Tool learning via Inference-time Scaling and Cycle Verifier

Xiaobo Liang[♣,♠], **Wenjing Xie**[♣,♠], **Juntao Li**[♣,♠*], **Wanfu Wang**[♣,♠],
Yibin Chen[♦], **Kehai Chen**[♥], **Min Zhang**[♣,♠]
[♣] Soochow University, [♦] Huawei Technologies, [♥] Harbin Institute of Technology
[♠] Key Laboratory of Data Intelligence and Advanced Computing, Soochow University
{xbliang, ljt, minzhang}@suda.edu.cn chenyibin4@huawei.com
{wjxie, wfwang}@stu.suda.edu.cn chenkehai@hit.edu.cn

Abstract

In inference-time scaling, Chain-of-Thought (CoT) plays a crucial role in enabling large language models (LLMs) to exhibit reasoning capabilities. However, in many scenarios, high-quality CoT data is scarce or even unavailable. In such cases, STaR-like methods can help LLMs synthesize CoT based on user queries and response, but they inevitably suffer from the risk of compounding errors. In this work, we tackle an even more challenging scenario: tool learning in the absence of user queries. We design a data scaling method using back-translation, which establishes an inference cycle to synthesize both user queries and CoT data. To reudce the compounding error of inference time, we introduce two rule-based verifiers to assess the validity of the synthesized CoT data. In particular, the Cycle Verifier facilitates performance improvement by continuously accumulating new data over multiple iterations. Our approach achieves a **75.4%** pass rate and a **79.6%** win rate using small models (7B) in StableToolBench. Notably, these results are obtained exclusively from self-synthesized high-quality data, without relying on external supervision or expert trajectories for warm-up.

1 Introduction

Tool use is a critical capability for LLMs, enabling them to perform complex reasoning tasks through interacting with real-world environments (Mallen et al., 2022; Wang et al., 2023b; Zeng et al., 2023; Xu et al., 2023b; Huang et al., 2024). *Imitation learning* is one of the most efficient ways to acquire new skills, as it allows LLMs to benefit from expert trajectories. However, collecting high-quality expert data (Yang et al., 2024; Qin et al., 2023a) is often time-consuming and inherently non-scalable.

In real-world scenarios, human experts use tools in a sequential manner: they first determine the query intent, select the appropriate tool, and then provide a CoT-based solution. In contrast, instruction backtranslation (Li et al., 2023) follows an inverse process, inferring the user's intent based on a given solution. As shown in Figure 1, we integrate these two approaches to establish an **inference cycle** by leveraging tool environments and LLMs. This cycle ensures data scalability by continuously generating new user queries and interacting with the environment to produce CoT data. STaR-like methods (Zelikman et al., 2022) provide a potential approach to bootstrapping reasoning capability in inference-time scaling (Snell et al., 2024). However, they rely on oracle responses[1] to filter low-quality data using rejection sampling, which is unavailable in our scenario.

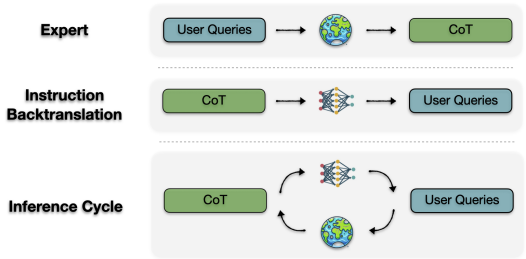

Figure 1: Illustration of Inference Cycle.

In multi-intent tool use, LLMs often require multiple rounds of reasoning and interaction. It is not suitable to obtain reward estimation through frequent trial-and-error, which is inefficient and unscalable. In other words, without a deterministic reward modeling, data augmentation is highly susceptible to compounding errors (Cundy and Ermon, 2023), a phenomenon known as LLM hallucination and error propagation in inference trajectories. Thus, we introduce two predefined rule-based verifiers, which provide process-level reward signals to evaluate each API invocation action. In

[*] Corresponding Author

[1] In mathematical reasoning, the oracle response corresponds to the final numerical solution, while in tool use, it refers to the invoked function name and specific parameters.

Findings of the Association for Computational Linguistics: ACL 2025, pages 24658–24671

particular, the format verifier aims to filter out parameter value inconsistencies by ensuring that the provided parameters align with the function's predefined types. The cycle verifier ensures the consistency of API invocation trajectories, preventing deviations from the conditions of the initially synthesized user query. Notably, these reward signal can be distributed across multiple reasoning steps, ensuring the effectiveness of process supervision.

Furthermore, we introduce the preference tree (Zhuang et al., 2023) to enhance sampling efficiency, which reduces the cost of redundant reasoning in multi-step inference tasks. Additionally, the preference tree provides pairwise samples (chosen and rejected) for Direct Preference Optimization (DPO) training. In our implementation, we adopt iterative training to perform multiple stage data synthesis, a process we call InfCycle, effectively unlocking the potential of inference-time computation. We evaluate InfCycle on a diverse set of benchmarks, including the interactive `StableToolBench` and the non-interactive `Berkeley Function Calling` dataset. Moreover, our evaluations across multiple foundation models, from the weakly toolcapable Mistral-7B-Instruct to the more advanced LLaMA3-8B-Instruct and Qwen2.5-7B-Instruct. Empirical results indicate that InfCycle not only enhances model performance but also outperforms robust baselines such as GPT-4 on `StableToolBench`. Additionally, on `Berkeley Function Calling`, InfCycle achieves relative improvements of $+\mathbf{16}$ and $+\mathbf{40}$ points, further demonstrating its effectiveness. Our main contributions are as follows:

- We show that InfCycle is an effective strategy for enabling LLMs to master candidate tools even when expert data is unavailable.
- In the absence of explicit reward signals, our findings indicate that combining LLMs with rule-based rewards effectively reduces compounding errors in data synthesis, thereby enhancing overall performance.

