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

Zhongyi Zhou[1,2,3], **Kohei Uehara**[2], **Haoyu Zhang**[2], **Jingtao Zhou**[1], **Lin Gu**[3,4], **Ruofei Du**[1], **Zheng Xu**[1], **Tatsuya Harada**[2,3]
[1]Google, [2]The University of Tokyo, [3]RIKEN AIP, [4]Tohoku University
zhongyi.zhou.work@gmail.com, harada@mi.t.u-tokyo.ac.jp

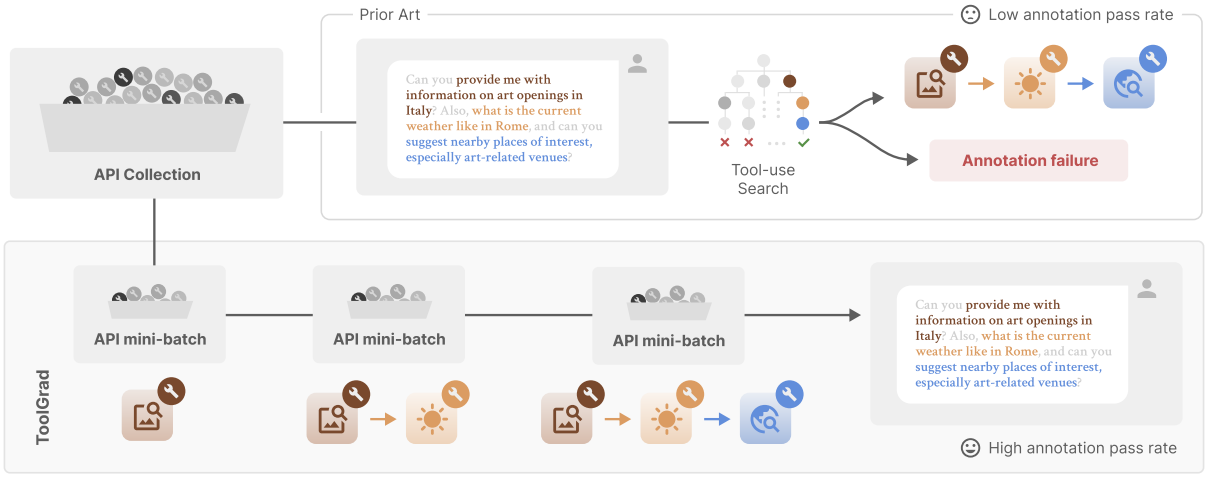

Figure 1: Prior art for tool-use dataset generation (top) starts with a user query, followed by an expensive, failure-prone tool search (*e.g.*, DFS). In contrast, *ToolGrad* (bottom) first generates successful tool-use chains, then annotates corresponding user queries, achieving superior efficiency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

Abstract

Prior work synthesizes tool-use LLM datasets by first generating a user query, followed by complex tool-use annotations like depth-first search (DFS). This 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-500*, a dataset generated with more complex tool use, lower cost, and almost 100% pass rate. Experiments show that *ToolGrad* models outperform those trained on expensive baseline datasets and proprietary LLMs. The ToolGrad source code, dataset, and models are available at https://github.com/zhongyi-zhou/toolgrad.

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 trajectories from a complex agent exploration for training an LLM. This implies that exploration must be expensive by nature.

Findings of the Association for Computational Linguistics: ACL 2026, pages 19040–19056
July 2-7, 2026

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-500*, a tool-use dataset that contains 0.5k 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-500* features more complex tool-use data and was generated with lower cost and an almost 100% pass rate. We further demonstrate that small LLMs fine-tuned on *ToolGrad-500* can outperform or match SoTA proprietary LLMs, even on out-of-distribution (OOD) datasets with unseen tools.

In summary, this work contributes 1) *ToolGrad*, an agentic framework for efficient data generation, 2) *ToolGrad-500*, 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. The latest work further showed that synthetic queries may misalign with human queries in the real world, and thus created new postprocessing modules to rewrite generated user queries (Wu et al., 2024; Zhang et al., 2024).

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 higher 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 prob-

	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"	$\nabla_\theta\mathcal{L}\big(f_\theta(x),y\big)$	LLM ("criticize it")	LLM ("select the best API")
Optimizer	$\theta_{t+1}\leftarrow\theta_t-\eta\,\nabla_\theta\mathcal{L}$	LLM updater	$\mathcal{W}_{t+1}\leftarrow\mathcal{W}_t.\texttt{add}(\texttt{API}_t)$

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.*, $(q,\mathcal{W},r)$.

lem 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 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 differences 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 finetune 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

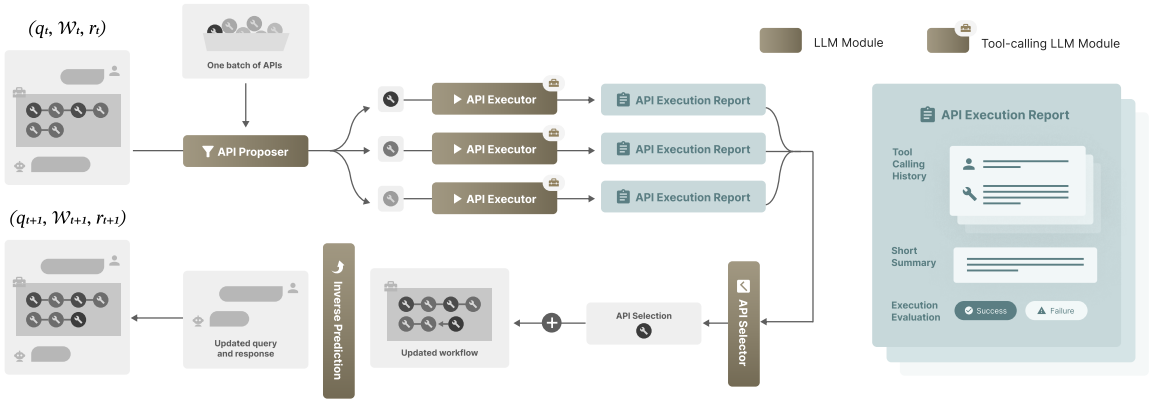

Figure 2: *ToolGrad* Framework. Each iteration starts with $(q_t, \mathcal{W}_t, r_t)$ and a mini-batch of APIs. An API Proposer first predicts up to m APIs, and then m API Executors perform tool calls and return execution reports. An API Selector finds the most valuable API to chain $\mathcal{W}_t \to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

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 E.

