

TOOL-USE REFINER: A LIGHTWEIGHT PLUG-AND-PLAY MODULE FOR ENHANCING LLM TOOL-USE

Anonymous authors

Paper under double-blind review

ABSTRACT

Large Language Models (LLMs) have demonstrated remarkable capabilities in Tool-Integrated Reasoning (TIR). However, the practical application is often hindered by frequent errors in tool invocation, such as incorrect parameters or malformed formats. Prevailing training paradigms like Supervised Fine-Tuning (SFT) and Reinforcement Learning (RL) can mitigate these issues but demand substantial computational resources. To address the limitation, we propose a novel, resource-efficient refinement framework that enhances the tool-use capabilities of large-scale LLMs without directly training on them. We introduce a small-scale model, termed the Tool-use Refiner, which operates as a post-processing module. This Refiner takes the initial tool-integrated reasoning from an upstream LLM and the user’s task as input, then performs its own reasoning to correct and enhance the invocation. The Refiner is trained using an advanced RL algorithm, Decoupled Clip and Dynamic Sampling Policy Optimization (DAPO), to ensure efficient and stable policy learning. On a diverse set of tool-use and reasoning benchmarks, our Refiner improves task completion rates and invocation accuracy over the raw outputs of various upstream LLMs. This highlights our Refiner as a lightweight, plug-and-play solution for improving the operational reliability of LLM-based agents. We release our code and model to facilitate future research.

1 INTRODUCTION

Recent advances in Large Language Models (LLMs) have enabled them to perform complex reasoning and interact with external tools (Schick et al., 2023; Wang et al., 2024). Techniques such as Tool-Integrated Reasoning (TIR) allow LLMs to dynamically invoke tools during conversations, significantly enhancing their problem-solving capabilities in areas like knowledge retrieval, program design, and agent task execution (Yao et al., 2023; Qin et al., 2024).

However, even state-of-the-art (SOTA) models frequently produce erroneous tool invocations, including incorrect tool names, invalid parameters, wrong tool-call order, or malformed invocation formats (Liu et al., 2024; Qian et al., 2025). These errors may arise from misinterpretations of user queries, failure to maintain context, or insufficient reasoning depth. Such inaccuracies can propagate through downstream applications, leading to inefficiencies or failures in real-world deployments.

Training LLMs for TIR tasks has primarily relied on Supervised Fine-Tuning (SFT) or Reinforcement Learning (RL) (Zeng et al., 2023; Feng et al., 2025). While these methods have shown promising results, they require considerable computational resources, as they involve direct training on large-scale LLMs. Moreover, when updates or corrections are needed, retraining becomes necessary, further consuming time and resources. Given these limitations, a compelling research question arises: *Is it possible to enhance large-scale LLMs on TIR tasks without directly training them, while using only minimal training resources?*

Inspired by self-refinement and model cascading techniques (Ji et al., 2024; Ngweta et al., 2024), we propose a novel refinement paradigm that introduces a small-scale LLM as a **Tool-use Refiner**. This model is designed to correct and enhance tool-invocations from a larger upstream LLM. Specifically, when a user submits a task, the upstream model generates an initial tool invocation. Our Refiner then processes both the user’s task and the upstream model’s output, performing its own tool-integrated reasoning to identify and rectify potential errors. The refined output is expected to align more closely with the user’s intent and the constraints of the available tools.

To train the Refiner, we employ an enhanced reinforcement learning algorithm called Decoupled Clip and Dynamic Sampling Policy Optimization (DAPO) (Yu et al., 2025), an extension of Group Relative Policy Optimization (GRPO) Shao et al. (2024) that supports more efficient and stable policy learning in complex reasoning. After the training, we evaluate our approach on several tool-use benchmarks covering general invocation, sequence ordering, and search invocation, demonstrating that the refined outputs lead to significant performance improvements over the raw outputs from the upstream model. Moreover, the Tool-use Refiner can be seamlessly attached to various upstream LLMs, offering plug-and-play versatility. These results validate that even a small Refiner, trained with minimal resources, effectively enhances tool invocation accuracy and task completion rates. This approach avoids the need for extensive modifications to the upstream LLM, substantially reducing the required resource.

The main contributions of this work are as follows:

- We propose a refinement-based pipeline that employs a small-scale **Tool-use Refiner** to correct and improve tool invocations generated by a larger upstream LLM.
- We construct training data and reward functions tailored for tool invocation correction, and then employ the DAPO RL algorithm to train our Refiner, which circumvents the need for resource-intensive fine-tuning on the upstream LLMs.
- We demonstrate that our plug-and-play approach effectively rectifies tool invocation errors (e.g., format, sequencing) for diverse upstream LLMs, yielding consistent performance gains across benchmarks and surpassing several direct fine-tuning methods.

This work highlights the potential of a small-scale, specialized LLM to correct larger LLMs in tool-integrated scenarios, providing a plug-and-play solution for deploying reliable LLM-agent systems. Our code is available at <https://anonymous.4open.science/r/Tool-use-Refiner-8C2D>.

2 RELATED WORKS

Tool-Integrated Reasoning. Tool-Integrated Reasoning enables LLMs to perform reasoning and acting through tool invocation. Specifically, FreshLLMs (Vu et al., 2023) and Search-R1 (Jin et al., 2025) empower LLMs to utilize search engines, boosting their ability to handle general knowledge and domain-specific questions. ToRL (Li et al., 2025) and PoT (Chen et al., 2022) strengthen the programming capabilities of LLMs, allowing them to execute code for task completion. MAMmoTH (Yue et al., 2023) and MathCoder (Wang et al., 2023) leverage multiple tools to address complex mathematical problems. Furthermore, frameworks such as xLAM (Zhang et al., 2024) and ToolACE (Liu et al., 2024) explore the use of LLMs as central controllers capable of orchestrating a wide array of tool calls, facilitating the construction of AI agent systems.

Self-Refinement. LLMs struggle to generate flawless outputs for complex tasks in a single attempt. Drawing inspiration from how humans refine their written text, a paradigm known as Self-Refinement has emerged, which enhances LLMs’ outputs through successive iterations. Specifically, Self-Refine (Madaan et al., 2023) directs an LLM to critique its own generation, leading to performance gains in areas such as dialogue generation, mathematical reasoning, and programming tasks. Idea2Img (Yang et al., 2024) utilizes feedback loops to refine text-to-image prompts, resulting in images that more faithfully adhere to design instructions. Self-Debugging (Chen et al., 2023b) empowers LLMs to rectify their own programs by inspecting execution outcomes and the code’s logic. Furthermore, self-correction has also been applied to e-commerce platforms, achieving better product attribute extraction by refining LLMs’ outputs (Brinkmann & Bizer, 2025).

From a methodological standpoint, beyond having an LLM generate its own feedback, an alternative involves using an external, powerful critic model (Hu et al., 2024). More recently, Aligner (Ji et al., 2024) introduces an additional, small-scale LLM specifically fine-tuned for refinement tasks, which is used to correct the outputs of upstream LLMs. Our method is conceptually aligned with this approach by introducing a small-scale LLM for refinement. This ensures low training and inference overhead and allows for “plug-and-play” integration with various upstream LLMs.

Reinforcement Learning for LLMs. Reinforcement learning (RL) has become a cornerstone for aligning LLMs with human preferences and downstream tasks. Early approaches primarily utilized

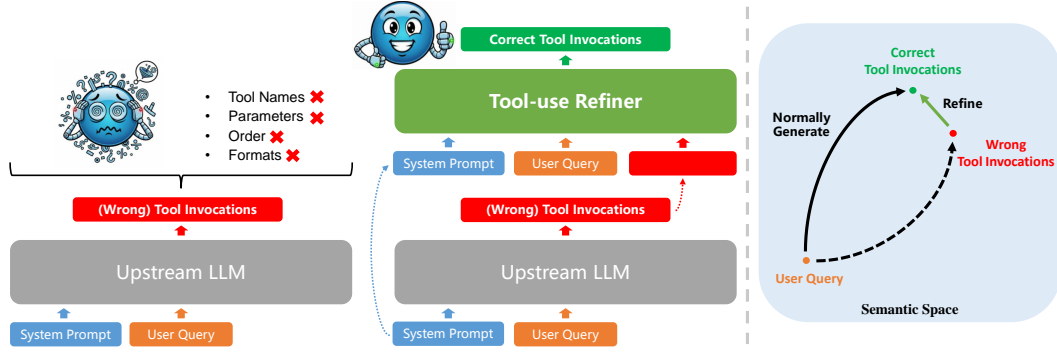


Figure 1: **(Left) Workflow of the Tool-Use Refiner.** The refiner receives a system prompt, a user query, and an upstream LLM’s output (containing potentially wrong tool invocations) as input, and outputs the refined tool invocations. **(Right) Comparison of semantic gap for refinement and conventional generation.** The refiner’s task is to navigate the short path from flawed tool invocations to their corrected version, whereas the conventional generation must bridge the much larger gap from the user query to the correct tool invocations. This makes the refiner easier to train.

Proximal Policy Optimization (PPO) (Schulman et al., 2017), an actor-critic framework that employs a separate critic network to estimate the advantage of generated actions. While effective, PPO introduces significant computational overhead, as the value function is a model comparable in size to the policy itself. To address the limitation, Group Relative Policy Optimization (GRPO) (Shao et al., 2024) was proposed as a streamlined alternative. It eliminates the value model by estimating advantages in a group-relative manner, thereby improving training efficiency.

Building upon GRPO, Decoupled Clip and Dynamic Sampling Policy Optimization (DAPO) (Yu et al., 2025) introduces four key refinements: First, it employs an asymmetric clipping range ($1 - \epsilon_{low}, 1 + \epsilon_{high}$), which better prevents policy entropy collapse by allowing more flexibility for policy improvement. Second, DAPO incorporates dynamic sampling, which filters out samples that provide no learning signal. Third, it introduces a soft penalty on overlong responses to discourage verbosity without harsh truncation. Finally, DAPO adopts a token-level loss, averaging the loss across all tokens in a batch rather than assigning an equal weight for each sample.

LLM Training in Tool Use Area. The predominant approach for training LLMs on tool use tasks has been Supervised Fine-Tuning (SFT) (Chen et al., 2023a; Zeng et al., 2023; Chen et al., 2024; Schick et al., 2023). In this paradigm, models are trained on a static dataset of expert-annotated trajectories, using a language modeling loss to determine which API call actually helps in predicting future tokens or generating better results. However, SFT-trained models tend to overfit to demonstrated trajectories and lack the dynamic capacity for trial-and-error learning or error recovery. To surmount the challenges, a growing body of work has shifted towards RL to refine the decision-making policies of LLMs. RL allows the model to explore the vast action space of possible tool invocations and learn directly from feedback signals that correlate with task success. To this end, researchers employ algorithms such as PPO and GRPO to train the model’s tool-calling policy (Qian et al., 2025; Li et al., 2025; Jin et al., 2025; Song et al., 2025). These RL-based approaches foster greater generalization and robustness, enabling the model to perform more reliably in dynamic and open-ended scenarios where optimal solutions cannot be easily enumerated. Following this trajectory, our paper also adopts an RL framework and employs the DAPO algorithm to train a small-scale Refiner specifically designed to rectify erroneous tool invocations.

3 METHODOLOGY

3.1 TOOL-USE REFINER

Given that direct fine-tuning of large-scale LLMs is computationally expensive, we propose a more efficient alternative by introducing a small-scale **Tool-use Refiner**, where Qwen3-1.7B (Team, 2025) serves as the foundational model for our main experiments. This specialized module is exclu-

sively trained to correct erroneous tool invocations generated by diverse larger, upstream LLMs. It functions as a post-processing step, enhancing the final output quality while keeping the upstream models frozen, ensuring both efficiency and generalizability.

As illustrated in Figure 1, the Refiner takes three inputs: the original system prompt, the user query, and the tool-integrated reasoning produced by the upstream LLM (including the tool invocations). Its sole task is to process these inputs and perform its own reasoning to generate the correct tool invocations. The detailed prompt structure used to guide the Refiner is provided in Appendix A.