2 Related Works

Tool Learning As pioneers, Toolformer (Schick et al., 2024), Gorilla (Patil et al., 2023), and ToolAlpaca (Tang et al., 2023) have explored the potential of LLMs in tool use. ToolLLaMA (Qin et al., 2023b) notably expanded the number of available tools, exceeding 10,000 APIs, and investigated the possibilities of data scaling. Many related works primarily seek improvements through two approaches: **Inherent Abilities**: This involves manipulating prompts or enhancing the execution framework. Xu et al. (2023b) utilize examples, in-context demonstrations, and generation styles to explore the potential of LLMs. AutoAct (Qiao et al., 2024) employed a multi-agent collaboration framework to complete reasoning tasks. RestGPT (Song et al., 2023) introduced a coarse-to-fine online planning mechanism by using three main modules (Planner, API Selector, and Executor). **Synthetic Data**: This strategy empowers model capabilities through synthetic data. ToolVerifier (Mekala et al., 2024) leveraged the LLaMA-2 70B model to verify the accuracy of synthetic data. APIGen (Liu et al., 2024b) used a strong model to filter API calls based on rules and semantics, ensuring data accuracy.

Inference-time Scaling LLMs can utilize techniques such as CoT (Wei et al., 2022) or Reflection (Shinn et al., 2024) to enhance their reasoning capabilities during testing. However, many studies show that these methods often have limited effectiveness for complex tasks (Huang et al., 2023; Stechly et al., 2023; Valmeekam et al., 2023). Nevertheless, this research direction remains crucial for the future, particularly in exploring the trade-offs between inference time and pre-training computing. Brown et al. (2024) demonstrate that scaling inference computing through repeated sampling leads to significant improvements in coverage across various tasks and models. Snell et al. (2024) introduce a compute-optimal strategy that enhances the efficiency of test-time compute scaling compared to a best-of-N methods.

In our scenario, no expert data trajectories or proprietary models are available, meaning the model improves purely through self-improvement. Under such strict conditions, obtaining explicit reward signals is highly challenging. Therefore, this work focuses on exploring the effectiveness of prior knowledge and consistency constraints as alternative forms of reward.

3 Preliminaries

We first provide the task definition of tool use and describe the scenarios in our works. Given a user query $\mathcal{Q}$ and a set of candidate API functions $\mathcal{A} = \{\texttt{API}_0, \texttt{API}_1, \ldots, \texttt{API}_{|\mathcal{A}|}\}$, LLMs require to fulfill the user's intent by executing a specific sequence of API function calls. The decision pro-

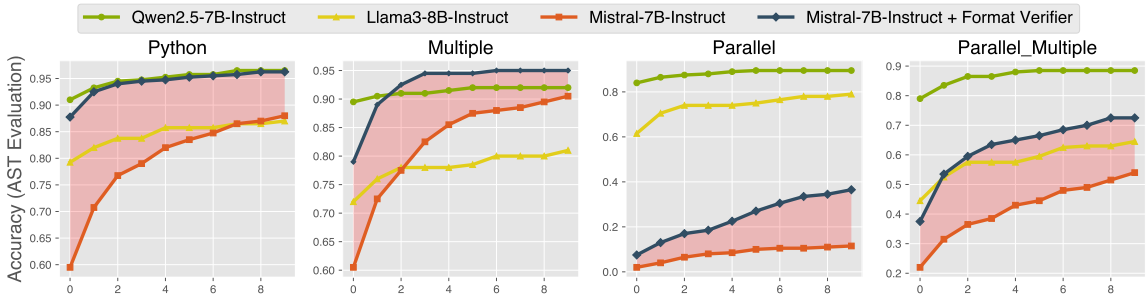

Figure 2: Pass@N results of model performance across different reasoning steps (from 1 to 10). The non-live dataset of the `Berkeley Function Calling` benchmark comprises six distinct subsets, with our analysis focusing exclusively on the test set using Python. In this context, *Multiple* refers to tasks that require the use of multiple distinct functions to achieve the desired outcome, whereas *Parallel* denotes cases in which the same function must be invoked multiple times to fulfill the user's intent.

cess can be described as $y \sim \pi_\theta(y|s_0, a_1, a_2, \cdots)$, where $\pi_\theta(\cdot)$ represents the policy, s_0 denotes the initial task state [2], and a represents the actions, such as selecting or executing a specific API function from $\mathcal{A}$. During inference-time scaling, we sample data and optimize the model using reinforcement learning algorithms. Beyond this, it is essential to understand the capabilities of different foundation models to achieve a better trade-off between inference time and performance optimization.

3.1 Trial Experiment Design

We chose to use the `Berkeley Function Calling` Benchmark to evaluate three different foundation models: Mistral-7B-Instruct, LLaMA3-8B-Instruct, and Qwen2.5-7B-Instruct. The `Berkeley Function-Calling` benchmark contains four non-live subsets to evaluate different intents. During the evaluation process, we follow the official scripts[3] and prompts for all models except Mistral-7B-Instruct[4]. The experimental results are presented in Figure 2, using *Pass@N* as the evaluation metric. *Pass@N* is a key metric for evaluating a model's reasoning capability. It represents the probability that the model generates at least one correct answer within *N* attempts. This metric reflects both the inherent model capability and its potential to leverage effective verifier to identify the correct answer.

Observation 1. The accuracy and model capability exhibit a positive correlation, meaning that as the number of inference steps increases, all models consistently improve their chances of obtaining the correct answer. For relatively simple tasks (such as *Python* and *Multiple*), Mistral-7B-Instruct benefits from a multi-step reasoning approach, which effectively reduces task complexity and allows it to outperform LLaMA3-8B-Instruct. However, for more challenging tasks (such as *Parallel* and *Parallel Multiple*), the model's inherent capability remains the key determinant of performance.

Furthermore, we introduced a simple verifier, termed the Format Verifier, to filter the data. A key observation is that, when selecting parameters from the candidate toolset, the model does not consistently adhere to the predefined parameter types. Consequently, when the model's output deviates from the expected function names, parameter names, or parameter types, we prompt it to retry. This approach effectively employs a rule-based reward mechanism to guide the model's behavior and enhance its output selection.

Observation 2. As observed, Mistral-7B-Instruct outperforms the Qwen2.5-7B-Instruct in both the *Python* and *Multiple* tasks. Even on more challenging tasks, it demonstrates strong performance.

Conclusion. For models with limited tool use capabilities, such as Mistral-7B-Instruct, compounding errors are inevitable during the generation. In simpler tasks, the prompt refinements or basic filtering mechanisms often ensures the generation of correct answers. However, for more complex tasks that inherently require multi-step reasoning, it is crucial to ensure the accuracy of each intermediate

[2]The inital state is the prompt, which contains user query, API candidates, and task instruction.