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 outputs a list of selected APIs with their corresponding instructions 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 a simple API configuration, and $\texttt{LLM}_{pr}$ cannot respond with a tool-calling request. This design distills the most valuable APIs 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,

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

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 $\mathcal{W}_t$.

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

where $V(\cdot,\cdot)$ is a hypothetical value function. In practice, instead of defining V and performing $\arg\max V(\cdot,\cdot)$, 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*:

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

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 follows 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: $\mathcal{W}, r = \texttt{LLM}_{DFS}(q)$, where $\texttt{LLM}_{DFS}$ 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 an RAG-like use case in our dataset, 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 $p = 10$ when sampling negative APIs and *gemini-2.5-flash-lite* as our LLM for data generation. We chose the lite model for its cost-effectiveness, and its benefit in low-latency response capability. The low-latency benefits are crucial for our framework to generate data within a reasonable amount of time. The framework for 500 times using different seeds, constituting *ToolGrad-500*. We then further format the generated samples into a supervised finetuning set for single-turn tool uses, equivalent to the tasks defined in Berkeley Function Calling Leaderboard (BFCL) v1/v2 (Patil et al., 2023). The LLM is provided with an OpenAI-style tool-use definition and trained to output a Python-style tool use. Appendix C provides more details on our prompt templates.

5 Experiments

We conducted experiments to demonstrate that *ToolGrad* 1) generates high-quality data with low cost, and 2) leads to a finetuned model with better tool use performance.

5.1 Setups

Model training and baselines. We then finetune three Gemma-3 models (1B, 4B and 12B) on these two datasets using supervised finetuning. More training configs are documented at Appendix B.1. We chose baselines from two aspects. We first chose ToolBench (Qin et al., 2023) as our query-first baseline dataset, and their DFS framework as our query-first data generation framework. We evaluate the quality of frameworks both explicitly (generation cost and generated data complexity) and implicitly (model performance trained on their generated datasets). To achieve a fair implicit comparison, we also finetune the Gemma-3 models on ToolBench datasets, and report results based on the finetuned models. We also consider model-wise baselines, where we experimented with Gemini-2.5, GPT-5 and Claude-4.5 series models as representatives of SoTA proprietary LLMs. In addition, we chose ToolACE (Liu et al., 2025) and Hammer (Lin et al., 2024) as baselines that represent finetuned models specifically for tool uses.

Evaluation benchmarks. We evaluated the fine-

	DFS	*ToolGrad*
Pass rate (%) ↑	63.8	**99.8**
# of gt tool uses ↑	2.1	**3.4**
LLM cost ↓	64.5	**63.9**
Tool cost ↓	34.3	**20.0**

Table 2: Generation efficiency comparison between DFS (Qin et al., 2023) and *ToolGrad*.

tuned models on two benchmarks. We first report results on ToolBench-I3 (Qin et al., 2023), the most challenging track in ToolBench with cross-category tool use. Therefore, we report our model performance on BFCL (Patil et al., 2023). Note that there is minimal tool definition overlap between ToolBench and BFCL, so this evaluation helps us understand whether our models can successfully use unseen tools. Our evaluation focuses on BFCL-v1/v2, the single-turn tool-use track, because multi-turn tool use and agent use (the latest v3/v4 track) are out of scope for our generation framework. We leave it as future work to explore enhancing agent performance with the *ToolGrad*-generated dataset.

Metrics. We report two groups of metrics to evaluate a given dataset generation approach. To evaluate the *cost* of data generation, we report 1) the pass rate, 2) the number of ground-truth tool uses, which measures the complexity of generated data, 3) the number of LLM/tool calls, and 4) the LLM API cost. We also report the *performance* of models finetuned on the given dataset on ToolBench and BFCL. We reused their corresponding evaluation metrics. Note that on ToolBench, we use absolute LLM judge scores instead of the original win rate metric. Our absolute scoring system fits more on our experiments with multiple baselines. Appendix D.1 provides more details on our design of the LLM judge, including the prompt design.

5.2 Main Results

Data Efficiency. Table 2 summarizes the cost-performance results comparison between our answer-first data generation framework, ToolGrad, and a DFS-based query-first approach on ToolBench (Qin et al., 2023). ToolGrad achieves 99.8% pass rate – a significant improvement from 63.8% for DFS, while producing more complex chains (an average of 3.4 ground-truth tool uses vs. 2.1 for DFS). More importantly, *ToolGrad* cuts down on the generation cost: LLM invocations drop from 64.5 to 63.9, and tool-use steps fall from 34.3 to 20.0. The results demonstrate the high efficiency of *ToolGrad* for data generation: it generates more complex tool-use chains with a higher pass rate and lower cost.

We reviewed the failure log and found that the agent failed to receive a successful response from 3 selected APIs across all 10 iterations, and thus saved an empty data sample. In practice, this case happens extremely rarely (0.2%).

Results on ToolBench. We then report how effective our "cheap" dataset can be used to teach LLM tool usage. Table 3 compares single-turn tool-use performance between our finetuned small LLMs and SoTA proprietary LLMs. Results show that our 12B and 4B models ranked first and second, outperforming even our teacher model, "gemini-2.5-flash-lite".