Furthermore, as shown in Figure 1, the Refiner is trained within a Self-Refinement paradigm. This paradigm presents an easier learning problem (Ji et al., 2024), since the semantic distance between a flawed tool invocation and its corrected version is considerably shorter than that between a high-level user query and the final tool invocation. The Refiner’s task is simplified to learning a corrective mapping from a nearly-correct state to a correct one.

3.2 TRAINING DATA CONSTRUCTION

This section details the construction methodology for our training data. The dataset is composed of two primary categories: conventional tool-calling datasets and nested tool-calling datasets. The conventional category comprises the glaive (AI, 2023), ToolACE (Liu et al., 2024), and xLAM (Zhang et al., 2024) datasets. The typical task in these datasets is to select the correct tool from a given list based on a user query. In addition, the nested category utilizes the NesTools (Han et al., 2024) dataset, whose task is more complex, requiring the model to manage a sequence of tool invocations where the output of one tool becomes the input for the next. Detailed examples from these datasets are provided in Appendix B.

To create a suitable training dataset for our Tool-use Refiner, which is designed to correct tool invocations, we structure the data into a (*The Original Task, Upstream LLM Output, Correct Tool Invocation*) triplet format. Accordingly, we generate four types of training data, categorized based on the type of upstream LLM outputs:

- **Erroneous Invocations:** This data contains incorrect tool invocations. We generate them by using five different LLMs (varying in scale and performance) to introduce errors into the ground-truth invocations. Each model is specifically instructed to introduce formatting errors, incorrect tool names, and erroneous tool parameters.
- **Correct Invocations:** These are the original, ground-truth tool invocations, which are used to train the Refiner not to alter correct tool invocations into incorrect ones.
- **Shuffled Nested Invocations (Full Format):** This data consists of nested tool calls from the NesTools dataset with the execution order randomized, while retaining the complete tool call format including parameters.
- **Shuffled Nested Invocations (Simplified Format):** Similar to the third type, this data also features randomized tool call orders but simplifies the task by requiring the model to output only the tool names, excluding any parameters (Ablation Study in Appendix H).

The final dataset is composed of 2,250 instances of Type 1 and 450 instances of Type 2, created by sampling equally from the three source datasets. For Type 3 and Type 4, we generate 1,000 instances each. The detailed construction procedures are available in Appendix C.

3.3 RULE-BASED REWARD FUNCTION

Rule-based reward functions have shown strong empirical performance and are widely employed in recent RL training (Shao et al., 2024). Our training with the DAPO algorithm also incorporates a multi-faceted rule-based reward function, where the total reward assigned to a given model output is an aggregate of several components, each targeting a specific aspect of correctness, as follows:

Format Reward. Format reward assesses the structural integrity and syntactic correctness of the tool invocation. It verifies two primary conditions: (1) the presence of all required special tags, and (2) the adherence to a predefined, resolvable format as specified in the system prompt (e.g., a JSON-like dictionary or a Python function signature). The validation is performed recursively, inspecting

each element within nested structures like lists and dictionaries to ensure complete compliance. R_{format} is set to 1 only when all criteria are satisfied, and to 0 otherwise.

Tool Name Reward. Tool Name reward evaluates the accuracy of the tool names invoked by the model. Let \mathcal{O} represent the set of tool calls generated by the model and \mathcal{G} be the set of ground-truth. We define O_n as the set of tool names in \mathcal{O} , and G_n as the set of tool names in \mathcal{G} . $R_{ToolName}$ is calculated based on the Jaccard similarity between these two sets, scaled to the range $[-2, 2]$:

$$R_{ToolName} = 4 \cdot \frac{|O_n \cap G_n|}{|O_n \cup G_n|} - 2$$

Tool Parameter Accuracy Reward. This reward is divided into two distinct yet complementary components: the accuracy of parameter names and the correctness of their corresponding values. First, the **parameter name reward** measures the recall of parameter names, ensuring all necessary parameters are identified. Let O_{pn} be the set of all unique parameter names present in the model’s generated output, and G_{pn} be the set from the ground-truth. Second, the **parameter content reward** assesses the fidelity of the parameter values. It also employs a recall-based formulation. Let O_{pc} be the multiset of parameter values in the model’s output and G_{pc} be the multiset from the ground-truth. Both rewards are scaled to the range $[-2, 2]$:

$$R_{ParamName} = 4 \cdot \frac{|O_{pn} \cap G_{pn}|}{|G_{pn}|} - 2, R_{ParamContent} = 4 \cdot \frac{|O_{pc} \cap G_{pc}|}{|G_{pc}|} - 2$$

Tool Order Reward. For tasks where the sequence of tool invocations is critical, this reward assesses the ordinal correctness. Let $S_G = (g_1, g_2, \dots, g_m)$ be the ground-truth sequence of the tool invocation and $S_O = (o_1, o_2, \dots, o_n)$ be the model-generated sequence. The reward is based on the proportion of tools that are correctly placed in the sequence relative to maximum length between ground-truth sequence and model-generated sequence, scaled to the range $[-2, 2]$:

$$R_{Order} = 4 \cdot \frac{\sum_{i=1}^{\min(m,n)} \mathbb{I}(g_i = o_i)}{\max(m, n)} - 2$$

where $\mathbb{I}(\cdot)$ is the indicator function, which returns 1 if the condition is true and 0 otherwise.

Penalty for Refinement Regression. To prevent the refinement process from degrading a correct tool invocation, we introduce a targeted penalty mechanism. Let $R_{pre-refine}$ be the total reward for the model’s output before refinement, and $R_{post-refine}$ be the total reward after refinement. If $R_{post-refine} < R_{pre-refine}$, the final reward for the sample will be overridden and set to the minimum possible value of the reward function. This mechanism strongly discourages “regressive” refinements.

Reward Normalization. Given the heterogeneity of our training data, where different samples may require different subsets of the aforementioned reward components (e.g., some tasks are order-agnostic and thus do not use R_{Order}), the theoretical minimum and maximum achievable reward can vary per sample. To ensure a consistent reward scale for the learning algorithm, we normalize the final aggregated reward. For each sample, the calculated total reward R_{total} is mapped from its specific theoretical range $[\min_reward_i, \max_reward_i]$ to a normalized range of $[0, 1]$.

3.4 RL TRAINING WITH DAPO

We train our Tool-use Refiner via reinforcement learning (RL) using the DAPO algorithm (see Section 2 for algorithm details). During training, the Refiner processes inputs comprising a system prompt, a user query, and the potentially flawed tool invocations from an upstream LLM, structured as per Section 3.1. The model’s goal is to output a corrected version of these tool invocations. To guide the process, we use the rewards defined in Section 3.3 as the training signal and compute the group relative advantages for the Refiner’s outputs to update the model. The hyperparameters and training details are available in Appendix D.

Table 1: Results on API-Bank Benchmark (General-Purpose)

Model	Overall Acc.	Δ	Level 1	Level 2	Level 3
Qwen2.5-3B-Instruct (Raw)	51.59	-	59.65	32.84	36.64
Qwen2.5-3B-Instruct (SFT)	52.76	+2.27%	59.65	50.75	32.82
Qwen2.5-3B-Instruct (SFT+PPO)	65.16	+26.30%	67.92	55.22	61.83
Qwen2.5-3B-Instruct (SFT+GRPO)	62.48	+21.11%	68.67	58.21	45.80
Qwen2.5-3B-Instruct (PPO Cold Start)	57.62	+11.69%	64.66	59.70	35.11
Qwen2.5-3B-Instruct (ToolRL)	67.00	+29.87%	73.43	67.16	47.33
Qwen2.5-3B-Instruct (Ours, w/ Refiner)	68.84	+33.44%	73.18	67.16	56.49
Qwen2.5-7B-Instruct (Raw)	62.48	-	70.68	49.25	44.27
Qwen2.5-7B-Instruct (SFT)	50.59	-19.03%	55.89	50.75	34.35
Qwen2.5-7B-Instruct (SFT+PPO)	63.15	+1.07%	72.43	58.21	37.40
Qwen2.5-7B-Instruct (SFT+GRPO)	54.10	-13.41%	61.40	52.24	32.82
Qwen2.5-7B-Instruct (PPO Cold Start)	61.64	-1.34%	68.67	44.78	48.85
Qwen2.5-7B-Instruct (ToolRL)	64.66	+3.49%	73.93	61.19	38.17
Qwen2.5-7B-Instruct (Ours, w/ Refiner)	73.70	+17.96%	78.70	67.16	61.83
Llama-3.2-3B-Instruct (Raw)	40.54	-	44.86	29.85	32.82
Llama-3.2-3B-Instruct (SFT)	52.76	+30.14%	60.65	35.82	37.40
Llama-3.2-3B-Instruct (SFT+PPO)	57.79	+42.55%	63.16	47.76	46.56
Llama-3.2-3B-Instruct (SFT+GRPO)	56.78	+40.06%	63.60	41.79	43.51
Llama-3.2-3B-Instruct (PPO Cold Start)	55.78	+37.59%	60.65	41.79	48.09
Llama-3.2-3B-Instruct (ToolRL)	59.13	+45.86%	65.66	52.24	42.75
Llama-3.2-3B-Instruct (Ours, w/ Refiner)	64.15	+58.24%	66.17	68.66	55.73
Llama-3.1-8B-Instruct (Raw)	67.17	-	71.93	62.69	54.96
Llama-3.1-8B-Instruct (Ours, w/ Refiner)	68.51	+1.99%	71.93	67.16	58.78
Qwen3-1.7B (Raw)	60.13	-	61.40	43.28	64.89
Qwen3-1.7B (Ours, w/ Refiner)	70.18	+16.71%	73.43	65.67	62.60
Qwen3-8B (Raw)	70.85	-	73.43	65.67	65.65
Qwen3-8B (Ours, w/ Refiner)	74.37	+4.97%	77.19	64.18	70.99

4 EVALUATION

4.1 OVERVIEW

To comprehensively evaluate the capability of our Tool-use Refiner in correcting erroneous tool invocations, we assess its performance on several benchmarks across a diverse range of tasks:

- **Refinement of General-Purpose Tool Invocation:** We use a general-purpose tool invocation benchmark called API-bank, which comprises 73 diverse API tools (Li et al., 2023), to assess the Refiner’s ability to correct fundamental invocation errors in general scenario, including incorrect formats, tool names, and parameter names or contents (**Section 4.2**).
- **Refinement of Sequential Errors:** We choose a nested tool-use benchmark called NESTFUL, which requires sequential tool calls to solve mathematical problems (Basu et al., 2024), to evaluate the correction of tool invocation order (**Section 4.3**).
- **Refinement of Search Tool Invocations:** We use two out-domain question-answering benchmarks, Musique and Bamboogle (Trivedi et al., 2022; Press et al., 2022), to assess the Refiner’s ability to enhance search tool invocations in the retrieval scenario (**Section 4.4**).
- **Quantitative Analysis of Format and Content Correction:** As the Bamboogle dataset is suitable for quantitative analysis, we adapt it to specifically evaluate the model’s effectiveness in format and content correction (**Section 4.5**).

4.2 REFINEMENT OF GENERAL-PURPOSE TOOL INVOCATION

Experiment Settings. To evaluate our Refiner’s capability in refining general-purpose tool invocation, we use the API-Bank benchmark. We adopt the evaluation script from ToolRL and compare our method with the baselines reported therein. The dataset and baseline details are in Appendix E.1. The results are shown in Table 1. Δ indicates the percentage improvement over a Raw baseline.

Significant Performance Gains by Correcting Format Errors. As demonstrated in Table 1, our Tool-use Refiner outperforms the methods that directly finetune the upstream LLMs. The improvement is particularly pronounced for weaker raw models, such as Qwen2.5-3B-Instruct and Llama-3.2-3B-Instruct. Upon reviewing the cases, we observe that these weaker models are prone

to generating format errors, including the omission of special tags and the use of incomplete or unmatched parentheses. The Refiner proves highly effective at rectifying these structural issues. Conversely, for stronger base models that inherently produce fewer format errors, the performance gains are more modest.