[3]https://github.com/ShishirPatil/gorilla/tree/main/berkeley-function-call-leaderboard

[4]Since Mistral-7B-Instruct has a weaker ability to follow instructions, it struggles to produce outputs that can be directly evaluated. To address this, we introduce a multi-step reasoning framework, which is detailed in the Methods section.

step to prevent errors from accumulating over multiple steps. Therefore, it may require enhancement through multiple iterations.

4 Methods

In scenarios lacking tool-specific training data, we first collect an available candidate toolset and synthesize data. Then, we filter high-quality CoT data for model training and inference, As shown in Figure 3. Considering the capability limitations of open-source small size LLMs, the entire process requires multiple iterative refinements.

4.1 Data Synthesizer

To handle diverse real-world tool-use scenarios, we collect a large set of interactable APIs for data collcetion. To support this, we create and deploy an execution environment for these APIs, enabling type checking and providing execution feedback. For more details, please refer to Section 5.

Collect API and Parameters. The interactive environment inherently ensures the successful execution of API requests while filtering out invalid ones. Available APIs are collected as tool candidates, with input consistency maintained through a unified format. Given the complexity of user intent, we incorporate API types that group functionally similar tools to construct *Parallel* and *Multiple* tasks. The sampled tool data, including API definitions, request parameters, and execution results, are utilized to build the candidate tool list.

User Query Synthesis We start with a sampled API list and use LLMs to generate potential user queries $\mathcal{Q} = \{q_1, q_2, \cdots, q_{|\mathcal{Q}|}\}$. However, not all queries are equally useful. To filter out irrelevant ones, we let LLMs act as semantic checkers (Liu et al., 2024b), ensuring that each query is plausible in real-world usage and that the expected response correctly aligns with user intent.

CoT Trajectories Synthesis Once we have a valid query set, we generate structured execution trajectories. Formally, we define a trajectory as $\{q, a_p, a_s, a_e\}$, where a_p represents hierarchical task planning, a_s represents API selection with its parameters, and the final execution outcome a_e. By explicitly modeling these transitions, we facilitate structured reasoning within LLMs, progressively improving their tool-use capabilities.

To fully capture the reasoning dynamics behind tool invocation, we break down the inference process into four steps: (1) Task Planning: The model first deconstructs the user query q into structured sub-tasks, a process analogous to hierarchical reasoning methods like ReWoo (Xu et al., 2023a). (2) Tool Selection: Leveraging its capability, the model selects the most suitable tools and parameter configurations to address the sub-tasks. (3) Tool Execution: The chosen tools interact with the environment to obtain execution results, which serve as inputs for subsequent reasoning steps. (4) Task Summarization: The model aggregates the execution results, synthesizing a well-formed summary that aligns with the user's intent. This reasoning pipeline allows LLMs to construct an explicit action trajectory, orchestrating the tool-use process in a manner that mimics human-like sequential decision-making.

Remark: *Task planning decomposes complex tasks into multiple simpler sub-tasks, leading to iterative cycles of Tool Selection and Tool Execution in the reasoning process.*

4.2 Inference Scaling via Cycle Verifier

The above data synthesis pipeline defines a symmetric process, where the flow progresses from the API candidate list to user query generation, and further into CoT-based reasoning trajectories via tool execution. However, CoT trajectory inference is particularly prone to compounding errors. To address this, we introduce two verifiers to filter out low-quality CoT data.

Format Verifier. Each API has predefined parameter names and types. During inference, LLMs often generate invalid parameters when attempting to fulfill the user query. The format verifier assist the LLMs in learning the parameter constraints specific to each API. Furthermore, our experiments enforce JSON-formatted outputs, filtering out responses with invalid formats, ensuring consistent formatting that benefits model training.

Cycle Verifier. In multi-step reasoning, LLMs often need to predict multiple tool invocations to fulfill a user query. Ensuring consistency between the invoked tools and the user query conditions is critical. This verifier evaluates coherence across different reasoning contexts, minimizing errors caused by hallucinations or noise.

Why is the Cycle Verifier Necessary? As illustrated in Figure 4, these issues are unpredictable when synthesized data is validated only through a

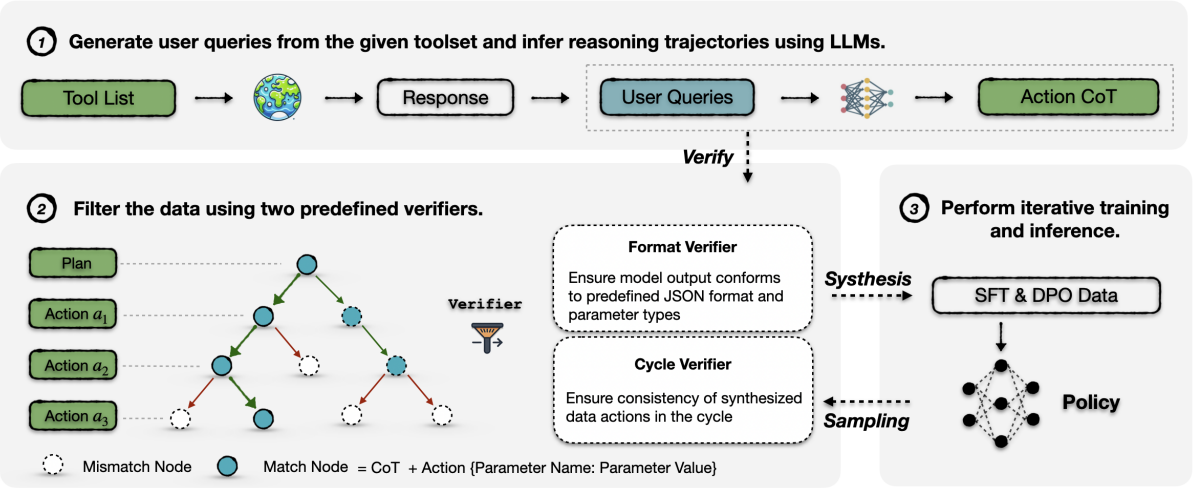

Figure 3: The entire InfCycle process consists of three key components: First, we construct user queries suitable for inference. Since to the lack of correct labels, two verifiers are employed to filter the data. For small size model, multiple iterative steps are also necessary to progressively enhance their capabilities.