Table 4 further compares our models on ToolGrad to the same models trained on ToolBench and the base Gemma models, using the ToolBench test set. We first compared ToolGrad models with base Gemma models, and results show that ToolGrad can effectively enhance Gemma's tool use capability (1B: +13.1, 4B: +6.4, 12B: +9.8). We then compare our models with ToolBench-trained models, which are 1) trained on more expensive datasets (see training budget data Appendix B.2), 2) use an advanced inference framework, and 3) evaluated in-distribution with their test set. Despite the "unfair" setup that strongly favors our baselines, the ToolGrad models still outperform the corresponding ToolBench models (with a tie on the 4B model). Additional discussion on Table 4 is available in Appendix D.2.

Note that ToolBench evaluation heavily rely on LLM judges, so we conducted a small-scale human evaluation to test its alignment with human scores. We randomly selected 8 queries and asked two human raters to evaluate the answers from the 12 models in Table 3, yielding 96 ratings per rater. Further details regarding the study are available in Appendix D.3. Our analysis indicates that the averaged judge scores correlate strongly with human ratings ($\rho = .88, p < .001$). This observation aligns with existing literature (Chan et al., 2023; Wang et al., 2025) and serves to validate both our judge design and the model performance results on ToolBench.

Results on BFCL. Figure 3 shows our evaluation results on BFCL. We only report overall score of single-turn tool uses in the figure. The score breakdowns are available in Table 5 and Table 6.

Model	ToolGrad			Gemini 2.5			Claude 4.5			GPT-5		
	1B	4B	12B	lite	flash	pro	haiku	sonnet	opus	nano	mini	base
Score	14.1	<u>17.6</u>	**19.6**	6.9	8.5	11.4	12.8	13.5	15.4	15.4	14.7	12.7

Table 3: ToolBench single-turn tool-use evaluation results. We highlight the **best** and <u>second best</u> value.

	Gemma-3-1B			Gemma-3-4B			Gemma-3-12B			Gemini-2.5
	Base	ToB	ToG	Base	ToB	ToG	Base	ToB	ToG	flash-lite
Standard	1	/	<u>14.1</u>	11.2	/	<u>17.6</u>	9.8	/	**<u>19.6</u>**	6.9
ReAct	0	3.3	/	0.1	<u>17.6</u>	/	0.5	18.1	/	12.3
DFS	0	3.2	/	0.1	13.4	/	0	12.3	/	15.6

Table 4: Comparison of models trained on ToolGrad and ToolBench datasets on ToolBench. ToB and ToG denote finetuning on ToolBench and ToolGrad datasets, respectively. The best performance per model size is <u>underlined</u>, and the **global best** is bolded.

The results demonstrate that our tool-use capability generalizes effectively to unseen tool usage scenarios. Specifically, the 1B, 4B, and 12B models achieve score increases of +8.1, +8.0, and +6.3, respectively. Table 6 further confirms these performance gains on both the live (+1.93, +4.74, +4.22) and non-live (+14.19, +11.34, +8.37) subscales. While the trend indicates that our fully synthetic dataset most significantly boosts non-live (synthetic) benchmark scores, it notably enhances model performance on real human data benchmark as well. Table 5 shows that our ToolGrad dataset is particularly beneficial to boost "Multiple Parallel" performance, defined as a combined task of 1) tool choices and 2) parallel tool calls. This result well aligns with our results reported in Table 2, where we show the *ToolGrad* dataset is good at generating complex tool uses (with low cost).

Table 6 also shows that ToolGrad-12B ranks the second in BFCL (-0.1 compared to Gemini 2.5 Pro). ToolGrad-12B also outperforms ToolACE (Liu et al., 2025), trained on a more advanced tool set than ToolBench, and Gemini-2.5-flash-lite, its teaching model in the *ToolGrad* framework.

5.3 Scaling Study

Different from other modularized projects that evaluate their full pipeline with an ablation study, modules in our framework work as a whole and cannot be reasonably ablated. Here, we conducted a scaling study that explores how *ToolGrad* performance varies by scaling the number of iterations and the number of samples.

Scaling iterations. We chose iteration = 4, 8, 12, 16 and number of samples = 500 for the study, and reported how 1) pass rate, 2) percentage of unique tool use, and 3) LLM API cost for generating the data using *ToolGrad*. Figure 4 summarizes our results using Gemini 2.5 flash. The figure shows that the pass rate tends to saturate between iteration = 8 and 12, demonstrating our choice of iteration=10 when crafting *ToolGrad-500* as a reasonably cost-effective choice. The "% of Unique Tool Use" curves demonstrate one major challenge in the "scaling law", i.e., the agent tends to generate similar tool uses in our training set. Such repetitive synthetic tool-use data will inevitably harm our model training when we scale our framework for large-scale data generation.

Scaling samples. We further studied the scaling effect on the number of samples. We chose 10 iterations per generation and trained Gemma-3-4B on 100, 500, 1k, 1.5k, and 2k data, using the same finetuning configuration for ToolGrad-4B. We then evaluate the finetuned models on BFCL. Results in Figure 5 show that, while all finetuned models still outperform the baseline, the model performance first increases and then decreases with the scaling of the dataset. These results provide further evidence showing the existence of a saturation point with our current *ToolGrad* system design. We encourage future work to formally investigate this phenomenon and then contribute to breaking this performance saturation point.

