Invocation Refiner: A Plug-and-Play Module for Rectifying LLM Tool Invocations

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have shown remarkable capabilities in Tool-Integrated Reasoning (TIR). However, the practical application is often hindered by frequent errors in tool invocations, such as incorrect parameters or malformed formats. Prevailing training paradigms, such as Supervised Fine-Tuning (SFT) and Reinforcement Learning (RL), can mitigate these issues but require modification on the base LLM. This lack of modularity necessitates extensive retraining when deploying the system across different base models. To address the limitation, we introduce the Invocation Refiner, a specialized post-processing module designed to enhance the tool-use reliability of base LLMs without directly training on them. The Refiner takes the output from a frozen upstream LLM and the user's query as input, performing independent reasoning to rectify the invocation. We construct a dedicated training dataset and train this module using an advanced RL algorithm. 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 plug-and-play solution for improving the operational reliability of LLM-based agents. We release our code 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.

Existing approaches to mitigate these errors primarily rely on Supervised Fine-Tuning (SFT) or Reinforcement Learning (RL) directly applied to the base LLM (Zeng et al., 2023; Feng et al., 2025). While effective, these methods suffer from a lack of modularity: they necessitate altering the parameters of the base model itself. As a result, when deploying the system across different base models, extensive retraining is required. Given the limitation, a compelling research question arises: *Is it possible to decouple tool invocation correction from the upstream generation process, enabling a flexible, model-agnostic enhancement of TIR capabilities?*

Inspired by self-refinement and model cascading techniques (Ji et al., 2024; Ngweta et al., 2024), we introduce the **Invocation Refiner**, a specialized post-processing module designed to rectify tool calls generated by arbitrary **upstream LLMs** (a term we use to denote the base LLMs responsible for initial reasoning). Unlike traditional alignment methods that directly fine-tune the upstream LLM, our approach treats it as a frozen generator. When a user submits a task, the upstream LLM produces an initial tool invocation. The Refiner then processes it alongside the original context, performing targeted reasoning to identify and correct mistakes in format, parameters, or logic.

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, format correction, and sequence ordering, demonstrating that the refined outputs lead to comprehensive performance improvements over the raw outputs from the upstream model. Moreover, the Invocation Refiner can be seamlessly attached to various upstream LLMs, offering plug-and-play versatility. By decoupling correction from generation, the Invocation Refiner offers a versatile, deployable solution for building robust LLM-based agents.

The main contributions of this work are summarized as follows:

- We introduce a specialized module for plug-and-play refinement, the **Invocation Refiner**, which corrects and standardizes tool invocations generated by diverse upstream LLMs while keeping the upstream LLMs frozen.
- We construct a dedicated training framework tailored for invocation correction and evaluate the efficacy of **SFT**, **DAPO**, and **SFT Cold Start + DAPO** in bolstering the Refiner's generalization across diverse error types.
- We demonstrate the versatility of our approach through extensive experiments, showing that the Invocation Refiner effectively rectifies errors (e.g., formatting, sequencing) across various upstream LLMs, consistently outperforming baselines and verifying its efficacy as a universal enhancement module.

This work highlights the potential of a specialized, decoupled module to rectify errors from upstream LLMs in tool-integrated scenarios, offering a versatile 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 (TIR). TIR empowers LLMs to perform reasoning related to tool usage and output tool invocations to execute diverse downstream tasks. 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) allow LLMs to execute code for task completion. MAmmoTH (Yue et al., 2023) and MathCoder (Wang et al., 2023) leverage multiple tools to solve complex mathematical problems. xLAM (Zhang et al., 2024) and ToolACE (Liu et al., 2024) employ LLMs as central controllers to orchestrate tool calls, facilitating the construction of AI agent systems.

Self-Refinement. LLMs often struggle to generate flawless outputs 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 dialogue generation and mathematical reasoning. Idea2Img (Yang et al., 2024) uses 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 outcomes and the code's logic. Furthermore, self-correction has also been applied to e-commerce platforms, achieving better product attribute extraction (Brinkmann and Bizer, 2025).