Case Analysis

Internal Logical Error: *#Query*: I have an equation that describes a signal over time, given by $x(t) = \text{Re}\left(Ae^{j\pi Bt}\right) + \text{Re}\left(De^{j\pi Et}\right)$, where A and D are constants. If A=3, D=0.5, E=5, and B=4, what are the values of B and E in this equation?
Planning Error: *#Sub-Plan*: Since find_triplet_equal_sum is supposed to solve the user's query directly, skip calling the twosum function for now.

Figure 4: The above case illustrates two types of errors found in the filtered samples.

semantic verifier: (1) Internal logical issues within the user query: These issues can lead to synthesized queries that lack executable trajectories. (2) Semantic problems in planned sub-tasks: Such problems can result in synthesized trajectories that are fundamentally unreasonable. Unlike recent approaches (Qin et al., 2023b; Liu et al., 2024b) that use an **outcome verifier** to filter samples, step-wise cycle consistency as a **process verifier** ensures the accuracy of the reasoning process, thereby guaranteeing the high quality of the synthesized data.

4.3 Iterative Training via Preference Tree

To further enhance the decision-making capability of LLMs, we aim to iteratively optimize them through data synthesis and inference. The preference tree has already been widely used as a general method for optimizing model strategies. To generate preference data, we define the entire reasoning process starting from the initial node a_p as an expansion into a decision tree. The successors of each node are generated through temperature sampling.

In conclusion, the entire process is divided into three stages: *Stage 1*: An instruction LLM interacts with the environment to generate CoT trajectories. *Stage 2*: Using the synthesized data from Stage 1, we train an initial model and employ a tree-search method to generate new trajectories, filtering out unreasonable samples through cycle consistency. *Stage 3*: From the synthesized trajectories within the tree, we compare sibling nodes to identify correct and incorrect pairs to build preference data. Direct Preference Optimization (DPO) is then applied to refine the model's ability to distinguish between similar yet distinguishable trajectories. This structured, iterative approach facilitates a gradual improvement in decision-making, ensuring that the model can better navigate complex reasoning tasks by learning from its mistakes and optimizing its decision boundaries.

Method	Model	Inf	I1-Ins	I1-Cat	I1-Tool	I2-Cat	I2-Ins	I3-Ins	Avg.
ToolLLaMA		CoT	$51.8_{\pm 0.4}$	$53.1_{\pm 0.6}$	$46.4_{\pm 1.2}$	$51.6_{\pm 1.1}$	$48.9_{\pm 0.4}$	$37.2_{\pm 0.8}$	48.2
ToolLLaMA		DFS	$61.0_{\pm 1.8}$	$58.8_{\pm 0.6}$	$45.6_{\pm 1.2}$	$60.3_{\pm 1.1}$	$53.5_{\pm 0.4}$	$48.1_{\pm 0.8}$	54.6
GPT4-Turbo		CoT	$52.8_{\pm 1.3}$	$56.6_{\pm 0.9}$	$51.9_{\pm 0.5}$	$51.9_{\pm 1.1}$	$52.8_{\pm 0.4}$	$52.5_{\pm 0.8}$	53.1
GPT4-Turbo		DFS	$59.2_{\pm 0.5}$	$61.7_{\pm 0.7}$	$65.7_{\pm 1.0}$	$55.6_{\pm 0.6}$	$55.2_{\pm 0.4}$	$52.5_{\pm 4.3}$	60.6
TP-LLaMA		DFS	$55.0_{\pm 0.0}$	$65.0_{\pm 0.0}$	$\mathbf{80.0}_{\pm 0.0}$	$75.0_{\pm 0.0}$	$67.0_{\pm 0.0}$	$61.0_{\pm 0.0}$	65.0
Tool-Planner		P&S	$66.0_{\pm 0.0}$	$78.5_{\pm 0.0}$	$75.0_{\pm 0.0}$	$\mathbf{83.5}_{\pm 0.0}$	$77.5_{\pm 0.0}$	$\mathbf{83.0}_{\pm 0.0}$	**77.3**
Tool-Planner		P&S	$64.0_{\pm 0.0}$	$77.0_{\pm 0.0}$	$59.5_{\pm 0.0}$	$79.5_{\pm 0.0}$	$76.5_{\pm 0.0}$	$78.0_{\pm 0.0}$	72.4
InfCycle		P&S	$68.6_{\pm 0.4}$	$57.7_{\pm 0.4}$	$44.5_{\pm 1.6}$	$50.3_{\pm 0.8}$	$69.1_{\pm 1.9}$	$63.7_{\pm 4.3}$	59.0
InfCycle		P&S	$70.6_{\pm 1.8}$	$69.3_{\pm 0.0}$	$70.7_{\pm 0.5}$	$55.8_{\pm 0.6}$	$71.8_{\pm 0.7}$	$70.8_{\pm 1.0}$	68.2
InfCycle		P&S	$\mathbf{71.2}_{\pm 1.4}$	$\mathbf{80.8}_{\pm 0.9}$	$78.3_{\pm 0.4}$	$63.2_{\pm 0.7}$	$\mathbf{80.8}_{\pm 1.2}$	$77.9_{\pm 0.7}$	75.4

Table 1: We calculate the pass rates (%) by averaging the results of each model over three trials. All evaluations are conducted using GPT-4 Turbo, following official guidelines, to ensure comparability.

5 Experiments

We chose `StableToolBench` (Guo et al., 2024) and the `Berkeley Function-Calling` (Yan et al., 2024) benchmark to evaluate the effectiveness of our proposed method. `StableToolBench` requires real-time interaction with the RapidAPI[5] to gather feedback, primarily evaluating the model's performance in a dynamic environment. In contrast, `Berkeley Function-Calling` employs a static evaluation set that emphasizes the model's ability to extract complex APIs and parameters. To achieve this, we collected 6k APIs from RapidAPI and converted 2k code problems from LeetCode into usable APIs. For additional statistics and experimental details, please refer to the Appendix A.

5.1 StableToolBench

5.1.1 Evaluation Setup

In this experiment, **I1** represents intra-category multi-tool instructions, **I2** denotes intra-collection multi-tool instructions, and **I3** includes unseen instructions for the same tools as those in the training data. We categorized unseen tools into three groups: (1) **Ins** for unseen instructions related to the same tools, (2) **Tool** for unseen tools within the same (seen) category, and (3) **Cat** for unseen tools in a different category. We compared the performance of different models based on the official evaluation metrics: **Pass Rate**: This metric measures the proportion of successfully completed instructions within limited budgets, indicating the executability of instructions for LLMs. **Win Rate**: This metric involves providing an instruction along with two solution paths to a GPT evaluator, which determines the preferred solution.