5.4 Discussion

Contaminated Data with Unsolvable Queries. Table 3 shows that all models perform poorly in the ToolBench. After investigation of the ToolBench test query, we find that there exist subopti-

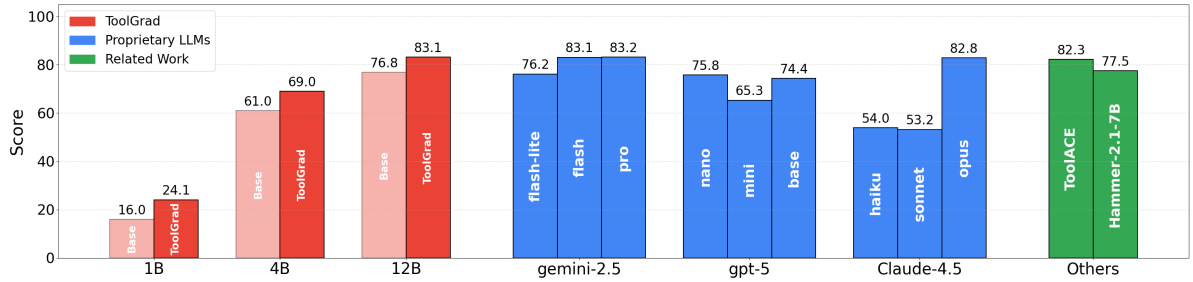

Figure 3: BFCL results. The graph shows single-turn results, and the corresponding breakdown into subscales is available at Table 6 and Table 5. For those supporting both "prompt" and "function call" setups, we chose the "prompt" condition as a fair comparison with Gemma-3, where there is no official function call support.

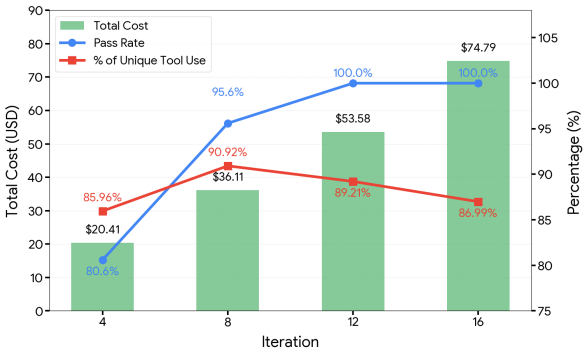

Figure 4: Cost-effective analysis on iterations scaling.

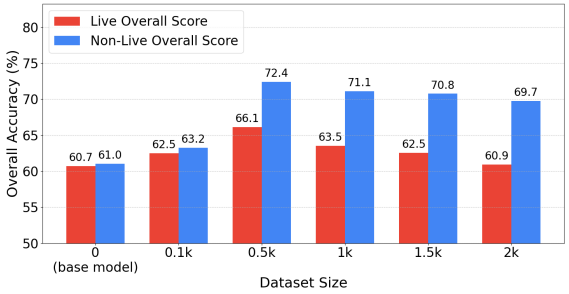

Figure 5: Gemma-3-4B model accuracy on BFCL when scaling ToolGrad samples.

mal queries where the given tool sets cannot support solving the query. This issue is inevitable in the query-first data generation framework, and it is almost impossible to guarantee the resolvability of a generated query. Additionally, even though the query is solved by DFS exploration, the Toolbench training set considers the traces that lead to success as the "ground truth" tool-use traces for LLMs to learn. However, we find such "ground truth" tool-use traces contain low-quality tool-use steps like wrong tool use that lead to data contamination. In comparison, our answer-first generation framework can avoid both issues: 1) all generated data are verified because the generation starts with at least one valid tool use, and 2) our framework does not allow the tool-use failure step to enter our training dataset.

Intelligence Bootstrap with *ToolGrad*. One common approach to eliminate the data contamination in query-first approaches 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, our results on both benchmarks provide strong signals of bootstrapping an LLM intelligence with our unique design of answer-first data generation. For example, Table 3 shows that even the "1B" model can outperform its teacher model, "gemini-2.5-flash-lite". More interestingly, Gemma-3-12B already outperforms "gemini-2.5-flash-lite" in both benchmarks, but "gemini-2.5-flash-lite"-generated data can still be useful to effectively teach a better ToolGrad-12B model.

Agent Memory. In Section 5.3, we reported that the model performance fails to increase with the scaling of our data generation. We believe one major reason is that *ToolGrad* does not contain a memory mechanism. That being said, each data sample in *ToolGrad* is generated independently. This explains the finding under Figure 4, where the framework tends to generate similar tool use in the training set. With a shared memory module, we envision a future version of *ToolGrad* will be able to avoid proposing repetitive tool usage during the data generation, and thus create a higher quality tool use training set.

6 Conclusion

This work introduces *ToolGrad*, a framework for efficient tool-use dataset generation. Our core con-

cept is to first generate tool-use answers with textual "gradients", followed by query generation. We further contribute a high-quality dataset, *ToolGrad-500*, generated with a lower cost and almost 100% pass rate. Experiments show that models trained on *ToolGrad-500* outperform those trained on expensive baseline datasets and proprietary LLMs.

7 Limitations

Multi-step tool use with reasoning. Our models are limited in inference with the ReAct/DFS framework because our fine-tuning dataset lacks reasoning examples. Recent work has introduced an increasing number of tool-use frameworks (Lu et al., 2025), and it is underexplored how our superior performance can be generalized into a broader range of frameworks.

Post-training with reinforcement learning (RL). This work focuses on demonstrating the usage of our dataset with supervised fine-tuning (SFT). Recent exploration highlights the benefit of teaching LLM tool usage with RL (Qian et al., 2025). The value of the datasets we generate for RL remains underexplored.

Aligning human behaviors in generated queries. While the ToolGrad framework enhances the generation efficiency of synthetic tool-use datasets, it does not address another critical issue of synthetic datasets: the alignment of real-world human behaviors. That being said, an LLM-generated user query may not accurately reflect how real-world humans express their intentions when interacting with LLMs. For example, generated queries may lack linguistic diversity. Future work should consider post-processing the generated queries (Wu et al., 2024; Zhang et al., 2024) or take humans in the ToolGrad framework, leading to another analogy to interactive machine learning (IML). With the combined effort of higher efficiency and better human alignment in synthetic approaches, we believe that future agents will be able to rapidly bootstrap their tool-use capability through self-instruction.

