Prompt Optimization With "Gradient Descent" and Beam Search. *Preprint*, arXiv:2305.03495.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. 2025. Toolrl: Reward is all tool learning needs. *Preprint*, arXiv:2504.13958.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-World APIs. *Preprint*, arXiv:2307.16789.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems*, volume 36, pages 68539–68551. Curran Associates, Inc.

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. *Advances in Neural Information Processing Systems*, 36:38154–38180.

Kurt Shuster, Spencer Poff, Moya Chen, Douwe Kiela, and Jason Weston. 2021. Retrieval Augmentation Reduces Hallucination in Conversation. *Preprint*, arXiv:2104.07567.

Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. ViperGPT: Visual Inference via Python Execution for Reasoning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 11888–11898.

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

Gemma Team. 2025. Gemma 3 Technical Report. *Preprint*, arXiv:2503.19786.

Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Ma, and Yitao Liang. 2023. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. *arXiv preprint arXiv:2302.01560*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc.

Qinzhuo Wu, Wei Liu, Jian Luan, and Bin Wang. 2024. ToolPlanner: A Tool Augmented LLM for Multi Granularity Instructions With Path Planning and Feedback. *Preprint*, arXiv:2409.14826.

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

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. In *Advances in Neural Information Processing Systems*, volume 36, pages 71995–72007. Curran Associates, Inc.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *Preprint*, arXiv:2406.12045.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations*.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. 2024. TextGrad: Automatic "Differentiation" via Text. *Preprint*, arXiv:2406.07496.

Zhongyi Zhou, Jing Jin, Vrushank Phadnis, Xiuxiu Yuan, Jun Jiang, Xun Qian, Jingtao Zhou, Yiyi Huang, Zheng Xu, Yinda Zhang, Kristen Wright, Jason Mayes, Mark Sherwood, Johnny Lee, Alex Olwal, David Kim, Ram Iyengar, Na Li, and Ruofei Du. 2024. InstructPipe: Building Visual Programming Pipelines With Human Instructions Using LLMs. *Preprint*, arXiv:2312.09672.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. 2023. ToolQA: A Dataset for LLM Question Answering With External Tools. In *Advances in Neural Information Processing Systems*, volume 36, pages 50117–50143. Curran Associates, Inc.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. 2024. Gptswarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*.

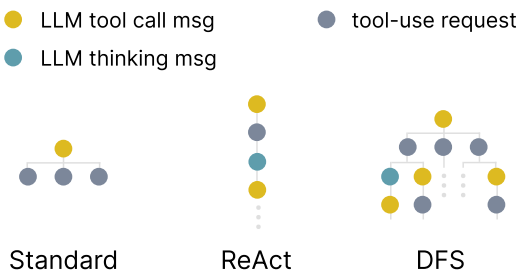

Figure 4: A visualized comparison among standard, ReAct, DFS inference frameworks.

Appendix

A Three Frameworks: Standard, ReAct and DFS

This work involves three different inference frameworks: 1) standard (*i.e.*, the *ToolGrad* inference framework), 2) ReAct, 3) DFS. Figure 4 visualizes their differences. In the standard framework that *ToolGrad* models use, an LLM is trained to predict multiple tool call requests in one shot, and thus, there is one single LLM step in the inference time. On the other hand, ToolLlama models (Qin et al., 2023) are trained to incorporate the ReAct (Yao et al., 2023) (a.k.a, CoT (Wei et al., 2022)) framework. In the ReAct framework, each LLM step returns a single tool call request. The LLM and tool use is called alternatively, with an optional thinking step inserted in between. The DFS framework extends the ReAct concept by enabling a tree search.

B License For Artifacts

This work has used ToolBench for data generation and benchmark. ToolBench is licensed under the Apache License 2.0, so we argue that our use is considered a fair use of the artifact.

Additionally, we will also open-source our source code, dataset, and fine-tuned models. These artifacts will be under a BY-CC license.

C Model Training Budgets

In this work, we train LLMs using SFTTrainer on Hugging Face, configured with "flash_attention_2" and gradient checkpointing. We use "adamw_8bit" and "bfloat16" for training.

We have trained three Gemma-3 models (1B, 4B, 12B) on *ToolGrad-5K*. Training the 1B and 4B models take 1.5 and 3.5 GPU hours on A100, respectively. The 12B model costs 4.5 GPU hours on H200.

We also trained three Llama-3 models (1B, 3B, 8B) on both *ToolGrad-5K* and ToolBench. On *ToolGrad-5K*, it takes 1 and 2.5 GPU hour(s) using A100 to train 1B and 3B models, respectively. The 8B model costs 4 GPU hours on H200. On ToolBench, it costs 13 GPU hours and 40 GPU hours on A100 to train 1B and 3B models, respectively. It costs 44 GPU hours to train the 8B models on H200.