5.1.2 Baselines

We compared several strong baselines: ToolLLaMA (Qin et al., 2023a), which is trained on distilled data from ChatGPT and employs depth-first tree search (DFS) for reasoning. TP-LLaMA (Chen et al., 2024), which leverages reinforcement learning on synthesized preference pairs. ToolPlanner (Liu et al., 2024a) introduces the Plan-and-Solve (P&S) (Wang et al., 2024) approach, which emphasizes task planning prior to function invocation. These planning methods often require models with strong reasoning capabilities and typically depend on closed-source LLMs. In contrast, our study explores data synthesis using less strong models to enhance reasoning ability, demonstrating an alternative pathway toward improving tool-use efficiency in open-source LLMs.

5.1.3 Results

Table 1 presents the pass rates of different models, indicating whether they successfully completed the given user queries. Table 2 presents the win rates by evaluating the inference paths of different models in comparison to GPT-4. When interacting with RapidAPI, InfCycle significantly outperforms previous models across six different test sets, achieving higher pass rates and win rates.

With different backbones (Mistral-7B-Instruct, LLaMA3-8B-Instruct, and Qwen2.5-7B-Instruct), our approach demonstrates outstanding tool invocation capabilities. Our data synthesis method relies solely on open-source models, with our 7B model outperforming GPT-4-based strategies such as ToolPlanner, illustrating the effectiveness and compatibility of our approach. Although TP-LLaMA also

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

Methods	Model	Inf	I1-I	I1-C	I1-T	I2-C	I2-I	I3-I	Avg.
ToolLLaMA		CoT	41.7	45.1	32.3	52.8	46.8	26.2	40.8
ToolLLaMA		DFS	42.3	51.0	31.0	67.0	54.0	31.1	54.0
GPT4-Turbo		CoT	71.2	77.1	61.4	79.2	71.8	67.2	71.3
GPT4-Turbo		DFS	73.0	75.2	68.4	77.4	66.9	60.7	70.2
TP-LLaMA		DFS	56.0	59.0	54.0	70.0	64.0	86.0	65.0
Tool-Planner		P&S	75.5	75.8	71.8	79.8	70.3	**92.0**	77.5
Tool-Planner		P&S	73.8	76.3	73.8	79.3	68.3	87.5	76.5
InfCycle		P&S	62.0	62.1	54.4	70.8	65.3	62.3	62.8
InfCycle		P&S	**78.5**	75.8	**77.8**	74.5	**78.2**	65.6	75.1
InfCycle		P&S	76.1	**86.9**	74.1	**81.1**	75.8	83.6	**79.6**

Table 2: We calculate the win rates (%) by averaging the results of each model over three trials. All evaluations are conducted using GPT-4 Turbo, following official guidelines, to ensure comparability.

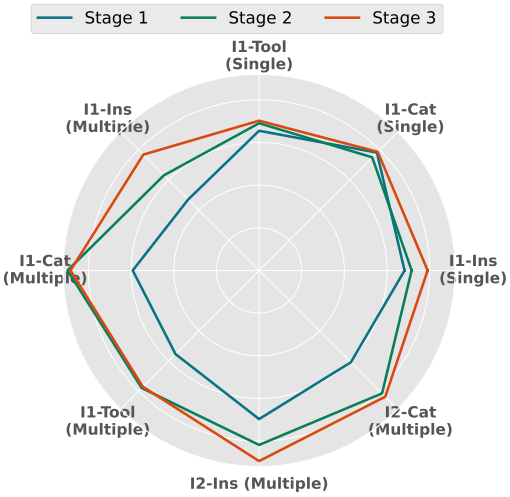

Figure 5: The API F1 scores of LLMs at different data synthesis stages.

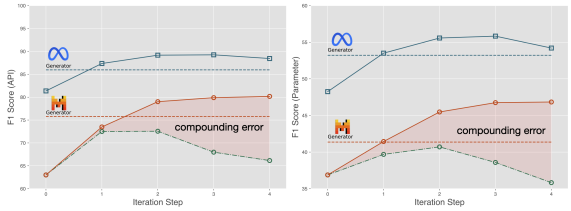

Figure 6: The figure illustrates the average API and Parameter F1 Scores on the human annotated StableToolBench testset across different data synthesis iterations.

leverages an open-source model for its Tree Search algorithm, which rely on preference learning fails to fully harness its tool invocation capabilities. This discrepancy highlights the gap between synthesized data and real data, emphasizing the importance of reliable verifiers to filter data and ensure accuracy.

5.1.4 Human Evaluation

Given that the evaluations in the main results rely on model judgments, which can often be unreliable (Wang et al., 2023a), we enhance accuracy by conducting human annotation of the StableToolBench test set. First, we remove inaccessible and invalid samples by interacting with RapidAPI Website and supplement them with new user queries from the same tool candidates of StableToolBench. Next, we categorize the collected samples into two groups: those that can fulfill the user's intent with a **single** API and those that require **multiple** APIs. Finally, we manually annotate the actual execution trajectories for these samples to ensure precision.

We examine InfCycle performance trends across various data synthesis stages using the API F1 score as our evaluation metric. This score evaluates the alignment between predicted and ground truth APIs, showcasing the model's ability to select appropriate tools. We deliberately use a small-size model (Mistral-7B-Instruct) as the backbone to determine if it can gradually synthesize higher-quality data. As Figure 5 illustrates, there is a clear improvement in performance throughout different stages. Notably, the most significant gains appear in testsets requiring multiple APIs reflecting the initial limitations of the small model in handling only simple tasks. Our multi-stage synthesis strategy enables the model to progressively acquire the capability to tackle complex scenarios and tasks.