Beyond having an LLM generate its own feedback, an alternative involves using an external critic model (Hu et al., 2024). More recently, Aligner (Ji et al., 2024) employs a small-scale LLM fine-tuned for refinement tasks, which is used to correct the outputs of upstream LLMs. We adopt this decoupled paradigm, motivated by the narrower semantic gap (see Figure 1) inherent in refinement: rectifying a flawed invocation is significantly simpler than generating one from scratch. This reduced complexity and flexible structure enable our Refiner to provide effective, "plug-and-play" integration.

Reinforcement Learning for LLMs. Early Reinforcement learning (RL) approaches primarily utilized 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. It eliminates the value model by estimating advantages in a group-relative manner, thereby improving training efficiency.

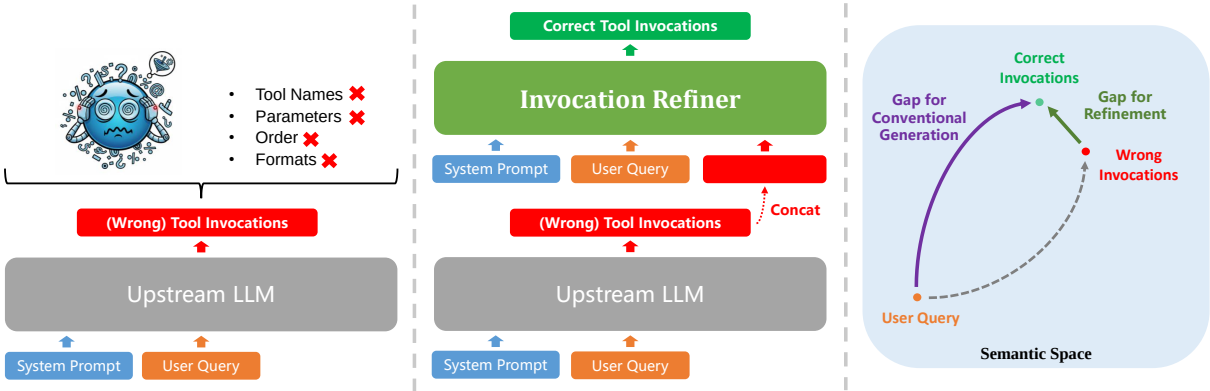


Figure 1: **(Left) Conventional Invocation Generation.** Direct invocation prediction is susceptible to frequent invocation errors. **(Middle) Workflow of the Invocation Refiner.** Acting as a plug-and-play module, the Refiner takes a system prompt, a user query, and the upstream model's raw output (containing reasoning traces and potential errors) to synthesize the corrected tool invocations. **(Right) Semantic Gap Comparison.** Unlike conventional generation that spans the wide gap from user queries to invocations, the Refiner targets the shorter semantic distance between a flawed invocation and its corrected form. This allows the Refiner to focus on syntax and logic constraints.

Building upon GRPO, Decoupled Clip and Dynamic Sampling Policy Optimization (DAPO) (Yu et al., 2025) introduces four key improvements: First, it employs an asymmetric clipping range $(1 - \varepsilon_{low}, 1 + \varepsilon_{high})$, 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. To surmount the challenges, a growing body of work has shifted towards RL (e.g., PPO, GRPO), which allows the model to explore the vast action space of possible tool invocations and learn directly from feedback signals that correlate with task success (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.

Building on these insights, we investigate three distinct training paradigms for our Invocation Refiner: standalone SFT, direct RL training via DAPO, and a hybrid approach utilizing SFT for cold-starting followed by DAPO optimization.

3 Methodology

3.1 Invocation Refiner

To address the rigidity and compatibility challenges associated with directly fine-tuning base LLMs, we propose a post-processing module: the Invocation Refiner. This specialized module is exclusively trained to rectify erroneous tool invocations generated by diverse upstream LLMs. As a model-agnostic post-processing unit, the Refiner enhances the reliability of tool use while keeping the upstream models frozen. This design ensures seamless, plug-and-play integration into existing agent pipelines, offering broad compatibility across various model families and sizes.