Scaling Failure. A major benefit of synthetic data is that it can scale up more easily than real human data collection. However, in Section 5.3, we show the scaling plateau appears at a small number, which limits the motivation of utilizing synthetic data instead of real human data for LLM post-training. We encourage future work to push this scaling limit.

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.

Chi-Min Chan, Weize Chen, Yusheng Su, Jianxuan Yu, Wei Xue, Shanghang Zhang, Jie Fu, and Zhiyuan Liu. 2023. Chateval: Towards better llm-based evaluators through multi-agent debate. *Preprint*, arXiv:2308.07201.

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.

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. In *European Conference on Computer Vision*, pages 297–314. Springer.

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, and 1 others. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474.

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

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong Wang, Yuxian Wang, Wu Ning, Yutai Hou, Bin Wang, Chuhan Wu, Xinzhi Wang, Yong Liu, Yasheng Wang, and 8 others. 2025. Toolace: Winning the points of llm function calling. *Preprint*, arXiv:2409.00920.

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 Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551.

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.

Victor Wang, Michael J. Q. Zhang, and Eunsol Choi. 2025. Improving llm-as-a-judge inference with the judgment distribution. *Preprint*, arXiv:2503.03064.

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*.

Qinzhuo Wu, Wei Liu, Jian Luan, and Bin Wang. 2024. Toolplanner: A tool augmented llm for multi granularity instructions with path planning and feedback. *arXiv 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.

Tianhua Zhang, Kun Li, Hongyin Luo, Xixin Wu, James Glass, and Helen Meng. 2024. Adaptive query rewriting: Aligning rewriters through marginal probability of conversational answers. *Preprint*, arXiv:2406.10991.

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*.

Appendix

A ToolGrad System

A.1 Prompts

The following content shows the prompt templates used in this work, including those in four LLM modules Figure 2 of *ToolGrad* (Prompt 1, Prompt 2, Prompt 3, and Prompt 4) and the template we used in the LLM judge Prompt 5.

```
You are tasked with augmenting an API-use
workflow with more APIs from a given library so
that it can serve for more advanced tasks.
Given the following information that provides
the context, please make three API-use proposals
 to augment the current workflow.

The current workflow:
{workflow_cur}

The following is a pool of APIs that you can use:

{api_all}

CRITICAL - Instruction Format Requirements:
- Each proposal's "instruction" field must be an
 ACTIONABLE COMMAND that tells an agent executor
 HOW to use the proposed API(s).
- Instructions must be IMPERATIVE commands (e.g.,
 "Get...", "Use...", "Retrieve..."), NOT
descriptive summaries.
- DO NOT reference previous executions or
results (e.g., avoid "This tool was used..." or
"It returned...").
- The instruction will be given to an agent
executor that will use it to decide which tools
to call and with what inputs.

Notes:
- Please reply in the required data structure.
- To select an API, you should return its name.
- If you do not have any additional tools to
propose, you can respond with None.
```

Prompt 1: "API Proposer" template.

```
You are tasked with exploring an API based on a
given plan.
The following shows a guide for you to follow:
1. Verify whether the API-calling result
provides a reasonable response for the given
plan.
2. Report `success = False` if you fail to get a
 reasonable result, and explain why.
3. Report `success = True` if you get a
reasonable result that addresses the plan, and
provide justification for the success.
4. If you report `success = True`, you should
also report which function calling step leads to
 the success.

Retry Logic:
- The API may return bad results (e.g., error
messages, completely irrelevant data, or
responses that don't address the plan).
- In such cases, you should try again with
different input parameters if reasonable
alternatives exist.
- You have a maximum number of iterations to
complete the task. Use them wisely.
- If retries fail or no reasonable alternative
inputs exist, report `success = False`.

The following is the plan:
{plan}

Notes:
- Success criterion: If the API execution
provides a reasonable result that addresses the
given plan, report `success = True`.
- If an API fails to execute or returns unusable
 results after reasonable attempts, report `
success = False`.
```

Prompt 2: "API Executor" template.

```
You are an API selector.

You need to select one API or refuse to select
any API from the given list of APIs to augment
the current workflow.

The current API-use workflow:
{workflow_cur}

Reports from the proposed APIs:
{api_reports}

When you select an API, you need to make the
following decisions:
1. Determine whether any API can be used to
augment the current workflow.
2. If yes, select one API to augment the current
 workflow.
3. Decide whether to append the selected API to
an existing API-use chain or create a new API-
use chain:
    3.1 **Append to existing chain**: Choose this
 when the API logically should follow another
call in an existing chain. This includes cases
where:
        - The `tool_input` contains data from a
previous API's `response`
        - The API's purpose depends on results
from a previous call
        - Even if `tool_input` is minimal, the
API conceptually continues work from a previous
step
        When you decide to append, you should
also select which API-use chain to append to.
    3.2 **Create new chain**: Choose this when
the API is independent and does not logically
depend on any existing API execution. For
example:
        - The API addresses a completely separate
 aspect of the workflow
        - The `tool_input` is self-contained and
doesn't rely on previous results
```