Analysis of Semantic Error Correction. We also investigate the Refiner’s ability to correct semantic errors, which include incorrect tool names and invalid parameters. We observe that weaker raw models often generate conspicuous semantic errors (e.g., calling a wrong tool). Our Refiner is highly effective at rectifying these conspicuous mistakes. However, its performance diminishes when faced with more nuanced errors. While some semantic mistakes are rectified, the Refiner still struggles to resolve them comprehensively, highlighting a key area for future improvement. Nevertheless, by leveraging the Refiner’s combined format and semantic correction, upstream LLMs achieve significant gains, outperforming all direct training methods in the table.

Impact of Chain-of-Thought Quality on Refinement Efficacy. In our methodology, the Refiner is also provided with the Chain-of-Thought (CoT) trace from the upstream LLM. We discover this to be a critical component, as the Refiner can use this to inform its refinement process. As shown in Table 1, an example arises when comparing Qwen3-1.7B and Llama-3.1-8B-Instruct. Although the latter has a stronger baseline performance, the former generates a higher-quality CoT. As a result, the Refiner delivers a more significant improvement to Qwen3-1.7B, underscoring that the quality of the intermediate reasoning path is a key determinant of the final refined output’s accuracy.

4.3 REFINEMENT OF SEQUENTIAL ERRORS

Experiment Settings. We assess the Refiner’s efficacy in correcting tool invocation order using the NESTFUL benchmark. This benchmark tasks models with executing a logical sequence of tool invocations (e.g., mathematical calculations) and storing intermediate results in temporary variables (e.g., `var.0`) for subsequent use. Details are shown in Appendix E.2. Performance is measured across three key metrics: (1) **Part. Acc.**: The average ratio of correct tool invocations within each sample. (2) **Full Acc.**: The percentage of samples with entirely correct tool invocations. (3) **Win Rate**: The percentage of samples achieving both perfect tool format and a correct final answer.

Table 2 presents the evaluation results. We report the performance improvements for six evaluated models, comparing their raw and CoT versions after being augmented by our Refiner (scores are presented in the “Raw / CoT” format). For a broader context, we also include results from other leading tool-calling models, such as Granite (Abdelaziz et al., 2024), Mixtral (Jiang et al., 2024), DeepSeek-V3 (Guo et al., 2024), and GPT-4o (Hurst et al., 2024).

Consistent Performance Gains. As demonstrated in Table 2, our Refiner yields consistent performance gains for both the raw models and the CoT enhanced models. Specifically, after the refinement, the enhanced models outperform nearly all other tool-calling models listed in the table, despite the latter having a markedly larger number of parameters.

Further case analysis demonstrates that our Refiner can perform a range of sophisticated corrections. Specifically, it is capable of rectifying format errors in tool invocations, inserting intermediate variables, and eliminating redundant parameter outputs. Moreover, it can optimize the overall execution plan by reordering the sequence of tool invocations and supplementing it with necessary but previously omitted ones.

Robustness to Flawed Reasoning. Table 2 reveals a counter-intuitive finding: enabling CoT causes a performance drop for some models. The case study indicates that this issue stems from the generation of low-quality reasoning chains, which misguide the models’ tool invocation decisions and reduce final accuracy. However, our Refiner proves highly effective in mitigating this problem. While the Refiner improves both raw and CoT-enabled models, the performance gains are more pronounced for the latter. As a result, their final performance reaches a comparable level, with the CoT-enhanced versions occasionally outperforming their raw counterparts. This demonstrates the robustness of our Refiner, as it effectively rectifies tool-calling errors without being misled by the preceding faulty reasoning chains.

Table 2: Results on NESTFUL Benchmark (Sequential Errors). Scores are reported as “Raw / CoT”.

Model	#Parameters	Part. Acc.	Full Acc.	Win Rate
xLAM-1b-fc-r	1B	0.09	0.03	0.02
xLAM-7b-fc-r	7B	0.23	0.15	0.14
Hammer2.0-7b	7B	0.29	0.22	0.25
ToolACE-8B	8B	0.13	0.00	0.00
Granite-20B-FunctionCalling	20B	0.26	0.21	0.20
Mixtral-8x7B-Instruct-v0.1	46.7B	0.14	0.09	0.09
Mixtral-8x22B-Instruct-v0.1	141B	0.28	0.21	0.23
xLAM-8x22b-fc-r	141B	0.25	0.17	0.06
Llama-3-1-405B-Instruct-fp8	405B	0.13	0.07	0.14
DeepSeek-V3	685B	0.37	0.29	0.60
GPT-4o (2024-08-06)	UNK	0.38	0.28	0.60
Llama-3.2-3B-Instruct	3B	0.13/0.11	0.04/0.03	0.04/0.04
Llama-3.2-3B-Instruct (Ours, w/ Refiner)	3B	0.19/0.21	0.09/0.12	0.13/0.17
Llama-3.1-8B-Instruct	8B	0.22/0.21	0.16/0.15	0.11/0.09
Llama-3.1-8B-Instruct (Ours, w/ Refiner)	8B	0.27/0.27	0.19/0.20	0.25/0.25
Qwen2.5-3B-Instruct	3B	0.15/0.10	0.09/0.04	0.11/0.13
Qwen2.5-3B-Instruct (Ours, w/ Refiner)	3B	0.21/0.19	0.13/0.10	0.21/0.23
Qwen2.5-7B-Instruct	7B	0.24/0.25	0.19/0.18	0.25/0.18
Qwen2.5-7B-Instruct (Ours, w/ Refiner)	7B	0.27/0.28	0.21/0.22	0.29/0.31
Qwen3-1.7B	1.7B	0.17/0.15	0.11/0.12	0.12/0.20
Qwen3-1.7B (Ours, w/ Refiner)	1.7B	0.21/0.22	0.13/0.15	0.19/0.31
Qwen3-8B	8B	0.29/0.25	0.23/0.21	0.25/0.33
Qwen3-8B (Ours, w/ Refiner)	8B	0.30/0.29	0.23/0.22	0.29/0.41

Table 3: Results on search benchmarks

	Musique	$\Delta_{Musique}$	Bamboogle	$\Delta_{Bamboogle}$
<i>Qwen2.5-3B-Instruct</i>				
RAG	0.047/0.094	-	0.080/0.324	-
Search-R1	0.103	+119.15%	0.264	+230.00%
Ours w/ Refiner	0.103	+9.57%	0.352	+8.64%
<i>Qwen2.5-7B-Instruct</i>				
RAG	0.058/0.129	-	0.208/0.376	-
Search-R1	0.146	+151.72%	0.368	+76.92%
Ours w/ Refiner	0.144	+11.63%	0.408	+8.51%
<i>Llama-3.2-3B-Instruct</i>				
RAG	0.052	-	0.212	-
Ours w/ Refiner	0.060	+15.38%	0.232	+9.43%
<i>Llama-3.1-8B-Instruct</i>				
RAG	0.119	-	0.324	-
Ours w/ Refiner	0.130	+9.24%	0.328	+1.23%
<i>Qwen3-1.7B</i>				
RAG	0.144	-	0.216	-
Ours w/ Refiner	0.183	+27.08%	0.224	+3.70%
<i>Qwen3-8B</i>				
RAG	0.125	-	0.400	-
Ours w/ Refiner	0.150	+20.00%	0.408	+2.00%

Table 4: Quantitative evaluation of correction efficacy (with Bamboogle*)

Model	Acc _{Full}	Acc _{Toolname}	Acc _{Parameter}
<i>Qwen2.5-3B-Instruct</i>			
CoT	0.00	54.27	52.27
Ours, w/ Refiner	48.13	79.67	79.47
<i>Qwen2.5-7B-Instruct</i>			
CoT	20.87	60.40	60.40
Ours, w/ Refiner	49.13	79.47	79.47
<i>Llama-3.2-3B-Instruct</i>			
CoT	1.60	54.20	45.70
Ours, w/ Refiner	36.53	74.73	73.80
<i>Llama-3.1-8B-Instruct</i>			
CoT	43.07	64.80	60.67
Ours, w/ Refiner	60.27	72.20	72.00
<i>Qwen3-1.7B</i>			
CoT	16.70	62.27	59.40
Ours, w/ Refiner	39.67	68.60	66.80
<i>Qwen3-8B</i>			
CoT	80.33	86.87	86.80
Ours, w/ Refiner	86.60	87.13	87.13

4.4 REFINEMENT OF SEARCH TOOL INVOCATIONS

Experiment Settings. To assess whether our Refiner enhances the upstream LLM in retrieval-augmented scenarios, we conduct evaluations on the Musique and Bamboogle benchmarks. For a rigorous comparison, we adopt the evaluation protocol of Search-R1, where models generate answers using multi-turn retrieved content. Further details are provided in Appendix E.3.

Nevertheless, we identify a limitation in the original evaluation script, which relies on exact match (EM) for assessing correctness. Upon manual inspection, we find that numerous model outputs, while not identical to the ground-truth, are semantically correct (i.e., one is a substring or superset of the other). Consequently, we modify the evaluation script to use a containment-based metric.

The final results are presented in Table 3. Scores are reported in the format of “original / revised”, representing the performance under the EM and our modified metric, respectively. Δ indicates the percentage improvement over a RAG baseline, with each comparison being made under its corresponding evaluation metric.

Performance Enhancement Constrained by the Upstream Model. As shown in Table 3, our Refiner yields performance gains for upstream LLMs across various scales, which validates its efficacy in correcting the upstream LLM’s output to generate well-formatted `<search>` call requests and extract more effective search queries. However, the performance gains are less pronounced compared to Search-R1, a method that fine-tunes the upstream LLM itself for search engine utilization. We attribute this performance gap to their different training objectives. Search-R1 not only trains the model to generate superior search queries but also to effectively utilize the retrieved information. In contrast, our Refiner focuses solely on improving the quality of the search invocation, while the upstream LLM remains unoptimized. Consequently, even when provided with higher-quality retrieved information, the upstream LLM can not leverage this information well.

4.5 QUANTITATIVE ANALYSIS OF FORMAT AND CONTENT CORRECTION

Experiment Settings. Given the well-structured nature of the Bamboogle benchmark, which features predefined retrieval categories and queries limited to celebrity birth information (dates or places), we reformulate its search queries into a tool-calling format, yielding the Bamboogle* dataset. This modified dataset enables a quantitative assessment of the model’s tool invocation accuracy, which we dissect by individually analyzing the correctness of the tool format, name, and parameters. Detailed specifications for this evaluation are provided in Appendix E.4.

The evaluation results are presented in Table 4. We report three key metrics: (1) Acc_{Full} : The percentage of predictions where the tool names, parameters, and format are all correct. (2) $\text{Acc}_{Toolname}$: The percentage of predictions that contain the correct tools, regardless of its parameters or format. (3) $\text{Acc}_{Parameter}$: The percentage of predictions that contain the correct tools and the correct parameters, irrespective of the format.

Validating the Refiner’s Contribution to Format and Content Accuracy. As shown in Table 4, integrating the Refiner leads to a substantial improvement in Acc_{Full} , demonstrating its high effectiveness in format correction. Moreover, $\text{Acc}_{Toolname}$ and $\text{Acc}_{Parameter}$ also show significant gains, which validates that the Refiner can extract more effective information from the user query and context, enhancing the overall tool-calling capability of the upstream LLM.

5 OVERVIEW OF TECHNICAL SUPPLEMENT

The appendices provide a comprehensive technical supplement to our work. We first detail the input prompt design of our Refiner, the training data curation process, and the complete training and evaluation configurations (Appendix A, B, C, D, E). Subsequently, we present rigorous ablation studies to demonstrate that the Refiner’s strong performance stems from our proposed training strategy and data choices, specifically by: comparing performance against an untrained baseline, ablating the overlong penalty, and ablating the use of simplified call sequence data (Appendix F, G, H). Finally, we include a disclosure on the use of LLM and provide case studies to substantiate our findings (Appendix I, J).