To initiate the refinement process, we first condition the Refiner with a global system prompt that defines its role as a tool invocation correction module. Subsequently, as shown in Figure 1, the Refiner takes three inputs: the task-specific system prompt, the user query, and the tool-integrated reasoning produced by the upstream LLM (including the initial tool invocations). Its sole objective is to analyze these inputs and perform targeted reasoning to generate the corrected tool invocations. By

standardizing the input-output format, the Refiner can be easily attached to different upstream models without requiring architectural modifications. The detailed prompt structure used to guide the Refiner is provided in Appendix A.

Furthermore, as shown in Figure 1, the Self-Refinement paradigm presents a more tractable learning objective, as 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, which is a highly efficient process, as confirmed by Aligner (Ji et al., 2024).

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 and use the correct tools from a given list based on a user query. In addition, the nested category comprises 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 Refiner, which is designed to correct tool invocations, we structure the data into a "*(Current Task, Upstream LLM Output, Correct Tool Invocations)*" triplet format. In addition, the "Upstream LLM Output" is preprocessed in advance, eliminating the need for real-time generation during training.

We generate four types of data, categorized based on the type of "Upstream LLM Output":

- **Erroneous Invocations:** The "Upstream LLM Output" in 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 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):** The "Upstream LLM Output" in this data contains nested tool calls from the NesTools dataset with the execution order randomized.

- **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 J).

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 invocations. 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 such as 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 $[-2, 2]$:

$$R_{\text{ToolName}} = 4 \cdot \frac{|O_n \cap G_n|}{|O_n \cup G_n|} - 2. \quad (1)$$

Tool Parameter Accuracy Reward. This reward is divided into two distinct yet complementary components. 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. The reward is calculated by equation (2). Second, the **parameter content reward** assesses the fidelity of the parameter values. Let O_{pc} be the multiset of parameter values in the model's output and G_{pc} be the multiset from the ground-truth. The reward is calculated by equation (3). Both rewards are scaled to the range $[-2, 2]$.

$$R_{\text{ParamName}} = 4 \cdot \frac{|O_{pn} \cap G_{pn}|}{|G_{pn}|} - 2. \quad (2)$$

$$R_{\text{ParamContent}} = 4 \cdot \frac{|O_{pc} \cap G_{pc}|}{|G_{pc}|} - 2. \quad (3)$$

Tool Order Reward. For tasks where the sequence of tool invocations is critical, this reward assesses the ordinal correctness. Let $S_G = (g_1, g_2, \ldots, g_m)$ be the ground-truth sequence of the tool invocations and $S_O = (o_1, o_2, \ldots, o_n)$ 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 $[-2, 2]$:

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

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 $R_{\text{pre-refine}}$ be the total reward for the model's output before refinement, and $R_{\text{post-refine}}$ be the total reward after refinement. If $R_{\text{post-refine}} < R_{\text{pre-refine}}$, the final reward for the sample will be overridden and set to the minimum possible value of the reward function. This mechanism can discourage "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 $[\text{min_reward}_i, \text{max_reward}_i]$ to a normalized range of $[0, 1]$.

3.4 Training Strategies

To strike a balance between inference efficiency and performance, we use Qwen3-1.7B (Team, 2025b) as the foundational architecture for the Refiner. To identify the most effective optimization paradigm for the refinement task, we investigate three distinct training strategies:

- **SFT:** A baseline approach where the model is optimized via standard supervised learning on the curated dataset.
- **DAPO:** We train the model directly using the DAPO RL algorithm to maximize the rule-based rewards defined in Section 3.3, without any prior fine-tuning.
- **SFT Cold Start + DAPO:** Following ToolRL (Qian et al., 2025), we perform an initial SFT cold start on 500 samples, followed by the DAPO training.

In all settings, the Refiner takes the "Current Task" and "Upstream LLM Output" as input, while the expected output is always the "Correct Tool Invocations", as shown in Section 3.1 and 3.2. Detailed hyperparameters and configurations are provided in Appendix D.