Prompt 3: "API Selector" template.

```
You are generating training data for a tool-use
language model. Given API execution traces,
```

```
create a natural user query that would trigger
these API calls, followed by an appropriate
response.

**API Execution Details:**
{api_use_chains}

**Task:** Generate (1) a natural user query and
(2) the agent's response based on the API
execution traces above.

**Important:** You will receive the API
execution chains for context, but you should NOT
 return them in your output. Only return the
query and response fields.

---

**CRITICAL: User Query Requirements**

1. ** DO**: Write queries like a real human
would ask
   - "What's the weather forecast for London
next week?"
   - "I'm researching Tesla stock. Show me
recent performance and news."
   - "Find me Italian restaurants near Golden
Gate Park with good ratings."

2. ** NEVER**: Mention APIs, tools, functions,
or technical implementation
   - Never say: "call the weather API", "use
get_forecast", "invoke the search tool"
   - Never ask: "which API should I use", "can
you run this function"
   - Never include: tool names, API endpoints,
function signatures

3. ** DO**: Provide specific, concrete details
   - Include exact values from tool_input (
locations, IDs, names, numbers)
   - Use specific examples: "123 Main St,
Oakland CA" not "an address"
   - Mention precise entities: "Tesla stock" not
 "a company's stock"

4. ** DO**: Create realistic scenarios
   - Explain WHY the user needs this information:

     * "I'm planning a trip..."
     * "I'm writing a research report on..."
     * "I need to prepare for a meeting about..."
   - Make the request feel natural and
purposeful

5. ** DO**: Cover ALL API calls implicitly
   - If 3 APIs were called, the query should
naturally require all 3
   - Don't list them ("do A, B, and C"), weave
them into a cohesive need
   - Example: Instead of "Get weather, find
hotels, search restaurants"
     --> "I'm visiting Paris this weekend. What
should I expect, and where should I stay and eat
?"

---

**Response Requirements:**
- Synthesize all API execution results into a
helpful, natural response
- Present information clearly without mentioning
 APIs or tools
- Reference concrete data from the execution
outputs
- Sound like a knowledgeable assistant answering
 a user's question

---

**Examples:**

**Example 1:**
API Chains: [weather(city="Tokyo"),
currency_convert(from="JPY", to="USD", amount
=5000)]
Query: "I'm traveling to Tokyo next month. What'
s the current weather like, and how much is
5,000 yen in US dollars?"
Response: "Tokyo is currently experiencing mild
temperatures around 18C with partly cloudy skies.
 As for the currency conversion, 5,000 Japanese
yen is approximately 33 US dollars."

**Example 2:**
API Chains: [github_search(topic="ML"),
github_get_repo(id="tensorflow/tensorflow"),
github_get_contributors(id="tensorflow/
tensorflow")]
Query: "I'm researching popular machine learning
 projects on GitHub. Can you tell me about
TensorFlow -- how active is the project and who
are the main contributors?"
Response: "TensorFlow is one of the most popular
 machine learning frameworks on GitHub with over
 180,000 stars. The project is very active with
regular updates. The main contributors include
members of the Google Brain team, with key
developers like Jeff Dean and Rajat Monga being
significant contributors."

---

Make the query sound like something a real
person would ask in a conversation or search bar.
```

Prompt 4: "Inverse Prediction" template.

```
Task Overview:
You are an expert evaluator for a Tool-Use agent.
 Your task is to determine if the provided "Tool
 use trace" successfully retrieves the
information needed to satisfy the "User query".

CRITICAL RULES:
1. STRICT EVALUATION: Evaluate ONLY the
information contained in the "Tool use trace".
Do NOT assume the agent can answer using
external knowledge.
2. ZERO SCORE: If the "Tool use trace" is empty,
 or if ALL tool calls in the trace returned an
error (e.g., timeout, invalid parameters,
authentication failure), the score MUST be 0.
3. GROUNDING: We are judging the SUFFICIENCY of
the retrieved data to answer the query, not a
natural language response.

Evaluation Criteria:
```

```
1. Atomic Request Decomposition:
   - Breakdown the user query into distinct,
atomic information needs (e.g., "Request 1: Find
 movie X", "Request 2: Get cast for movie X").

2. Individual Request Scoring:
   - For EACH atomic request, assign a score
based on the sufficiency of retrieved data:
     - 0: No information retrieved. No tools
called, or the trace is empty, or ALL relevant
tool calls failed (e.g., timeouts,
authentication errors).
     - 25: Unsuccessful attempt. Tools were
called with intent but returned errors (e.g., "
invalid parameter", "missing argument"), and NO
useful data was retrieved. This acknowledges the
 effort and the error message provides
information for a next step improvement.
     - 50: Partial success. Some relevant data
was retrieved, but it is incomplete or lacks
critical details (e.g., found the movie ID but
failed to get its streaming links).
     - 75: Near complete success. The bulk of the
 requested information is present. Minor details
 might be missing, or the retrieval was slightly
 indirect/cluttered but sufficient.
     - 100: Complete success. All requested
information was retrieved accurately via valid,
grounded tool calls with no errors.
   - You can choose scores between the above
values to better fit the real scenario.

3. Final Score Calculation:
   - Compute the final score by averaging the
scores of all identified atomic requests.

Example:
Query: "Find 'Documentary' on Vimeo AND get
streaming link for YT ID '123'" (2 requests)
- Trace: Vimeo search succeeded. YT retrieval
failed. -> Score: (100 + 0) / 2 = 50.
- Trace: Both tools failed with timeouts. ->
Score: (0 + 0) / 2 = 0.
- Trace: No tools called. -> Score: 0.

Input Data:
User query:
{query}

Tool use trace (a list of objects with tool_name,
 tool_description, tool_input, and response):
{tool_use_trace}
```

Prompt 5: LLM judge template.

A.2 Tool-use Error Handling

Real-world tools may lead to execution failure, such as network timeouts and invalid parameters. In the "API Executor" module, we configure the timeout to 10 seconds. When an "API Executor" fails, it will reflect in the corresponding reports (see "API Execution Report" in Figure 2). The followup "API Selector" will not consider those failure report and only perform selection on those APIs that lead to successful API calls.

Our system cannot self-instruct while generating the data. That being said, "API Executor" cannot learn from the tool-use experiences of other "API Executor" within or even outside a generation session. To further enhance our system, we encourage future work to incorporate a memory system in our current implementation of *ToolGrad*.