5.1.5 The Effect of Reducing Compounding Errors

As shown in Figure 6, models incorporating iterative data synthesis exhibit consistent improvements in performance. Early in the synthesis process, models often filter out a substantial amounts of usable data, struggling to identify correct execu-

Method	InfP	Model	Abstract Syntax Tree (AST) Evaluation				Avg
			Simple	Multiple	Parallel	Parallel Multiple	
GPT-4o	Prompt		73.58	92.50	91.50	84.50	85.52
GPT-4o-mini	Prompt		79.67	89.50	89.00	88.00	86.54
o1-mini	Prompt		68.92	89.00	73.50	70.50	75.48
Command-R-Plus	FC		71.10	85.00	80.00	66.00	75.54
Open-Mixtral-8x22	Prompt		50.50	95.00	8.50	70.50	56.12
Mistral-7B	Prompt		0.70	0.00	0.00	0.00	0.18
Mistral-7B*	Prompt		19.83	60.50	2.00	22.00	26.08
InfCycle	Prompt		65.50$_{\uparrow 45.67}$	79.50$_{\uparrow 19}$	72.00$_{\uparrow 70}$	61.50$_{\uparrow 39.5}$	69.63
Meta-LLaMA-3-8B	Prompt		58.53	78.00	59.50	53.25	62.32
InfCycle	Prompt		67.00$_{\uparrow 8.47}$	90.00$_{\uparrow 12}$	81.00$_{\uparrow 21.5}$	76.50$_{\uparrow 23.25}$	78.63

Table 3: The model performance on BFCL. InfP represents the Inference Pattern, where FC directly returns JSON, while Prompt requires post-processing for results. Mistral-7B* uses the inference framework with InfCycle because the official scripts fail to produce effective calls.

tion trajectories. However, as the synthesized data grows, both Mistral and LLaMA effectively leverage the generated data, ultimately outperforming models that rely solely on CoT-based synthesis. We further conduct iterative experiments without applying cycle consistency filtering. Notably, as the number of synthesis iterations grows, the performance gap gradually expands. This highlights the critical role of reliable verifiers in improving model capabilities through inference-time computation.

5.2 Berkeley Function-Calling Benchmark

5.2.1 Evaluation Setup

In the non-live`Berkeley Function Calling` benchmark, four distinct test sets are included: The *Simple* dataset consists of tasks across three programming languages: Python, Java, and JavaScript. The *Multiple* and *Parallel* datasets, as described earlier, involve complex human intents that require multiple tool invocations to complete. The benchmark employs Abstract Syntax Tree (AST) evaluation to rigorously assess function-calling capabilities and diagnose specific model errors, such as incorrect function names, missing required parameters, and improper data types. For comparison, we select multiple baselines, including the state-of-the-art GPT-4 series, the powerful reasoning model OpenAI o1-mini, and several open-source models such as Command-R-plus and Open-Mistral-8x22B. Additionally, we use Mistral-7B-Instruct and Meta-LLaMA3-8B as foundation models for data synthesis and model training. It is worth noting that LLMs typically support both function call (FC) and prompt-based JSON-formatted output.

5.2.2 Results

As shown in Table 3, InfCycle significantly enhances tool use capabilities for Mistral-7B-Instruct and Meta-LLaMA3-8B. Before applying our methods, Mistral-7B-Instruct struggled to generate meaningful outputs, which fails to output JSON format response. With our multi-step reasoning framework and iterative training approach, Mistral-7B-Instruct achieves an improvement of nearly **43%** points. Even for the stronger Meta-LLaMA3-8B-Instruct, InfCycle still yields **16%** point performance boost. Notably, we did not select Qwen2.5-7B-Instruct as a backbone model because, it exhibited a strong tendency to overfit to the data. This suggests that Qwen LLMs requires better API data and a more robust verifier to facilitate self-improvement. Nevertheless, these results strongly demonstrate the effectiveness of our approach in enhancing LLMs' ability to autonomously refine their tool invocation capabilities.

6 Discussion and Conclusion

In this work, we enhance the model's tool use capabilities without relying on external supervision. Inspired by inference-time scaling, which increases the sampling space to enhance performance, this approach is particularly suitable for small size models to facilitate self-improvement. We demonstrate that InfCycle effectively synthesizes high-quality data using the LLMs and cycle consistency acting as process verifiers. In the future, we plan to incorporate more tools and parameters into our research. The trade-off between the number of parameters and inference cost remains a key focus, as

it directly impacts the efficiency and applicability. Additionally, we plan to conduct more experiments to investigate further scalability factors.

7 Limitation

Tool invocation is an essential capability for LLMs. However, different tools require diverse abilities to handle various tasks. In this work, we explore a preliminary tool usage scenario that involves only a limited set of tools and simple intents, without considering complex recursive intent structures. This remains one of the biggest challenges in current tool-use scenarios. Nevertheless, our approach demonstrates a viable pathway, enabling tool invocation based on the toolset itself without relying on labeled data. This idea still requires further validation in more complex real-world scenarios.

Acknowledgments

We want to thank all the anonymous reviewers for their valuable comments. This work was supported by the National Science Foundation of China (NSFC No. 62206194), the Natural Science Foundation of Jiangsu Province, China (Grant No. BK20220488), the Young Elite Scientists Sponsorship Program by CAST (2023QNRC001), and the Priority Academic Program Development of Jiangsu Higher Education Institutions. We also acknowledge MetaStone Tech. Co. for providing us with the software, optimisation on high performance computing and computational resources required by this work.

References

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. 2024. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*.

Sijia Chen, Yibo Wang, Yi-Feng Wu, Qing-Guo Chen, Zhao Xu, Weihua Luo, Kaifu Zhang, and Lijun Zhang. 2024. Advancing tool-augmented large language models: Integrating insights from errors in inference trees. *arXiv preprint arXiv:2406.07115*.

Chris Cundy and Stefano Ermon. 2023. Sequencematch: Imitation learning for autoregressive sequence modelling with backtracking. *arXiv preprint arXiv:2306.05426*.

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. 2024. Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models. *Preprint*, arXiv:2403.07714.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2023. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*.

Shijue Huang, Wanjun Zhong, Jianqiao Lu, Qi Zhu, Jiahui Gao, Weiwen Liu, Yutai Hou, Xingshan Zeng, Yasheng Wang, Lifeng Shang, et al. 2024. Planning, creation, usage: Benchmarking llms for comprehensive tool utilization in real-world complex scenarios. *arXiv preprint arXiv:2401.17167*.

Xian Li, Ping Yu, Chunting Zhou, Timo Schick, Omer Levy, Luke Zettlemoyer, Jason Weston, and Mike Lewis. 2023. Self-alignment with instruction backtranslation. *arXiv preprint arXiv:2308.06259*.