4 Evaluation

4.1 Overview

To comprehensively evaluate our Refiner's ability to correct erroneous invocations, we assess its performance across a diverse range of tasks:

- **Refinement of General-Purpose Tool Invocations:** We use the Berkeley Function-Calling Leaderboard (BFCL) (Patil et al., 2025) to evaluate performance across single-turn reasoning, multi-turn tool use, real-time

Model	Single Turn Non-Live (AST)				Single Turn Live (AST)				Multi Turn				Web Search	
	S	M	P	MP	S	M	P	MP	B	MF	MPm	LC	B	NS
Qwen2.5-3B-Instruct	73.67	89.00	79.00	76.50	67.83	64.67	56.25	37.50	6.50	3.50	4.50	4.50	0.00	0.00
+ Refiner (SFT)	45.83	59.50	64.00	67.50	39.92	49.10	50.00	45.83	9.50	6.00	5.00	4.00	2.00	1.00
+ Refiner (DAPO)	**75.67**	**92.00**	**85.00**	**83.50**	74.81	69.52	50.00	**58.33**	**11.00**	6.00	**7.00**	4.50	**4.00**	**3.00**
+ Refiner (SFT + DAPO)	75.33	91.50	82.00	82.50	**77.52**	**72.27**	**62.50**	54.17	10.00	**6.50**	5.50	**6.00**	3.00	**3.00**
Qwen2.5-7B-Instruct	74.42	91.00	87.50	81.50	78.68	73.69	62.50	62.50	10.50	3.00	8.00	5.00	1.00	1.00
+ Refiner (SFT)	47.33	75.00	77.00	74.50	52.33	58.02	56.25	54.17	15.00	8.50	10.00	8.00	10.00	3.00
+ Refiner (DAPO)	76.17	92.50	88.50	**83.50**	80.62	74.93	62.50	62.50	15.00	10.00	11.00	6.50	**12.00**	5.00
+ Refiner (SFT + DAPO)	**76.58**	**93.50**	**89.00**	83.00	**81.78**	**76.64**	**75.00**	**66.67**	**18.00**	**11.50**	**11.50**	**12.00**	9.00	**6.00**
Llama-3.2-3B-Instruct	70.67	92.50	88.50	78.50	65.12	57.26	18.75	37.50	5.00	4.00	3.00	3.00	2.00	1.00
+ Refiner (SFT)	45.08	74.00	77.00	75.00	48.45	48.91	43.75	50.00	7.50	3.50	3.00	5.50	2.00	**2.00**
+ Refiner (DAPO)	**77.25**	92.50	**89.50**	**85.50**	74.03	**68.57**	62.50	45.83	**11.00**	**8.50**	6.00	6.50	3.00	**2.00**
+ Refiner (SFT + DAPO)	75.33	**93.00**	89.00	**85.50**	**74.42**	68.09	**68.75**	**54.17**	9.50	8.00	**7.00**	**7.00**	**6.00**	**2.00**
Llama-3.1-8B-Instruct	71.92	93.00	86.50	83.50	71.71	71.23	50.00	45.83	12.50	8.00	7.50	10.00	6.00	2.00
+ Refiner (SFT)	28.50	32.50	67.00	66.00	27.52	24.31	56.25	54.17	8.00	8.00	6.50	9.50	5.00	3.00
+ Refiner (DAPO)	**72.50**	**95.50**	**88.50**	**86.50**	74.03	74.36	62.50	**75.00**	**18.00**	9.00	10.00	11.50	7.00	**7.00**
+ Refiner (SFT + DAPO)	72.33	93.00	88.00	85.00	**78.68**	**75.21**	**68.75**	50.00	15.50	**10.50**	**10.50**	**13.00**	**8.00**	6.00
Qwen3-1.7B	74.92	88.00	80.00	77.50	74.03	66.29	**68.75**	70.83	10.00	11.00	8.00	2.50	5.00	3.00
+ Refiner (SFT)	46.33	75.00	72.00	71.50	55.04	53.09	50.00	66.67	12.50	14.50	9.50	8.00	2.00	0.00
+ Refiner (DAPO)	76.17	86.00	**82.00**	81.00	**80.62**	66.19	62.50	**75.00**	14.00	13.50	**13.00**	**11.00**	5.00	**4.00**
+ Refiner (SFT + DAPO)	**77.50**	**91.50**	81.50	**83.00**	80.23	**68.19**	62.50	**75.00**	**15.08**	**15.50**	12.00	9.50	**7.00**	**4.00**
Qwen3-8B	77.50	94.50	90.50	**91.00**	80.23	77.59	81.25	70.83	35.50	37.00	22.50	20.00	12.00	4.00
+ Refiner (SFT)	51.25	75.00	80.00	79.00	50.78	55.37	81.25	66.67	26.50	28.00	21.00	18.00	8.00	7.00
+ Refiner (DAPO)	77.50	87.00	89.50	81.50	82.17	74.07	87.50	75.00	**36.00**	**37.50**	**25.00**	**22.00**	**15.00**	**9.00**
+ Refiner (SFT + DAPO)	**79.33**	**95.50**	**91.50**	90.50	**83.72**	**79.77**	**93.75**	**83.33**	34.50	**37.50**	23.62	**22.00**	14.00	8.00
Hammer2.0-7B	72.92	92.50	89.50	89.50	74.81	74.26	81.25	70.83	10.00	**8.00**	8.50	6.00	0.00	0.00
+ Refiner (SFT)	69.75	92.50	91.00	88.00	80.23	75.50	75.00	66.67	9.50	6.50	8.50	5.50	-	-
+ Refiner (DAPO)	**74.92**	**94.50**	**94.00**	88.50	75.97	73.41	81.25	**75.00**	10.50	6.50	9.00	**6.50**	-	-
+ Refiner (SFT + DAPO)	73.67	**94.50**	93.00	**90.50**	**77.13**	**74.36**	**87.50**	70.83	**13.50**	**8.00**	**10.00**	**6.50**	-	-
ToolACE-2-8B	74.00	91.50	**93.50**	91.00	71.32	**79.77**	75.00	62.50	**49.00**	**29.50**	**32.00**	**45.50**	**13.00**	6.00
+ Refiner (SFT)	51.25	68.00	77.50	75.00	52.71	60.59	68.75	50.00	26.50	22.50	18.00	23.50	8.00	3.00
+ Refiner (DAPO)	**77.50**	93.50	91.50	90.50	77.91	75.40	**81.25**	75.00	33.50	22.00	19.50	33.00	12.00	4.00
+ Refiner (SFT + DAPO)	76.75	**95.50**	93.00	**92.00**	**80.62**	78.82	**81.25**	**79.17**	39.00	27.00	28.00	34.50	**13.00**	**7.00**