A.3 API Library

We used the API library provided by ToolBench (Qin et al., 2023). 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 *gemini-2.5-flash-lite* to perform two rounds of filtering. In the first round, we filter APIs with low-quality annotations. In the second round, we create an agent to execute the tools, and aim to receive a non-failure response at least once within 10 LLM call budgets. See Prompt 6 for our instruction for the data filtering agent. This results in 15,368 qualified APIs.

```
You are an expert at testing APIs. Your goal is
to successfully call the given API at least once.

You will be provided with the API documentation.
You should generate valid JSON inputs for the
API.
If the API call fails, analyze the error message
 and try a different valid input.
You have a maximum of 10 attempts.

IMPORTANT:
1. If you believe it is impossible to get a
successful response (e.g., API is permanently
broken, requires unreachable dependencies), you
MUST set "action" to "stop".
2. Do NOT simply repeat the same input if it
failed. You MUST try different parameters or
values.
3. If you decide to "call" the API, you MUST
provide "tool_input".

Output your response in the following JSON
format:
{
  "thought": "Your reasoning for the current
attempt.",
  "action": "call" or "stop",
  "tool_input": { ... key-value pairs for the
API arguments, required if action is 'call' ... }

}
```

Prompt 6: Instruction for the ToolBench-API-filtering agent.

B Model Training

B.1 Configurations

The training code is based on "SFTTrainer" in the "trl" package. We trained ToolGrad-1B based on "gemma-3-1b-it" using learning rate of 1e-05. We trained ToolGrad-4B and ToolGrad-12B based on "gemma-3-4b-it" and "gemma-3-12b-it" using LoRA adapters with a rank of 64, an alpha of 16, and a dropout rate of 0.1. The learning rate is 5e-06 and 2e-05, respectively. All ToolGrad models are trained with batch size of 1, using the adamw optimizer for three epochs. For ToolBench models, we followed the official configuration reported in the paper. All models are trained with 8k context windows size, and we computed the training loss exclusively on the assistant's completions by masking the user instruction tokens, i.e., "assistant_only_loss=True" in "SFTConfig".

B.2 Training Budgets

ToolGrad-1B, 4B, 12B was trained on four A100 GPUs and costs 1, 1.67, and 2.67 GPU hours in total, respectively. ToolBench-1B, 4B, 12B was trained on eight H100 GPUs and cost 29, 68, and 370 GPU hours in total, respectively.

C Dataset Formatting

Prompt 7 shows how we setup the dataset for our SFT tasks. In additional to the 1-on-1 formatting, we also add 20% more negative samples where we replace all the positive tools in the selection pool with unrelated tools, and the corresponding output is "[]", aiming to reduce model hallucination.

We recommend the readers to review our attached dataset (under "train.jsonl")for examples of our generated data.

```
You are an expert in composing functions. You
are given a question and a set of possible
functions. Based on the question, you will need
to make one or more function/tool calls to
achieve the purpose.
If none of the functions can be used, point it
out. If the given question lacks the parameters
required by the function, also point it out.
You should only return the function calls in
your response.

If you decide to invoke any of the function(s),
you MUST put it in the format of [func_name1(
params_name1=params_value1, params_name2=
params_value2...), func_name2(params)]
You SHOULD NOT include any other text in the
response.

The following is a list of apis in the library
that you can use.
Each api is a JSON object with 'name', '
description' and 'parameters'.

```json
{selection_pool}
```
```

Prompt 7: Instruction template for SFT.

D Evaluation

D.1 LLM Judges

We utilized multiple LLMs as judges to evaluate the prediction, including two SoTA proprietary models (gemini-2.5-pro, claude-4.5-sonnet) and two SoTA opensourced models (deepseek-v3.2, qwen3-235b). Prompt 5 shows the prompt template we used for our judges. This approach is inspired by (Qin et al., 2023)'s evaluation design. We improve their approach by only showing the LLM judges the query and the tool use traces without a generated response. Our design can effectively avoid biases introduced in the response writer LLM, where the write may answer a given user query even though there is no tool call.

D.2 Supplementary Discussion on ToolBench Results

Table 3 did not contain the results of ToolBench models under the "standard" condition. The ToolBench models are trained to output texts in the format of "Thought: ... Action: ... Action Input: ...", and call one single tool in each round. The "standard" condition allows only one round of LLM execution and expects the model to predict all tool use. Therefore, the ToolBench model cannot be fairly evaluated in this single-turn tool-use condition, as it will make at most one tool call. The format limitation also explains why all Gemma-3 models perform worse in the ReAct/DFS condition – base Gemma models cannot strictly follow this customized template, so the ToolBench evaluation parser fails to parse tool calls.

This explains 1) why ToolBench models cannot be fairly evaluated in our single-turn tool use condition ("standard"), and 2) why all Gemma-3 performs worse in ReAct/DFS condition – base Gemma models cannot strictly follow this customized template, and thus the ToolBench evaluation parser fails to parse tool calls.

D.3 Human Evaluation Design

We recruited two volunteers to annotate the results. Both participants work full-time as programmers in the AI industry. We obtained participants' consent before the study. This study is determined exempt by an ethics review board.

Here we provide details of our human evaluation study design mentioned in our response to Weakness 2. For each question, one author manually breaks it down into a list of subqueries. The following query is an example we selected in the evaluation:

> *"I want to explore different genres of movies. Fetch the genre names and IDs for me. Also, provide me with the basic information about a specific cast member, including their name, profession, birth and death years, and best titles."*

For this query, we break it down into two subqueries:

1. *"Fetch the genre names and IDs for me."*,
2. *"Also, provide me with the basic information about a specific cast member, including their name, profession, birth and death years, and best titles."*

For each subquery, the rater needs to choose one score from 0, 25, 50, 75, 100. The definition is the same as the following quote in Prompt 5:

0: *No information retrieved*. No tools called, or the trace is empty, or ALL relevant tool calls failed (e.g., timeouts, authentication errors).