6 CONCLUSION

We introduce a novel refinement paradigm centered on a small-scale **Tool-use Refiner**, designed to correct tool invocations generated by larger, upstream LLMs. This approach directly addresses the pervasive issue of invocation errors in Tool-Integrated Reasoning while circumventing the high computational costs and inflexibility associated with directly fine-tuning large-scale LLMs. We demonstrate that by training this lightweight refiner using the DAPO algorithm, it can effectively learn to identify and rectify a range of invocation errors, including incorrect formats, parameters, and sequences. Our comprehensive evaluations across multiple tool-use benchmarks reveal that this plug-and-play approach leads to consistent performance gains for various upstream models. Notably, the refined outputs not only surpass the raw LLM generations but also outperform several direct fine-tuning methods, validating the efficacy of our refinement strategy. This work highlights the potential of using smaller, specialized models to augment the capabilities of larger ones, paving the way for more robust, adaptable, and deployable AI systems in real-world applications.

ETHICS STATEMENT

This submission does not have any ethics issues to the best of our knowledge.

REPRODUCIBILITY STATEMENT

We are committed to the reproducibility of this research. The source code for our experiments is anonymously available in <https://anonymous.4open.science/r/Tool-use-Refiner-8C2D>. Descriptions of our methodology, the datasets, and the benchmarks, including all preprocessing procedures, can be found in the main text and the appendix. Further details on the experimental setup required to reproduce our results are also provided in the appendix.

REFERENCES

- Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Sadhana Kumaravel, Matthew Stallone, Rameswar Panda, Yara Rizk, GP Bhargav, Maxwell Crouse, Chulaka Gunasekara, et al. Granite-function calling model: Introducing function calling abilities via multi-task learning of granular tasks. *arXiv preprint arXiv:2407.00121*, 2024.
- Glaive AI. glaive-function-calling-v2, August 2023. URL <https://huggingface.co/datasets/glaiveai/glaive-function-calling-v2>.
- AI@Meta. Llama 3 model card. 2024. URL https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.
- Kinjal Basu, Ibrahim Abdelaziz, Kiran Kate, Mayank Agarwal, Maxwell Crouse, Yara Rizk, Kelsey Bradford, Asim Munawar, Sadhana Kumaravel, Saurabh Goyal, et al. Nestful: A benchmark for evaluating llms on nested sequences of api calls. *arXiv preprint arXiv:2409.03797*, 2024.
- Alexander Brinkmann and Christian Bizer. Self-refinement strategies for llm-based product attribute value extraction. *arXiv preprint arXiv:2501.01237*, 2025.
- Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. Fireact: Toward language agent fine-tuning. *arXiv preprint arXiv:2310.05915*, 2023a.
- Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.
- Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023b.
- Zehui Chen, Kuikun Liu, Qiuchen Wang, Wenwei Zhang, Jiangning Liu, Dahua Lin, Kai Chen, and Feng Zhao. Agent-flan: Designing data and methods of effective agent tuning for large language models. *arXiv preprint arXiv:2403.12881*, 2024.
- Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.
- Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, et al. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*, 2022.
- Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjuan Zhong. Retool: Reinforcement learning for strategic tool use in llms. *arXiv preprint arXiv:2504.11536*, 2025.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

- Han Han, Tong Zhu, Xiang Zhang, Mengsong Wu, Hao Xiong, and Wenliang Chen. Nestools: A dataset for evaluating nested tool learning abilities of large language models. *arXiv preprint arXiv:2410.11805*, 2024.
- Chi Hu, Yimin Hu, Hang Cao, Tong Xiao, and Jingbo Zhu. Teaching language models to self-improve by learning from language feedback. *arXiv preprint arXiv:2406.07168*, 2024.
- Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.
- Jiaming Ji, Boyuan Chen, Hantao Lou, Donghai Hong, Borong Zhang, Xuehai Pan, Tianyi Alex Qiu, Juntao Dai, and Yaodong Yang. Aligner: Efficient alignment by learning to correct. *Advances in Neural Information Processing Systems*, 37:90853–90890, 2024.
- Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.
- Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Serkan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-rl: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.
- Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick SH Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In *EMNLP (1)*, pp. 6769–6781, 2020.
- Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*, 2023.
- Xuefeng Li, Haoyang Zou, and Pengfei Liu. Torl: Scaling tool-integrated rl. *arXiv preprint arXiv:2503.23383*, 2025.
- Qiqiang Lin, Muning Wen, Qiuying Peng, Guanyu Nie, Junwei Liao, Jun Wang, Xiaoyun Mo, Jiamu Zhou, Cheng Cheng, Yin Zhao, et al. Hammer: Robust function-calling for on-device language models via function masking. *arXiv preprint arXiv:2410.04587*, 2024.
- Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, et al. Toolace: Winning the points of llm function calling. *arXiv preprint arXiv:2409.00920*, 2024.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.
- Lilian Ngweta, Mayank Agarwal, Subha Maity, Alex Gittens, Yuekai Sun, and Mikhail Yurochkin. Aligners: Decoupling llms and alignment. *arXiv preprint arXiv:2403.04224*, 2024.
- Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah A Smith, and Mike Lewis. Measuring and narrowing the compositionality gap in language models. *arXiv preprint arXiv:2210.03350*, 2022.
- Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiushi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. Toolrl: Reward is all tool learning needs. *arXiv preprint arXiv:2504.13958*, 2025.
- Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Xuanhe Zhou, Yufei Huang, Chaojun Xiao, et al. Tool learning with foundation models. *ACM Computing Surveys*, 57(4):1–40, 2024.
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.

- John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.
- Huatong Song, Jinhao Jiang, Yingqian Min, Jie Chen, Zhipeng Chen, Wayne Xin Zhao, Lei Fang, and Ji-Rong Wen. R1-searcher: Incentivizing the search capability in llms via reinforcement learning. *arXiv preprint arXiv:2503.05592*, 2025.
- Qwen Team. Qwen3 technical report, 2025. URL <https://arxiv.org/abs/2505.09388>.
- Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. Musique: Multihop questions via single-hop question composition. *Transactions of the Association for Computational Linguistics*, 10:539–554, 2022.
- Tu Vu, Mohit Iyyer, Xuezhi Wang, Noah Constant, Jerry Wei, Jason Wei, Chris Tar, Yun-Hsuan Sung, Denny Zhou, Quoc Le, et al. Freshllms: Refreshing large language models with search engine augmentation. *arXiv preprint arXiv:2310.03214*, 2023.
- Hongru Wang, Yujia Qin, Yankai Lin, Jeff Z Pan, and Kam-Fai Wong. Empowering large language models: Tool learning for real-world interaction. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 2983–2986, 2024.
- Ke Wang, Houxing Ren, Aojun Zhou, Zimu Lu, Sichun Luo, Weikang Shi, Renrui Zhang, Linqi Song, Mingjie Zhan, and Hongsheng Li. Mathcoder: Seamless code integration in llms for enhanced mathematical reasoning. *arXiv preprint arXiv:2310.03731*, 2023.
- Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. Text embeddings by weakly-supervised contrastive pre-training. *arXiv preprint arXiv:2212.03533*, 2022.
- Zhengyuan Yang, Jianfeng Wang, Linjie Li, Kevin Lin, Chung-Ching Lin, Zicheng Liu, and Lijuan Wang. Idea2img: Iterative self-refinement with gpt-4v for automatic image design and generation. In *European Conference on Computer Vision*, pp. 167–184. Springer, 2024.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.
- Qiyang Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.
- Xiang Yue, Xingwei Qu, Ge Zhang, Yao Fu, Wenhao Huang, Huan Sun, Yu Su, and Wenhao Chen. Mammoth: Building math generalist models through hybrid instruction tuning. *arXiv preprint arXiv:2309.05653*, 2023.
- Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. Agenttuning: Enabling generalized agent abilities for llms. *arXiv preprint arXiv:2310.12823*, 2023.
- Jianguo Zhang, Tian Lan, Ming Zhu, Zuxin Liu, Thai Hoang, Shirley Kokane, Weiran Yao, Juntao Tan, Akshara Prabhakar, Haolin Chen, et al. xlam: A family of large action models to empower ai agent systems. *arXiv preprint arXiv:2409.03215*, 2024.

APPENDIX

We present further details and experimental results in the appendix, organized as follows:

1. Supplementary Model Details and Data Curation:

- **Appendix A. Crafting the Input Prompt for Tool-use Refiner:** We present the template and format of the input prompt for our Tool-use Refiner.
- **Appendix B. Examples from the Source Tool-calling Datasets:** We present examples from the source tool-calling datasets, including Glaive, ToolACE, xLAM, and NesTools.
- **Appendix C. Crafting the Training Datasets for Tool-use Refiner:** We detail the process of creating the training datasets for our Tool-use Refiner.

2. Experimental Setup and Training Details:

- **Appendix D. Training Details:** We present the training configurations for the Tool-use Refiner used in our main experiment, along with its reward and response length curves during the training process.
- **Appendix E. Evaluation Setup:** We present our evaluation setup, detailing our assessments on refining general-purpose tool invocations, sequential errors, and search tool invocations, alongside a quantitative correction analysis.

3. Model Performance and Ablation Studies:

- **Appendix F. Performance Comparison with an Untrained Refiner:** We compare our Tool-use Refiner against an untrained counterpart (Qwen3-1.7B), demonstrating that its strong performance is attributable to our RL training for tool invocation correction, rather than the inherent capabilities of the base model.
- **Appendix G. An Ablation Study of the Overlong Penalty:** We conduct an ablation study on the overlong penalty, highlighting its necessity for guiding the Refiner to produce concise yet effective reasoning.
- **Appendix H. An Ablation Study of the Simplified Call Sequence Data:** We conduct an ablation study on the simplified call sequence data, demonstrating that this data yields a more stable Refiner with improved performance on both general tool-use and specific tool-order correction tasks.

4. Supplementary Materials and Disclosure:

- **Appendix I, The Use of Large Language Models (LLMs):** We disclose the use of LLM within our writing process.
- **Appendix J, Case Studies:** We present representative case studies to substantiate the findings from our comparative evaluation.

A CRAFTING THE INPUT PROMPT FOR TOOL-USE REFINER

This section outlines the structure of the input prompt for our Tool-use Refiner.

The input begins with a comprehensive system prompt designed to define the Refiner’s role and operational parameters. First, the system prompt explicitly instructs the Tool-use Refiner that its primary function is to identify and rectify errors within tool invocations generated by other models. Furthermore, it details the expected input prompt format and enumerates the potential categories of errors the refiner may encounter. The complete System Prompt utilized in our experiments is presented below:

Your role is a highly professional tool-calling instruction corrector. Your primary task is to identify and fix errors in tool-calling instruction generated by other models. Note that the original tool-calling instruction from another model may already be entirely correct - in such cases, you do not need to make any modifications. You will be provided with: the list of available tools, the user’s

original query and the tool-calling instruction generated by another model. Your responsibility is to carefully analyze this information and output only the corrected, properly formatted tool-calling instruction and the special tags. When identifying format inaccuracies, character errors, special token mistakes, or tool-calling order errors, confidently implement corrections. If you identify any errors in the tool-calling instruction from another model, correct it and output only the fixed, correct tool-calling instruction and the special tags.

Subsequently, for each call of the Refiner, we provide the system prompt for the current task, the user query for the current sample, and the potentially erroneous tool invocations generated by the upstream LLMs. The specific input is structured as follows:

The original system prompt and the list of available tools are: "SYSTEM". The user's original query is: "QUERY". The tool-calling instruction generated by another model is: "UPSTREAM". Please correct the instruction from another model and output only the correct tool-calling instruction.

Within this template, the placeholders are defined as follows: SYSTEM refers to the original system prompt of the task, QUERY contains the user query for the current sample, and UPSTREAM holds the tool invocations from the upstream LLM. It is important to note that while this structured format is strictly adhered to during the training phase, we allow for minor modifications during inference. Such adjustments, tailored to specific scenarios, can be implemented to obtain optimal performance.

B EXAMPLES FROM THE SOURCE TOOL-CALLING DATASETS

B.1 GLAIVE

The tool invocation format of the glaive dataset is: `<functioncall> { "name": function_name, "arguments": { "param_name": param_value, "param_name2": param_value2, ... } } <|endoftext| >`. Each invocation typically calls only one tool at a time.