Yanming Liu, Xinyue Peng, Yuwei Zhang, Jiannan Cao, Xuhong Zhang, Sheng Cheng, Xun Wang, Jianwei Yin, and Tianyu Du. 2024a. Tool-planner: Dynamic solution tree planning for large language model with tool clustering. *arXiv preprint arXiv:2406.03807*.

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, et al. 2024b. Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets. *arXiv preprint arXiv:2406.18518*.

Alex Mallen, Akari Asai, Victor Zhong, Rajarshi Das, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. When not to trust language models: Investigating effectiveness of parametric and non-parametric memories. *arXiv preprint arXiv:2212.10511*.

Dheeraj Mekala, Jason Weston, Jack Lanchantin, Roberta Raileanu, Maria Lomeli, Jingbo Shang, and Jane Dwivedi-Yu. 2024. Toolverifier: Generalization to new tools via self-verification. *arXiv preprint arXiv:2402.14158*.

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

Shuofei Qiao, Ningyu Zhang, Runnan Fang, Yujie Luo, Wangchunshu Zhou, Yuchen Eleanor Jiang, Chengfei Lv, and Huajun Chen. 2024. Autoact: Automatic agent learning from scratch via self-planning. *arXiv preprint arXiv:2401.05268*.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, Yi Ren Fung, Yusheng Su, Huadong Wang, Cheng Qian, Runchu Tian, Kunlun Zhu, Shihao Liang, Xingyu Shen, Bokai Xu, Zhen Zhang, Yining Ye, Bowen Li, Ziwei Tang, Jing Yi, Yuzhang Zhu, Zhenning Dai, Lan Yan, Xin Cong, Yaxi Lu, Weilin Zhao, Yuxiang Huang, Junxi Yan, Xu Han, Xian Sun, Dahai Li, Jason Phang, Cheng

Yang, Tongshuang Wu, Heng Ji, Zhiyuan Liu, and Maosong Sun. 2023a. Tool learning with foundation models. *Preprint*, arXiv:2304.08354.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023b. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2024. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*.

Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, et al. 2023. Restgpt: Connecting large language models with real-world restful apis. *arXiv preprint arXiv:2306.06624*.

Kaya Stechly, Matthew Marquez, and Subbarao Kambhampati. 2023. Gpt-4 doesn't know it's wrong: An analysis of iterative prompting for reasoning problems. *arXiv preprint arXiv:2310.12397*.

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. *arXiv preprint arXiv:2306.05301*.

Karthik Valmeekam, Matthew Marquez, and Subbarao Kambhampati. 2023. Can large language models really improve by self-critiquing their own plans? *arXiv preprint arXiv:2310.08118*.

L Wang, W Xu, Y Lan, Z Hu, Y Lan, RK Lee, and E Lim. 2024. Plan-and-solve prompting: improving zero-shot chain-of-thought reasoning by large language models (2023). *arXiv preprint arXiv:2305.04091*.

Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Qi Liu, Tianyu Liu, and Zhifang Sui. 2023a. Large language models are not fair evaluators. *arXiv preprint arXiv:2305.17926*.

Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Ma, and Yitao Liang. 2023b. 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, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Binfeng Xu, Zhiyuan Peng, Bowen Lei, Subhabrata Mukherjee, Yuchen Liu, and Dongkuan Xu. 2023a. Rewoo: Decoupling reasoning from observations for efficient augmented language models. *arXiv preprint arXiv:2305.18323*.

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

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. 2024. Gpt4tools: Teaching large language model to use tools via self-instruction. *Advances in Neural Information Processing Systems*, 36.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. 2022. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. 2023. Agenttuning: Enabling generalized agent abilities for llms. *arXiv preprint arXiv:2310.12823*.

Yuchen Zhuang, Xiang Chen, Tong Yu, Saayan Mitra, Victor Bursztyn, Ryan A Rossi, Somdeb Sarkhel, and Chao Zhang. 2023. Toolchain*: Efficient action space navigation in large language models with a* search. *arXiv preprint arXiv:2310.13227*.

A Dataset and Experiment Details

A.1 StableToolBench Dataset

In this study, we use the StableToolBench (Guo et al., 2024) environment and test set for method validation, rather than ToolBench (Qin et al., 2023b). StableToolBench applies manual filtering and caching of HTTP requests to reduce API failures. As shown in Table 4, ToolBench has a significantly higher proportion of failed APIs, making StableToolBench more reliable for ensuring comparability and enabling a fair evaluation of model performance. However, it's important to note that StableToolBench generates virtual request results using GPT-4, which can introduce biases during data synthesis. Therefore, we do not use the StableToolBench environment for synthesizing training data.

Benchmark	I1-Ins	I1-Cat	I1-Tool	I2-Ins	I2-Cat	I3-Ins	Sum.
ToolBench	200	200	200	200	200	200	1100
StableToolBench	163	153	158	106	124	61	765

Table 4: This table illustrates the proportion of solvable data filtered out by StableToolBench.

A.2 Human Annotated StableToolBench Dataset

For each user query, we conducted real-time access to the RapidAPI website and constructed the following testset, which includes both Single and Multiple types of user queries. As shown in Table 5, each sample includes the required API, corresponding parameters, and the access sequence.

Benchmark	I1-Ins (S)	I1-Cat (S)	I1-Tool (S)	I1-Ins (M)	Sum.
StableToolBench	151	64	113	28	506
	I1-Cat (M)	I1-Tool (M)	I2-Ins (M)	I2-Cat (M)	
	36	46	22	46	

Table 5: Statistics on the sample counts for different datasets (S represents Single samples, and M represents Multiple samples).

A.3 Berkeley Function Calling Dataset

We also gathered data statistics on the Berkeley Function Calling in Table 6. This evaluation dataset primarily focuses on Python but includes other programming languages (such as Java and JavaScript), which increases the performance requirements for base models, as many models may not be proficient in languages beyond Python.

Python	Java	JavaScript	Multiple	Parallel	Parallel_Multiple	Sum.
400	100	50	200	200	200	1150

Table 6: This table shows the distribution of different types of data.