25: *Unsuccessful attempt*. Tools were called with intent but returned errors (e.g., "invalid parameter", "missing argument"), and NO useful data was retrieved. This acknowledges the effort, and the error message provides information for the next step improvement.

50: *Partial success*. Some relevant data was retrieved, but it is incomplete or lacks critical details (e.g., found the movie ID but failed to get its streaming links).

75: *Near complete success*. The bulk of the requested information is present. Minor details might be missing, or the retrieval was slightly indirect/cluttered but sufficient.

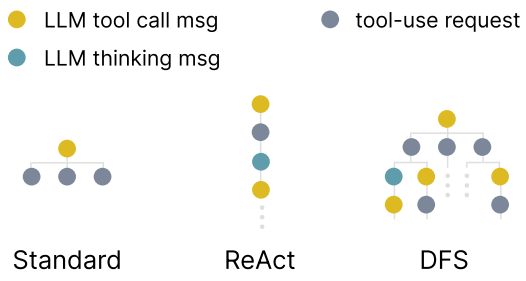

Figure 6: A visualized comparison among standard, ReAct, DFS inference frameworks.

100: *Complete success*. All requested information was retrieved accurately via valid, grounded tool calls with no errors.

The final human evaluation score for a given answer to the full query is the average across all subqueries. This definition aligns with those in Prompt 5 to ensure we follow the same instructions in both human and LLM studies.

D.4 BFCL Subscale Results

BFCL evaluates tool-use capability in different scenarios. "Simple" involves a single invocation of a provided tool. "Multiple" tests the model's ability to select the correct tool from a candidate set. "Parallel" evaluates the generation of multiple simultaneous calls for a single tool. "Multiple Parallel" assesses the combined ability to select from multiple tools and generate multiple parallel calls within a single turn Most of our formatted data are "Multiple Parallel" data.

E 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 6 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 (**?**)) 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.

Model		Non-live				Live			
		Simple	Multi	Par	MultiPar	Simple	Multi	Par	MultiPar
Gemini 2.5	Pro	78.67	**94.00**	93.50	92.00	85.66	74.36	87.50	**83.33**
	Flash	77.33	91.50	**96.00**	87.50	**87.21**	75.97	81.25	75.00
	Lite	70.08	86.00	90.00	89.50	67.05	51.66	75.00	50.00
GPT-5	Base	72.50	86.50	82.00	77.50	77.52	67.43	68.75	58.33
	Mini	59.17	72.50	71.50	69.00	69.77	61.16	75.00	37.50
	Nano	69.25	86.00	87.50	80.50	76.36	69.71	68.75	54.17
Claude 4.5	Opus	**79.58**	93.50	93.00	**92.50**	84.50	74.17	81.25	62.50
	Sonnet	47.25	79.50	53.50	59.00	73.26	40.17	56.25	33.33
	Haiku	55.67	84.00	38.00	44.00	66.67	49.76	56.25	16.67
Hammer-2.1-7B		72.50	92.50	91.00	86.00	66.67	69.99	75.00	75.00
ToolACE-2-8B		73.42	91.00	93.00	91.00	71.32	**79.39**	68.75	62.50
Gemma 3	12B	76.25	**94.00**	91.00	56.50	85.66	71.89	**87.50**	45.83
	4B	64.50	88.00	56.00	36.00	70.93	59.35	25.00	41.67
	1B	43.33	36.00	0.00	1.50	36.43	6.27	0.00	0.00
ToolGrad	12B	75.25↓	**94.00**↑	93.50↑	88.50↑	85.66↑	77.11↑	75.00↓	62.50↑
	4B	65.33↑	86.50↓	73.00↑	65.00↑	71.32↑	64.86↑	43.75↑	50.00↑
	1B	49.08↑	33.50↓	34.00↑	21.00↑	30.62↓	9.97↑	6.25↑	4.17↑

Table 5: BFCL score breakdowns under Non-live and Live tracks. The test is divided in Simple (single call), Multiple (tool selection + call), Parallel (parallel calls), and Multiple Parallel (tool selection + parallel calls). We highlighted **the highest** and the second highest scores in each breakdown scale. On the ToolGrad rows, we annotated whether the score has increased (↑) or decrease (↓) compared to its corresponding Gemma 3 models. ToolGrad models show increase (↑) in 19 out of 24 subcales.

F 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 the Apache License 2.0.

G AI Assistant Usage

We utilized an AI assistant to support grammatical check and code writing in this project.

Model		Non-live	Live	Halluc.	
		Overall	Overall	Rel.	Irrel.
Gemini 2.5	Pro	89.54	76.83	62.50	86.97
	Flash	88.08	78.16	91.09	62.50
	Lite	83.90	54.85	93.33	50.00
GPT-5	Base	79.62	69.21	73.42	87.10
	Mini	68.04	62.55	55.71	93.75
	Nano	80.81	70.69	45.75	93.75
Claude 4.5	Opus	**89.65**	76.02	90.75	68.75
	Sonnet	59.81	46.56	95.03	37.50
	Haiku	55.42	52.48	95.29	31.25
Hammer-2.1-7B		85.50	69.50	50.00	90.12
ToolACE-2-8B		87.10	77.42	75.00	90.79
Gemma 3	12B	79.44	74.24	70.29	93.75
	4B	61.12	60.84	53.94	100.00
	1B	20.21	11.84	33.18	37.50
ToolGrad	12B	87.81↑	**78.46**↑	93.75	59.27
	4B	72.46↑	65.58↑	93.75	59.27
	1B	34.40↑	13.77↑	81.25	26.89

Table 6: BFCL subscale scores. We highlight **highest** and second highest scores. ↑ indicates increase over baseline Gemma 3 models. Halluc. (Hallucination) is split into Rel. (Relevance) and Irrel. (Irrelevance).