An example of a system prompt is: *You are a helpful assistant with access to the following functions. Use them if required - { "name": "generate_password", "description": "Generate a random password", "parameters": { "type": "object", "properties": { "length": { "type": "integer", "description": "The length of the password" }, "include_symbols": { "type": "boolean", "description": "Whether to include symbols in the password" } }, "required": ["length"] } } { "name": "create_task", "description": "Create a new task in a task management system", "parameters": { "type": "object", "properties": { "title": { "type": "string", "description": "The title of the task" }, "due_date": { "type": "string", "format": "date", "description": "The due date of the task" }, "priority": { "type": "string", "enum": ["low", "medium", "high"], "description": "The priority of the task" } }, "required": ["title", "due_date", "priority"] } }.*

An example of a user query (including conversations) is: *USER: I need a new password. Can you generate one for me? ASSISTANT: Of course. How long would you like your password to be? And would you like it to include symbols? <|endoftext| > USER: I would like it to be 12 characters long and yes, please include symbols.*

An example of a correct tool invocation is: `<functioncall> { "name": "generate_password", "arguments": { "length": 12, "include_symbols": true } } <|endoftext| >`

B.2 TOOLACE

The tool invocation format of the ToolACE dataset is: `[func1(params_name=params_value, params_name2=params_value2...), func2(params)]`. This is a list containing Python functions. It's worth noting that some function names in the dataset contain spaces, which could not be properly parsed. Such samples are filtered out during preprocessing.

To train the Refiner to reconstruct missing special tags, we enclose the ToolACE format with the tags `<function_list>` and `</function_list>` at the beginning and end, respectively.

An example of a system prompt is: *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 function 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 call in tools call sections. Here is a list of functions in JSON format that you can invoke: [{"name": "Financial.Fundamentals.API", "description": "Retrieves the profitability (ROA ratio) for a specified financial year of a specific share.", "parameters": {"type": "dict", "properties": {"shareuid": {"description": "Unique identifier for the share searched", "type": "int"}, "from": {"description": "Start string of the searched period in American notation year-month-day with leading 0", "type": "string"}, "to": {"description": "End string of the searched period in American notation year-month-day with leading 0", "type": "string"}}, "required": ["shareuid", "from", "to"]}, {"name": "Get Supported Currencies", "description": "Retrieve a list of supported currencies by Coinbase.", "parameters": {"type": "dict", "properties": {}, "required": []}, {"name": "watchlists", "description": "Returns a list of private watchlists for the authenticating user.", "parameters": {"type": "dict", "properties": {"callback": {"description": "Define your own callback function name, add this parameter as the value.", "type": "string", "default": ""}}, "required": ["callback"]}, {"name": "Should you decide to return the function call(s). Put it in the format of [func1(params_name=params_value, params_name2=params_value2...), func2(params)]. NO other text MUST be included.

An example of a user query is: Can you please provide the ROA ratio for the company with shareuid 6789 for the last financial year?

An example of a correct tool invocation is: [{"name": "Financial.Fundamentals.API", "results": {"roa_ratio": 0.123, "financial_year": "2025"}}]

B.3 xLAM

The tool invocation format of the xLAM dataset is: [{"name": funcio_name, "arguments": {params_name: params_value, params_name2: params_value2...}}, ...]. This is a list containing multiple Python dictionaries.

To train the Refiner to reconstruct missing special tags, we enclose the xLAM format with the tags <func_call> and </func_call> at the beginning and end, respectively.

An example of a available tool list is: [{"name": "live_giveaways_by_type", "description": "Retrieve live giveaways from the GamerPower API based on the specified type.", "parameters": {"type": {"description": "The type of giveaways to retrieve (e.g., game, loot, beta).", "type": "str", "default": "game"}}}].

An example of a user query is: Where can I find live giveaways for beta access and games?

An example of a correct tool invocation is: [{"name": "live_giveaways_by_type", "arguments": {"type": "beta"}}, {"name": "live_giveaways_by_type", "arguments": {"type": "game"}}].

B.4 NESTOOLS

The tool invocation format of the NesTools dataset is:

[{"api_name": →, "parameters": {"arg0": "value0", "arg1": "value1", ...}, "responses": ["API_call_0", ..., "API_call_n"]}, {"api_name": →, "parameters": {"arg0": "value0", "arg1": "value1", ...}, "responses": ["API_call_{n+1}", ...]}, ...].

This format captures sequential information, with the "step" field indicating the order in which the tool is called. Additionally, the output of each tool invocation is represented using API_call_x.

To train the Refiner to reconstruct missing special tags, we enclose the NesTools format with the tags <nested_function> and </nested_function> at the beginning and end, respectively.

An example of a available tool list is: [{"api_name": "scan_isbn", "api_description": "Scan the ISBN of a book to retrieve information.", "parameters": {"isbn": {"type": "str", "description": "the ISBN code of the book"}}, "required": ["isbn"], "responses": {"book_details": {"type": "str", "description": "detailed information about the book"}, "availability": {"type": "bool", "description": "availability status of the book in the library"}}, {"api_name": "locate_book", "api_description": "Locate the physical book in the library.", "parameters": {"book_info": {"type": "str", "description": "detailed information about the book"}}, "re-

quired”: [“book_info”], “responses”: {“location_desc”: {“type”: “str”, “description”: “description of the exact location within the library”}}, {“api_name”: “engage_ar_experience”, “api_description”: “Activate an augmented reality experience related to the book.”, “parameters”: {“availability”: {“type”: “bool”, “description”: “availability status of the book”}, “exact_location”: {“type”: “str”, “description”: “exact location of the book”}}, “required”: [“availability”, “exact_location”], “responses”: {“ar_message”: {“type”: “str”, “description”: “message to explain the AR experience”}, “ar_duration”: {“type”: “int”, “description”: “estimated duration of the AR experience in minutes”}}}]].

An example of a user query is: Scan the ISBN “978-3-16-148410-0” of a book to extract comprehensive data and verify its presence in the library. Upon identifying the book situated, launch an augmented reality interaction correlating to the book.

An example of a correct tool invocation is: [{“api_name”: “scan_isbn”, “parameters”: {“isbn”: “978-3-16-148410-0”}, “responses”: [“API_call_0”, “API_call_1”]}, {“api_name”: “locate_book”, “parameters”: {“book_info”: “API_call_0”}, “responses”: [“API_call_2”]}, {“api_name”: “engage_ar_experience”, “parameters”: {“availability”: “API_call_1”, “exact_location”: “API_call_2”}, “responses”: [“API_call_3”, “API_call_4”]}].

C CRAFTING THE TRAINING DATASETS FOR TOOL-USE REFINER

C.1 GENERATION OF ERRONEOUS TOOL INVOCATIONS

To construct a training dataset of erroneous tool invocations for our Tool-use Refiner, we systematically generate flawed data by leveraging five LLMs with diverse scales and performance characteristics: Qwen3-1.7B, Qwen3-8B, Qwen3-14B (Team, 2025), Llama-3.1-8B-Instruct, and Llama-3.2-3B-Instruct (AI@Meta, 2024). The core strategy is to instruct these models to deliberately introduce errors into tool invocations, focusing on three specific types of corruption: incorrect formatting, incorrect tool or parameter names, and incorrect parameter content.

Our generation pipeline involves a two-step process. In the first step, we have each LLM generate an initial response to the original problem, yielding a set of tool invocations which we refer to as the “original answer.” These initial generations might be correct or faulty, reflecting the natural capabilities of each model. In the second step, we reframe the task. Instead of generating a correct answer, the models are prompted to intentionally modify the “original answer” to introduce a specific error. The instructions provided are as follows, wherein the term “refined answer” refers to correct tool invocations:

- **For Incorrect Format:** Your task is to modify the ‘original answer’ below so that the tool call format differs from the ‘refined answer’ and becomes incorrect. The ‘original answer’ is: “ORIGINAL”. You only need to output your modified ‘original answer’.
- **For Incorrect Tool/Parameter Name:** Your task is to modify the ‘original answer’ below so that the tool names or tool call parameter names differ from the ‘refined answer’ and become incorrect. The ‘original answer’ is: “ORIGINAL”. You only need to output your modified ‘original answer’.
- **For Incorrect Parameter Content:** Your task is to modify the ‘original answer’ below so that the tool call parameter content differs from the ‘refined answer’ and becomes incorrect. The ‘original answer’ is: “ORIGINAL”. You only need to output your modified ‘original answer’.

This process yields a collection of tool invocations with intentionally injected errors from each of the five LLMs. From this collection, we curate our final dataset by randomly sampling 150 instances for each of the three error types from each model. To ensure diversity, each set of 150 samples is composed of 50 instances originating from the glaive dataset, 50 from ToolACE, and 50 from xLAM. This strategy produces a total of 2,250 erroneous tool invocation samples, which simulate the outputs of an upstream LLM in our training framework.

C.2 GENERATION OF RANDOMIZED CALL SEQUENCES

To train the Refiner’s capability in correcting tool call sequences, we leverage nested tool data (NesTools) and generate a training set of randomized call sequences by permuting the ground-truth solutions. We create two distinct batches of this data: one in a full format and another in a simplified format.

The full-format batch adheres to the original NesTools data structure, requiring the model to output the tool name, its parameters, and a temporary placeholder for the output (e.g., `API_call_x`). Conversely, the simplified-format batch only requires the model to output the name of the tool to be called at each step, omitting parameters and placeholders. The introduction of the simplified format constitutes a form of curriculum learning, enabling the model to first master the simpler task of sequence correction before progressing to the more complex task.

Specifically, to generate the full-format randomized call sequences, we employ Qwen3-14B to inject sequential errors into the original NesTools data using the following prompt:

Your task is to modify the “original answer” below so that the tool call order becomes incorrect. The ‘original answer’ is: “ORIGINAL”. You only need to output your modified ‘original answer’.

For the simplified-format randomized call sequences, we first convert the NesTools data into a streamlined structure: `<order_func>[{"step": 1, "tool_list": [tool_name1, tool_name2, ...]}, {"step": 2, "tool_list": [tool_name3, tool_name4, ...]}, ...]</order_func>`. Subsequently, we utilize Qwen3-14B to introduce errors into this simplified data with the following prompt:

Your task is to modify the “original answer” below so that the tool call order and the number of steps becomes incorrect. This involves either splitting tools from a single step into multiple steps (e.g., converting [{"step": 1, "tool_list": [tool_name1, tool_name2]}] into [{"step": 1, "tool_list": [tool_name1]}, {"step": 2, "tool_list": [tool_name2]}]) or merging/redistributing tool names from different steps into incorrect step groupings. The ‘original answer’ is: “ORIGINAL”. You only need to output your modified ‘original answer’.

A key outcome of this generation process is that the synthesized data inherently includes a mix of correct sequences, incorrect sequences, and malformed instances. This diverse composition perfectly suits our requirement for a challenging training corpus. Each of the two resulting datasets contains a total of 1,000 samples.

D TRAINING DETAILS

D.1 TRAINING CONFIGURATIONS

For the training phase, we configure the following hyperparameters. The learning rate is set to $1e-6$, and the Refiner is trained for a single epoch. We use a maximum context window of 4096 tokens. The clipping ratio for the optimization algorithm is configured with a lower clipping range (`clip_ratio_low`) of 0.2 and a higher clipping range (`clip_ratio_high`) of 0.28.

The training batch size is set to 16. For each sample in a batch, 16 corresponding responses are generated. To ensure the quality of training instances, DAPO employ a dynamic sampling strategy that filters out samples providing no learning signal. Following this, to form a complete batch of 16 qualified samples, the trainer could perform up to 48 sampling attempts.

Furthermore, an overlong response punishment mechanism is implemented to discourage excessively long generations. Responses exceeding 1024 tokens are penalized, with the penalty value increasing linearly from 0 (at 1024 tokens) to a maximum of 1 (at 4096 tokens).

D.2 TRAINING TIME AND RESOURCE CONSUMPTION

We trained the Tool-use Refiner (for our main experiments) on four NVIDIA RTX A5000 GPUs. The entire training process lasted for 6 hours and occupied 22GB of VRAM on each GPU.

D.3 TRAINING RESULTS

Figure 2 illustrates the training dynamics of the Tool-use Refiner used in our main experiments, plotting the changes in reward and average response length.