A.4 Training Details

In our experiments, we utilize Mistral-7B-Instruct-v0.2, LLaMA3-8B-Instruct, and Qwen2.5-7B-Instruct as the foundation models, and the training process is conducted based on the alignment-handbook framework in a multi-round conversation mode. During the 1-epoch supervised fine-tuning (SFT) phase, we use a total batch size of 8, a learning rate of 7.0e-06, and a maximum sequence length of 4096. For the 1-epoch direct preference optimization (DPO) phase, we maintain a total batch size of 2, a learning rate of 5.0e-7, and a maximum sequence length of 1024, with the β parameter set to 0.01. All experiments are conducted on a single machine equipped with 8 NVIDIA A100 GPUs, each with 40GB of memory.

A.5 Prompts

In this section, we present the key prompt templates we use in our data synthesis process, as shown in Figure 7, 8, and 9. We do not carefully select these prompts, as we focus on leveraging iterative synthesis techniques to generate data, rather than employing them for reasoning during inference. This approach emphasizes data generation while ensuring modeling flexibility.

Prompt for the Generator to Generate Query

Simple Query
You are a query generator tasked with creating realistic and natural queries based on a provided API call. Please generate a specific and complex user query based on the given API information.
Requirements:
- Realistic: The query should reflect what actual users might inquire about when trying to understand or utilize the function in real scenarios.
- Fluent: The query should be well-structured, clear, and free of grammatical errors.
- Parameter Reasoning: The generated query should demonstrate an understanding of the function's parameters, reasoning about how they should be correctly used or what their values should be.
Please don't mention API in the user query, but it needs to contain the parameters required to call the APIs. Now generate query description for given API function call.
Input: function calling = {}

Parallel Query
You are a query generator tasked with creating realistic and natural query based on provided API call which necessitates multiple times using different parameter-value pairs.
The query should align with realistic user needs, structuring a scenario where the function's application is clearly required.
Requirements:
- Logical Flow: The query should be logically structured to necessitate multiple API calls, with the sequence of calls matching the order of parameters provided.
- Parameter Inclusion: Incorporate all necessary parameters into the query. The query should allow each paramerter for API call to be logically derived from the context.
- Clarity and Realism: The user query must be clear, grammatically correct, and realistically framed, resembling a genuine request that might prompt such an API interaction in a real-world application.
Please don't mention API in the user query, but it needs to contain the parameters required to call the APIs. Now generate query description for given API function call chain.
Input: function calling = {}

Multiple Query
You are a query generator tasked with creating realistic and natural queries based on a provided API call. Using the provided list of APIs, select and combine given APIs to create a specific and complex user query.
Requirements:
- Establish Logical Relationships: Ensure that the API calls are logically related and form a coherent sequence. The query should reflect a natural flow where the output of one API informs the next, or where multiple APIs are combined to achieve a complex objective.
- Parameter Validity: Construct the query so that valid parameters for each API call can be inferred from the context. Ensure the query provides sufficient information to logically deduce the required parameters where applicable.
- Clarity and Realism: The user query must be clear, grammatically correct, and realistically framed, resembling a genuine request that might prompt such an API interaction in a real-world application.
Please don't mention API in the user query, but it needs to contain the parameters required to call the APIs. Now generate query description for given API function call chain.
Input: function calling = {}

Figure 7: Instruction prompt for query generation.

Prompt for the Generator to Generate Golden Trajectory

Simple Plan
Given [User Query], [Function call] that can be used to solve the query, your task is to provide a concise, logical, and well-organized task plan centered around API function call to address the query
Provide the task plan without any "execution details", "result handling", or "error management".
***
Input:
[User Query]:{query}
[Function Call]:{function_call}
***
Please generate a plan in one sentence:

Simple Answer
Given [User Query], [Function call] with returned response used to resolve the query, your task is to generate a concise, coherent, and reasonable answer based on the available information from the function.
Requirements:
- Ensure fluency and clarity: The answer should be well-structured and articulated in a clear, fluent, and natural manner.
- Match the function response: The answer must directly reflect the response of the function call. Avoid adding any outside knowledge or assumptions not provided by the function. - Correct details: The answer must fulfill the user's requirements and resolve the query satisfactorily. Now generate an answer for the given query and function call.
***
Input:
[User Query]:{query}
[Function Call]:{function_call}
***
Please generate an answer:

Final Answer
Given [User Query] and [Subtask with Subanswer], your task is to summarize the results of executing all subtasks and effectively consolidate these results to provide a comprehensive and accurate answer to the user query. The final answer should be detailed, complete, and well-structured, directly addressing the user query.
Requirements:
- Subanswer Utilization: The final answer must be based entirely on the subanswer, without incorporating any external knowledge or assumptions.
- Answer Resolution: Ensure that the final answer fulfills the user's requirements and resolves the query satisfactorily.
- Answer Quality: The final answer should be clear, detailed, and logically structured, providing a high-quality solution to the user query.
***
Input:
[User Query]:{query}
[Subtask with Subanswer]:{context}
***
Please generate a summary answer:

Figure 8: Instruction prompt for trajectory generation.

Prompt for the Generator Semantic Checker

Query Check
Please compare the given query with the target function call parameter and determine if they match. The evaluation criteria include:
- Query Clarity and Coherence: Assess whether the query is articulated in a clear, fluent, and natural manner.
- Parameter Derivability: Ensure that all parameter values required for the function call can be either directly extracted from the query or logically inferred based on the provided information. Please note that common sense knowledge can be used to infer parameters from problems.
If all the above criteria are met please first output YES/NO, and then give reasons for the judgment.
***
Input:
[query]: {query}
[parameters]: {parameters}
***
Output:

Single Answer Check
Please evaluate if the current answer effectively addresses the user query based on the information provided by the function response. The evaluation criteria include:
- Response Utilization: Check if the answer is obtained based on the given function response.
- Query Resolution: Determine whether the answer fulfills the user's requirements and resolves the query satisfactorily.
- Clarity and Coherence: Assess whether the answer is articulated in a clear, fluent, and natural manner.
If all the above criteria are met please first output YES/NO, and then give reasons for the judgment.
***
Input:
[query]: {query}
[function response]:{function_resp}
[answer]:{answer}
***
Output:

Final Answer Check
Please assess whether the given answer effectively solves the user's problem and whether the language is smooth, fluent, accurate, and concise. Please consider whether the answer responds directly to the query, is complete, and is clearly expressed.
If all the above criteria are met please first output YES/NO, and then give reasons for the judgment.
***
Input:
[query]:{query}
[answer]:{answer}
***
Output:

Figure 9: Instruction prompt for semantic checker