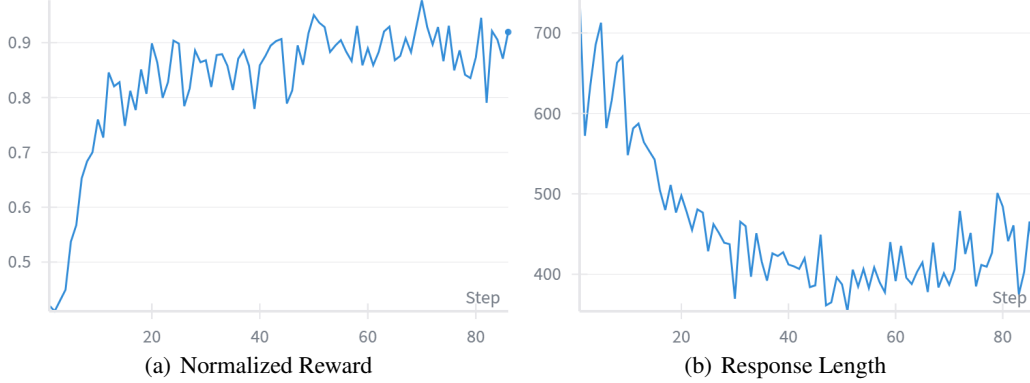


Figure 2: Training dynamics of the Tool-use Refiner.

As shown in Figure 2(a), the reward score exhibits a consistent upward trend, rising from 0.4 and stabilizing at approximately 0.9. This serves as strong evidence that our proposed reward function successfully guides the Refiner to improve its proficiency in correcting tool invocations.

Furthermore, Figure 2(b) highlights a substantial reduction in output verbosity. The average response length, inclusive of the reasoning tokens, drops from 741 to around 400. This trend towards more succinct outputs is crucial as it mitigates the risk of exceeding the context window limit.

E EVALUATION SETUP

E.1 EVALUATION SETUP FOR REFINING GENERAL-PURPOSE TOOL INVOCATION

E.1.1 API-BANK BENCHMARK SPECIFICATION

API-Bank is a tool invocation benchmark featuring three levels of difficulty: Level 1 (tool name is apparent in the dialogue), Level 2 (tool name requires retrieval from an available tool list), and Level 3 (sustained tool calls are required).

For our evaluation, we utilize the API-bank dataset provided by the official ToolRL repository. This dataset has been pre-processed into a multi-turn dialogue format. In this setup, the model’s objective is to select the appropriate tools to address the user’s final request at the conclusion of the dialogue. The list of available tools is provided to the model within the system prompt. The difficulty of each task is stratified based on two factors: (1) the number of tools available in the prompt, and (2) the presence of hints or cues provided to the model during the conversational turns. We measure performance using exact match accuracy, which requires the model’s predicted tool invocations to be identical to the ground truth.

E.1.2 BASELINES FOR COMPARISON

In this subsection, we detail the training procedures for the various baselines compared in our evaluation on API-bank, following the experimental setup proposed in the ToolRL (Qian et al., 2025).

First, a balanced training dataset is curated to ensure exposure to a wide range of tasks. Specifically, 2,000 samples are extracted from the ToolACE (Liu et al., 2024) dataset, and 1,000 samples are drawn from each of the Hammer (Lin et al., 2024) and xLAM (Zhang et al., 2024) datasets. This combination yields a total of 4,000 training instances, creating a balanced dataset that spans diverse levels of complexity and tool use.

The methods listed in Table 1 represent different stages and strategies of model training. The **Raw** method serves as our baseline, representing the original model’s performance without any task-specific fine-tuning or reinforcement learning (RL). The **SFT** model is an instruction-tuned model that has been further fine-tuned on 400 data points randomly sampled from the aforementioned training data. Building upon this, the **SFT+PPO** and **SFT+GRPO** methods take the SFT model as a starting point and apply the PPO or GRPO RL algorithms, respectively, using the entire training dataset. In contrast, the **PPO Cold Start** method bypasses the SFT step, applying the PPO algorithm directly to the original model. Finally, the **ToolRL** method involves training the original model from scratch with the GRPO algorithm on all available training data.

E.1.3 ADAPTING THE REFINER FOR THE API-BANK BENCHMARK

The API-Bank benchmark presents a unique formatting requirement where models must simultaneously generate reasoning (<think>), tool invocations (<tool_call>), and a final response (<response>). This multi-part structure is inconsistent with our Refiner’s original training objective, creating ambiguity regarding the placement of the final response. To address this, we re-task the Refiner to focus exclusively on correcting tool calls. We provide it with a specific instructional prompt:

As a tool-calling instruction corrector, you do not need to output the information related to “;response;”. Please correct the instructions from another model and output only the correct tool-calling instructions. Your output must strictly follow this format: ‘<tool_call>{“name”: “Tool name”, “parameters”: {“Parameter name”: “Parameter content”, “... ..”: “... ..”}}{“name”: “... ..”, “parameters”: {“... ..”: “... ..”, “... ..”: “... ..”}}...< /tool_call>’. If the original instructions from another LLM are correct, do not make any modifications and directly output these instructions. If you determine that additional instructions are necessary, append them directly after the original instructions from another LLM.

Furthermore, we identify a logic error in the benchmark’s ground truth: for certain samples, the task requires the model to generate parameters of int or float types, whereas the ground-truth parameters are formatted as string type. This inconsistency leads to an inaccurate assessment, so we have rectified this issue in our evaluation. We add an instruction to align the Refiner’s output with the ground truth format: “For any numeric parameters you generate (including int and float types), please enclose them in quotes to output them as strings.” These adjustments align the Refiner’s behavior with the benchmark’s specific requirements, ensuring a more accurate evaluation.

E.2 EVALUATION SETUP FOR REFINING SEQUENTIAL ERRORS

Our evaluation is conducted on the NESTFUL benchmark (Basu et al., 2024), which is designed to assess the ability to execute a logical sequence of tool invocations for multi-step problem-solving. A core challenge within this benchmark is the management of intermediate results: models are required to store the output of a tool call in a temporary variable (e.g., var_0) and subsequently use this variable as an argument in a later tool call. This process tests the model’s capacity for multi-step reasoning and planning, mirroring complex tasks that involve sequential manipulation, such as chained mathematical calculations.

Following the NESTFUL paper, we employ a standardized prompt format: “**SYSTEM:** You are a helpful assistant with access to the following function calls. ... <[function_call_library]>{**TOOL_LIBRARY**} Here are some examples:{**ICL_EXAMPLES**} **USER:** {**QUERY**} **ASSISTANT:**”.

Each prompt begins with a system message defining the model’s role as an assistant with access to a specific function library, which is provided under the {**TOOL_LIBRARY**} placeholder. Consistent with the NESTFUL paper, we include three in-context learning examples, denoted by {**ICL_EXAMPLES**}, to guide the model’s response generation (Dong et al., 2022). The prompt concludes with the specific user task, presented as the {**QUERY**}. The model is then expected to generate the response which contains the sequence of tool calls required to fulfill the user’s request.

The model’s output is a JSON-formatted list of tool calls, where each call specifies the tool name, its arguments, and a critical label parameter (e.g., \$var_1) that assigns the tool’s return value to a variable. Our evaluation protocol rigorously verifies the correctness of this entire execution chain.

We sequentially execute each tool call generated by the model, computing the value of each intermediate variable in the specified order. The final computed result is then compared against the ground truth answer for the sample. Performance is measured by the Win Rate, which is the percentage of tasks where the model’s final result exactly matches the ground truth.

E.3 EVALUATION SETUP FOR REFINING SEARCH TOOL INVOCATIONS

Following the Search-R1 paper, we first deploy a search engine that utilizes the 2018 Wikipedia dump (Karpukhin et al., 2020) as the knowledge corpus and E5 (Wang et al., 2022) as the retriever. During inference, the model can iteratively issue a “<search>” tag to query the corpus when it deems more information is necessary. In response, the search engine returns the top-3 most relevant documents. This process repeats until the model has sufficient information to generate a final response, which is then enclosed within an “<answer>” tag.

All tasks are structured using the following prompt format: *Answer the given question. You must conduct reasoning inside <think> and </think> first every time you get new information. After reasoning, if you find you lack some knowledge, you can call a search engine by <search> query </search>, and it will return the top searched results between <information> and </information>. You can search as many times as you want. If you find no further external knowledge needed, you can directly provide the answer inside <answer> and </answer> without detailed illustrations. For example, <answer> xxx </answer>. Question: question.*

Building on the paradigm of the Search-R1 paper, we implement an evaluation framework that facilitates multi-turn retrieval, with a maximum of three turns allowed per sample. Our Tool-use Refiner is activated when the upstream LLM fails to produce a conclusive “<answer>” tag, either by generating a new “<search>” tag or by exceeding the context window limit: (1) If the upstream LLM generates a “<search>” tag, the Refiner processes the full generation history and the current output to produce a revised search query. (2) If the upstream LLM exceeds the context limit, the Refiner is given the history concatenated with the first 500 tokens of the current output to generate an appropriate search query.

E.4 EVALUATION SETUP FOR QUANTITATIVE CORRECTION ANALYSIS

This evaluation is conducted on the Bamboogle benchmark, which comprises 17 distinct retrieval tasks. A key characteristic of this benchmark is that all queries revolve around retrieving temporal or geographical information based on the birth dates or birthplaces of celebrities. Leveraging this structure, we reformulate the retrieval tasks into a tool-calling framework. This involves converting each of the 17 retrieval tasks into a distinct API. Additionally, we create foundational APIs to first retrieve a celebrity’s birth date or birthplace. Consequently, the model’s task is transformed from direct question-answering to generating the correct sequence of API calls to solve the problem.

For instance, a foundational API to retrieve a celebrity’s birthplace is defined as: {“name”: “find_birthplace”, “description”: “Returns the birthplace information of a celebrity.”, “parameters”: {“celebrity_name”: {“type”: “string”, “description”: “One of the input parameters: the name of the celebrity.”}}, “output_name”: “output_birthplace”}.

An example of a secondary tool that requires the output of the above api is find_capital: {“name”: “find_capital”, “description”: “Returns a country’s capital.”, “parameters”: {“information_type”: {“type”: “string”, “description”: “One of the input parameters, applicable information for search, limited to either ‘birthplace’, ‘birthdate’ or ‘birthyear’.”}, “information_content”: {“type”: “string”, “description”: “Stores the specific content provided to the tool (can use codes starting with ‘output_’).”}}, “output_name”: “output_capital”}.

Furthermore, to assess the model’s ability to adhere to specific formatting instructions, we introduce three distinct tool invocation formats. For each sample, the model is randomly prompted to generate its response in one of the following formats:

- <tool_call> [{“name”: “tool_name1”, “arguments”: {“parameter_name1”: “parameter_content1”, “parameter_name2”: “parameter_content2”, ...}}, {“name”: “tool_name2”, “arguments”: {“parameter_name3”: “parameter_content3”, ...}}, ...] </tool_call>.

- $[func1(params_name=params_value, params_name2=params_value2...), func2(params)]$.
- $\{ \{ "type": "tool_use", "name": "tool_name1", "input": \{ "parameter_name1": "parameter_content1", "parameter_name2": "parameter_content2", ... \} \}, \{ "type": "tool_use", "name": "tool_name2", "input": \{ "parameter_name3": "parameter_content3", ... \} \}, ... \}$.

This controlled setup, where the required format, tool names, and parameter contents are all programmatically parsable from the model’s output, enables a quantitative, multi-faceted assessment of a model’s ability to generate syntactically valid and semantically correct tool invocations under diverse constraints. Moreover, this framework provides an ideal testbed for evaluating the correction capabilities of our Tool-use Refiner.

F PERFORMANCE COMPARISON WITH AN UNTRAINED REFINER

In this section, we present an ablation study to isolate the contribution of our RL training process. We evaluate the performance of upstream models when using the original Qwen3-1.7B as a tool-use corrector. This allows us to disentangle the improvements stemming from our RL training from the pre-existing capabilities of Qwen3-1.7B itself.

Following the protocol of our main evaluations, we quantitatively evaluate the corrected tool invocations. Specifically, we use the modified Bamboogle benchmark to assess accuracy across three metrics: Acc_{Full} , $Acc_{Toolname}$, $Acc_{Parameter}$. Furthermore, we employ the NESTFUL benchmark to evaluate the sequential accuracy (with CoT). The detailed results are presented in Table 5.

Table 5: Performance comparison with an untrained Refiner (w/ Raw Refiner)

Model	Bamboogle*			NESTFUL		
	Acc.	$Acc_{Toolname}$	$Acc_{Parameter}$	Part. Acc.	Full Acc.	Win Rate
<i>Qwen2.5-3B-Instruct</i>						
CoT	0.00	54.27	52.27	0.10	0.04	0.13
w/ Raw Refiner	14.13	56.33	55.53	0.15	0.08	0.22
Ours, w/ Refiner	48.13	79.67	79.47	0.19	0.10	0.23
<i>Qwen2.5-7B-Instruct</i>						
CoT	20.87	60.40	60.40	0.25	0.18	0.18
w/ Raw Refiner	8.67	57.53	57.47	0.24	0.19	0.27
Ours, w/ Refiner	49.13	79.47	79.47	0.28	0.22	0.31
<i>Llama-3.2-3B-Instruct</i>						
CoT	1.60	54.20	45.70	0.11	0.03	0.04
w/ Raw Refiner	22.13	53.13	50.40	0.17	0.09	0.12
Ours, w/ Refiner	36.53	74.73	73.80	0.21	0.12	0.17
<i>Llama-3.1-8B-Instruct</i>						
CoT	43.07	64.80	60.67	0.21	0.15	0.09
w/ Raw Refiner	31.40	59.47	58.67	0.22	0.16	0.21
Ours, w/ Refiner	60.27	72.20	72.00	0.27	0.20	0.25
<i>Qwen3-1.7B</i>						
CoT	16.70	62.27	59.40	0.15	0.12	0.20
w/ Raw Refiner	22.47	64.40	62.73	0.15	0.11	0.24
Ours, w/ Refiner	39.67	68.60	66.80	0.22	0.15	0.31
<i>Qwen3-8B</i>						
CoT	80.33	86.87	86.80	0.25	0.21	0.33
w/ Raw Refiner	56.53	81.80	81.67	0.25	0.20	0.36
Ours, w/ Refiner	86.60	87.13	87.13	0.29	0.22	0.41

As demonstrated in Table 5, our trained Tool-use Refiner comprehensively outperforms the original Qwen3-1.7B on both the Bamboogle* and NESTFUL benchmarks. The superior performance is evident across all evaluated aspects, including the correction of tool invocation format, selection, parameters, and order. On certain metrics, the original Qwen3-1.7B even leads to a decrease in scores. This result indicates that the strong performance of our refiner in the main evaluation stems from our specialized RL training for the tool invocation correction task, rather than the inherent capabilities of the base model, validating the effectiveness of our proposed training methodology.

G AN ABLATION STUDY OF THE OVERLONG PENALTY

To validate the effectiveness of the overlong penalty, we conduct an ablation study by removing it from the training objective. Our hypothesis is that without this penalty, the Refiner would be more prone to generating outputs that exceed the context window, impairing its performance. We evaluate both our full Refiner (w/ Refiner) and the ablated Refiner (w/o Overlong Penalty) on the modified Bamboogle dataset (Bamboogle*) and the API-Bank benchmark.

Table 6: Ablation study on the overlong penalty (w/o Overlong Penalty)

Model	Bamboogle*			API-bank
	Acc.	Acc _{Toolname}	Acc _{Parameter}	
<i>Llama-3.2-3B-Instruct</i>				
CoT	1.60	54.20	45.70	39.03
w/o Overlong Penalty	35.13	73.60	70.67	55.28
Ours, w/ Refiner	36.53	74.73	73.8	64.15
<i>Llama-3.1-8B-Instruct</i>				
CoT	43.07	64.80	60.67	67.17
w/o Overlong Penalty	63.80	68.13	68.00	57.45
Ours, w/ Refiner	60.27	72.20	72.00	68.51
<i>Qwen3-1.7B</i>				
CoT	16.70	62.27	59.40	60.13
w/o Overlong Penalty	38.67	70.66	69.00	60.30
Ours, w/ Refiner	39.67	68.60	66.80	70.18
<i>Qwen3-8B</i>				
CoT	80.33	86.87	86.80	70.85
w/o Overlong Penalty	86.60	87.33	87.33	60.47
Ours, w/ Refiner	86.60	87.13	87.13	74.37

The results, summarized in Table 6, confirm our hypothesis. On the API-Bank benchmark, the full Refiner significantly outperforms the ablated version. Error analysis further reveals that the ablated model exhibits a high incidence of context window violations, with overlong outputs occurring in 13.5% of cases on average. This finding underscores the crucial role of the overlong penalty. In contrast, on the Bamboogle* dataset, both models deliver comparable performance, with no instances of overlong generation observed.

This performance difference can be attributed to the complexity and diversity of the tasks in each benchmark. API-Bank, as a general-purpose tool-use dataset, features a wide array of distinct tools, often demanding intricate and lengthy reasoning chains. The ablated model, lacking the length constraint, tends to over-analyze, leading to context window breaches. Bamboogle*, however, is constrained to 17 similar sub-tasks (see Section E.4) focused on generating search queries from celebrity data. The similarity of these tasks necessitates a less complex reasoning process, which remains well within the context limits for both models.

Ultimately, this study demonstrates that the overlong penalty is a critical component for ensuring that the Refiner produces concise yet effective reasoning, particularly in complex scenarios. It effectively mitigates the risk of generating excessively long outputs that break the context window, confirming its value in our training methodology.

H AN ABLATION STUDY OF THE SIMPLIFIED CALL SEQUENCE DATA

The introduction of the simplified call sequence data serves three primary strategic purposes in our training methodology. First, it functions as a form of curriculum learning, reducing the initial task complexity by decoupling sequence ordering from parameter generation. This allows the model to master the foundational logic of tool dependency first, ensuring a more stable and efficient training convergence. Second, it enhances model generalization. By introducing a novel structure (i.e., a sequence of steps, each potentially containing concurrent calls), we intentionally diverge from standard flat-list formats. This structural variance compels the model to learn the abstract principles of tool coordination rather than overfitting to a specific data schema. Finally, and most critically, this format enables targeted, high-quality negative sampling. By deliberately placing tools with sequential dependencies into a single concurrent step, we create explicit, instructive error cases. This

efficiently teaches the model the crucial distinction between parallelizable and sequential operations, a fundamental aspect of complex tool use.

In this section, we conduct an ablation study on the simplified call sequence data, investigating the performance change of the Refiner when this data is excluded from the training process. We perform evaluations on the modified Bamboogle dataset (Bamboogle*), API-bank, and NESTFUL benchmarks. The results are presented in Table 7.

Table 7: Ablation study on the simplified call sequence data (w/o Simplified Order Data)

Model	NESTFUL			API-bank	Bamboogle*		
	Part. Acc.	Full Acc.	Win Rate		Acc.	Acc _{Toolname}	Acc _{Parameter}
<i>Qwen2.5-3B-Instruct</i>							
CoT	0.10	0.04	0.13	40.54	0.00	54.27	52.27
w/ Refiner (No simplified order data)	0.18	0.09	0.22	65.83	40.86	72.93	72.60
Ours, w/ Refiner	0.19	0.10	0.23	68.84	48.13	79.67	79.47
<i>Qwen2.5-7B-Instruct</i>							
CoT	0.25	0.18	0.18	68.34	20.87	60.40	60.40
w/ Refiner (No simplified order data)	0.26	0.20	0.28	71.36	21.6	59.80	59.80
Ours, w/ Refiner	0.28	0.22	0.31	73.70	49.13	79.47	79.47
<i>Llama-3.2-3B-Instruct</i>							
CoT	0.11	0.03	0.04	39.03	1.60	54.20	45.70
w/ Refiner (No simplified order data)	0.19	0.11	0.14	63.17	26.73	59.67	58.00
Ours, w/ Refiner	0.21	0.12	0.17	64.15	36.53	74.73	73.80
<i>Llama-3.1-8B-Instruct</i>							
CoT	0.21	0.15	0.09	67.17	43.07	64.80	60.67
w/ Refiner (No simplified order data)	0.28	0.20	0.28	70.18	58.73	67.00	66.73
Ours, w/ Refiner	0.27	0.20	0.25	68.51	60.27	72.20	72.00
<i>Qwen3-1.7B</i>							
CoT	0.15	0.12	0.20	60.13	16.70	62.27	59.40
w/ Refiner (No simplified order data)	0.19	0.15	0.25	73.70	42.53	72.27	70.33
Ours, w/ Refiner	0.22	0.15	0.31	70.18	39.67	68.60	66.80
<i>Qwen3-8B</i>							
CoT	0.25	0.21	0.33	70.85	80.33	86.87	86.80
w/ Refiner (No simplified order data)	0.26	0.21	0.35	73.87	83.93	86.40	86.40
Ours, w/ Refiner	0.29	0.22	0.41	74.37	86.60	87.13	87.13

As shown in Table 7, on the NESTFUL benchmark, which specifically evaluates tool sequencing, our full Refiner slightly outperforms the version trained without simplified call sequence data. This confirms that the introduction of simplified call sequence data effectively enhances the Refiner’s ability to correct tool invocation sequences.

Furthermore, the performance gap between the two versions becomes more pronounced on benchmarks that use conventional tool-calling formats (Bamboogle*, API-bank). On these tasks, our full model demonstrates significantly better performance. In contrast, the ablated Refiner not only yields lower scores across the board but, when paired with certain upstream LLMs, its performance is merely on par with the CoT baseline, indicating a minimal improvement. This degradation is particularly evident in scenarios where the upstream LLM produces convoluted or ambiguous reasoning chains. The ablated model’s inability to identify these failures suggests a lack of stability and poorer generalization.

In conclusion, training with simplified call sequence data leads to a more stable Refiner, enhancing its proficiency in both general tool-use correction and specific tool order correction.

I THE USE OF LARGE LANGUAGE MODELS (LLMs)

During the preparation of this manuscript, we utilize the Large Language Model (LLM) Gemini 2.5 Pro (Comanici et al., 2025) for language refinement and polishing.

J CASE STUDIES

J.1 CORRECTION OF FORMATTING ERRORS

The following case demonstrates our Refiner’s proficiency in fixing formatting errors. Here, the Refiner first validates the semantic correctness of the tool invocation but then pinpoint a formatting issue. The error is then resolved through a precise modification targeted solely at the structure.

The Original System Prompt: You are a helpful multi-turn dialogue assistant capable of leveraging tool calls to solve user tasks and provide structured chat responses.
Available Tools: In your response, you can use the following tools:
 1. **GetOccupationSalary:** API for querying the salary of a given occupation.
 Parameters: {‘occupation’: {‘type’: ‘str’, ‘description’: ‘The occupation to query.’}}
 Steps for Each Turn
 1. **Think:** Recall relevant context and analyze the current user goal.
 2. **Decide on Tool Usage:** If a tool is needed, specify the tool and its parameters.
 3. **Respond Appropriately:** If a response is needed, generate one while maintaining consistency across user queries.
Output Format:
 plaintext
 <think> Your thoughts and reasoning </think>
 <tool_call> {‘name’: ‘Tool name’, ‘parameters’: {‘Parameter name’: ‘Parameter content’, ‘...’: ‘...’}}
 <name’: ‘...’, ‘parameters’: {‘...’: ‘...’, ‘...’: ‘...’}}
 <name’: ‘...’, ‘parameters’: {‘...’: ‘...’, ‘...’: ‘...’}}
 <tool_call> <response> AI’s final response </response>
Important Notes:
 1. You must always include the ‘<think>’ field to outline your reasoning. Provide at least one of ‘<tool_call>’ or ‘<response>’. Decide whether to use ‘<tool_call>’ (possibly multiple times), ‘<response>’, or both.
 2. You can invoke multiple tool calls simultaneously in the ‘<tool_call>’ fields. Each tool call should be a JSON object with a ‘name’ field and an ‘parameters’ field containing a dictionary of parameters. If no parameters are needed, leave the ‘parameters’ field an empty dictionary.
 3. Refer to the previous dialogue records in the history, including the user’s queries, previous ‘<tool_call>’, ‘<response>’, and any tool feedback noted as ‘<obs>’ (if exists).

The User Query: Dialogue Records History
 <user> Calculate the monthly salary after taxes for a Data Scientist
 </user>
 Based on our conversation above, please only make one tool call to solve my need.
 </user>

Ground Truth: <tool_call> {‘name’: ‘GetOccupationSalary’, ‘parameters’: {‘occupation’: ‘Data Scientist’}}
 </tool_call>

Upstream LLM’s Thought: The user wants to know the monthly salary of a Data Scientist after taxes. We need to use the GetOccupationSalary tool to fetch the salary information for a Data Scientist and then calculate the salary after taxes. However, the tool does not provide tax information, so we will assume a standard tax rate for the calculation.

Upstream LLM’s Answer: <tool_call> {‘name’: ‘GetOccupationSalary’, ‘parameters’: {‘occupation’: ‘Data Scientist’}}
 <response> AI will fetch the salary information for a Data Scientist and then calculate the salary after taxes. Please wait.
 </response>
 </tool_call>

Refiner’s Thought: Okay, let’s see. The user wants to calculate the monthly salary after taxes for a Data Scientist. The available tool is GetOccupationSalary, which queries the salary of a given occupation. The user’s query is about calculating the monthly salary after taxes, so the tool call should be to GetOccupationSalary with the occupation ‘Data Scientist’. The parameters are correct as the occupation is provided. The user also mentioned that only one tool call is needed. The original instruction from the model is correct, so no corrections are needed. The output should include the tool call with the parameters. The response is not needed here because the user wants only the tool call. So the correct instruction is the one provided by the model.

Refiner’s Answer: <tool_call> {‘name’: ‘GetOccupationSalary’, ‘parameters’: {‘occupation’: ‘Data Scientist’}}
 </tool_call>

Figure 3: Correction of formatting errors.

J.2 PRESERVING A CORRECT TOOL INVOCATION

The following case shows our Refiner’s response to a correct tool invocation from the upstream LLM. The output shows that our Refiner correctly recognizes the tool invocation as valid and preserves it without modification.

The Original System Prompt: You are a helpful multi-turn dialogue assistant capable of leveraging tool calls to solve user tasks and provide structured chat responses.\n\n**Available Tools**\n\nIn your response, you can use the following tools:\n1. Name: QueryHealthData\nDescription: This API queries the recorded health data in database of a given user and time span.\nParameters: {'user_id': {'type': 'str', 'description': 'The user id of the given user. Cases are ignored.'}, 'start_time': {'type': 'str', 'description': 'The start time of the time span. Format: %Y-%m-%d %H:%M:%S'}, 'end_time': {'type': 'str', 'description': 'The end time of the time span. Format: %Y-%m-%d %H:%M:%S'}}\n2. Name: CancelRegistration\nDescription: This API cancels the registration of a patient given appointment ID.\nParameters: {'appointment_id': {'type': 'str', 'description': 'The ID of appointment.'}}\n3. Name: ModifyRegistration\nDescription: This API modifies the registration of a patient given appointment ID.\nParameters: {'appointment_id': {'type': 'str', 'description': 'The ID of appointment.'}, 'new_appointment_date': {'type': 'str', 'description': 'The new appointment date. Format: %Y-%m-%d.'}, 'new_appointment_doctor': {'type': 'str', 'description': 'The new appointment doctor.'}}\n\n**Steps for Each Turn**\n1. **Think:** Recall relevant context and analyze the current user goal.\n2. **Decide on Tool Usage:** If a tool is needed, specify the tool and its parameters.\n3. **Respond Appropriately:** If a response is needed, generate one while maintaining consistency across user queries.\n\n**Output Format**\n\nplaintext\n<think> Your thoughts and reasoning\n</think>\n<tool_call>\n{\n "name": "Tool name",\n "parameters": {\n "Parameter name": "Parameter content",\n "...": "..."\n }\n}\n</tool_call>\n<response> AI's final response\n</response>\n\n**Important Notes**\n1. You must always include the <think> field to outline your reasoning. Provide at least one of <tool_call> or <response>. Decide whether to use <tool_call> (possibly multiple times), <response>, or both.\n2. You can invoke multiple tool calls simultaneously in the <tool_call> fields. Each tool call should be a JSON object with a <name> field and an <parameters> field containing a dictionary of parameters. If no parameters are needed, leave the <parameters> field an empty dictionary.\n3. Refer to the previous dialogue records in the history, including the user's queries, previous <tool_call>, <response>, and any tool feedback noted as <obs> (if exists).

The User Query: **Dialogue Records History**\n<user>Can you please modify my appointment scheduled for March 25th with Dr. Kim to March 26th with Dr. Lee?\n</user>\n<response>Sure, I can help you with that. Please provide me with the appointment ID and the new appointment date and doctor's name.\n</response>\n<user>The appointment ID is 34567890 and the new date is March 26th with Dr. Lee.\n</user>\n<response>Alright. I'll modify your appointment now.\n</response>\n<user> Based on our conversation above, please only make one tool call to solve my need.\n</user>

Ground Truth: <tool_call>\n{\n "name": "ModifyRegistration",\n "parameters": {\n "appointment_id": "34567890",\n "new_appointment_date": "2023-03-26",\n "new_appointment_doctor": "Dr. Lee"\n }\n}\n</tool_call>

Upstream LLM's Thought: The user needs their appointment modified. They have provided the appointment ID, the new appointment date, and the new doctor's name. I need to use the ModifyRegistration API to perform this action.

Upstream LLM's Answer: <tool_call>\n{\n "name": "ModifyRegistration",\n "parameters": {\n "appointment_id": "34567890",\n "new_appointment_date": "2023-03-26",\n "new_appointment_doctor": "Dr. Lee"\n }\n}\n</tool_call>

Refiner's Thought: Okay, let's see. The user wants to modify their appointment from March 25th with Dr. Kim to March 26th with Dr. Lee. The original tool-calling instruction uses "ModifyRegistration" with parameters "appointment_id": "34567890", "new_appointment_date": "2023-03-26", "new_appointment_doctor": "Dr. Lee".\n\nWait, the user's original query says "March 26th with Dr. Lee". The parameter for the new appointment date is "new_appointment_date" and the doctor's name is "new_appointment_doctor". The user provided the appointment ID as 34567890, which is correct. The date format is "2023-03-26" which is in the correct format. The doctor's name is "Dr. Lee" which is correct.\n\nBut wait, the user wrote "March 26th with Dr. Lee" which is a bit ambiguous. The date is March 26th, so the format should be "2023-03-26" and the doctor's name is "Dr. Lee". The parameters in the tool-calling instruction are correct. The only thing I notice is that the user might have intended the date to be in the format "2023-03-26" instead of "2023-03-26" (without the hyphen). But the user's input is already in the correct format. So the tool-calling instruction is correct. The user might have made a typo, but the parameters are correct. Therefore, the tool-calling instruction is correct and no corrections are needed.

Refiner's Answer: <tool_call>\n{\n "name": "ModifyRegistration",\n "parameters": {\n "appointment_id": "34567890",\n "new_appointment_date": "2023-03-26",\n "new_appointment_doctor": "Dr. Lee"\n }\n}\n</tool_call>

Figure 4: Preserving a correct tool invocation.

J.4 CORRECTION OF SEQUENTIAL ERRORS.

The following case demonstrates our Refiner’s ability to rectify flawed tool invocation sequences. Specifically, it identifies and corrects both parameter and sequential errors in the output from the upstream LLM.

The Original System Prompt: 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 function 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 call in tools call sections. If you decide to invoke any of the function(s), you MUST put it in the format of `<think> Your thoughts and reasoning </think>` `{\{"name": "\func_name1", "arguments": {\{"argument1": "\value1", "argument2": "\value2"}, \label": "\$var_1\"}}, ... (more tool calls as required)}` You SHOULD NOT include any other text in the response. Here is a list of functions in JSON format that you can invoke. `{FUNCTION_STR}` Here are some examples: `{ICL_EXAMPLES}`.

The User Query: The average age of 15 students of a class is 15 years. Out of these, the average age of 4 students is 14 years and that of the other 10 students is 16 years. The age of the 15th student is

Ground Truth: `{\{"name": "multiply", "arguments": {\{"arg_0": 4, "arg_1": 14}, "label": "\$var_1"}, {"name": "multiply", "arguments": {\{"arg_0": 10, "arg_1": 16}, "label": "\$var_2"}, {"name": "add", "arguments": {\{"arg_0": "\$var_1.result", "arg_1": "\$var_2.result"}, "label": "\$var_3"}, {"name": "multiply", "arguments": {\{"arg_0": 15, "arg_1": 15}, "label": "\$var_4"}, {"name": "subtract", "arguments": {\{"arg_0": "\$var_4.result", "arg_1": "\$var_3.result"}, "label": "\$var_5"}}`

Upstream LLM’s Thought: To find the age of the 15th student, we need to first find the total age of all 15 students, then subtract the total age of the first 14 students. We can find the total age of the first 14 students by finding the total age of the 4 students with an average age of 14 and the total age of the 10 students with an average age of 16. We can use the ‘multiply’ function to find the total age of each group and then add them together. We can use the ‘add’ function to find the total age of all 15 students. We can then use the ‘subtract’ function to find the age of the 15th student.

Upstream LLM’s Answer: `{\{"name": "multiply", "arguments": {\{"arg_0": 15, "arg_1": 15}, "label": "\$var_1"}, {"name": "multiply", "arguments": {\{"arg_0": 4, "arg_1": 14}, "label": "\$var_2"}, {"name": "subtract", "arguments": {\{"arg_0": "\$var_1.result", "arg_1": "\$var_2.result"}, "label": "\$var_3"}, {"name": "multiply", "arguments": {\{"arg_0": 4, "arg_1": "\$var_2.result"}, "label": "\$var_4"}, {"name": "multiply", "arguments": {\{"arg_0": 10, "arg_1": 16}, "label": "\$var_5"}, {"name": "add", "arguments": {\{"arg_0": "\$var_4.result", "arg_1": "\$var_5.result"}, "label": "\$var_6"}, {"name": "subtract", "arguments": {\{"arg_0": "\$var_1.result", "arg_1": "\$var_6.result"}, "label": "\$var_7"}}`

Refiner’s Thought: Okay, let’s see. The user is asking about the age of the 15th student in a class. The average age of 15 students is 15 years. Then, the average age of 4 students is 14, and the other 10 students is 16. So, the total age for the 4 students is $4 \times 14 = 56$, and for the 10 students is $10 \times 16 = 160$. The total age for all 15 students is $15 \times 15 = 225$. The 15th student’s age is the total age minus the sum of the other 14 students. Wait, but the user says the average age of 15 students is 15, so total age is $15 \times 15 = 225$. Then, the 4 students have 56 and 10 students have 160. So, the total is $56 + 160 = 216$. Then, the 15th student’s age is $225 - 216 = 9$. So, the answer should be 9. The tool-calling instruction has some errors. The multiply functions are called with `arg_0` and `arg_1`, but the parameters are not correct. The user might have intended to call the add function with the sum of the 4 and 10 students’ ages. So, the correct approach is to call multiply for 4 and 14, multiply for 10 and 16, add them, then subtract from the total. The original instruction has the multiply function called with the result of the multiply functions, but the parameters are incorrect. The correct instruction should call multiply, add, and subtract in the right order.

Refiner’s Answer: `{\{"name": "multiply", "arguments": {\{"arg_0": 4, "arg_1": 14}, "label": "\$var_1"}, {"name": "multiply", "arguments": {\{"arg_0": 10, "arg_1": 16}, "label": "\$var_2"}, {"name": "add", "arguments": {\{"arg_0": "\$var_1.result", "arg_1": "\$var_2.result"}, "label": "\$var_3"}, {"name": "multiply", "arguments": {\{"arg_0": 15, "arg_1": 15}, "label": "\$var_4"}, {"name": "subtract", "arguments": {\{"arg_0": "\$var_4.result", "arg_1": "\$var_3.result"}, "label": "\$var_5"}}`

Figure 6: Correction of sequential errors.