Table 1: Results on the Berkeley Function-Calling Leaderboard (General-Purpose). Abbreviations denote: Simple, Multiple, Parallel, and Multiple-Parallel for **Single-Turn**; Base, Missing Functions, Missing Parameters, and Long Context for **Multi-Turn**; Base, No Snippet for **Web Search**. Detailed setups are provided in Appendix E.2.

execution, and web search (**Section 4.2**). Additionally, results on the API-Bank benchmark (Li et al., 2023) are provided in Appendix G.

- **Refinement of Sequential Errors:** We choose NESTFUL (Basu et al., 2024), which requires sequential tool calls to solve mathematical problems, to evaluate the correction of tool invocation order (**Section 4.3**).

- **Quantitative Analysis of Format and Content Correction:** We adapt the Bamboogle dataset (Press et al., 2022) to quantitatively analyze our Refiner's efficacy in rectifying format and parameter errors (**Section 4.4**).

To demonstrate plug-and-play versatility, we deploy our Refiner across a diverse set of upstream models, including six general-purpose LLMs from the Qwen and Llama series (Team, 2025a; AI@Meta, 2024), as well as two LLMs optimized for tool-use: Hammer2.0-7B (Lin et al., 2024) and ToolACE-2-8B (Liu et al., 2024).

4.2 Refinement of General-Purpose Tool Invocations

Experiment Settings. To evaluate our Refiner's capability in refining general-purpose tool invocations, we utilize the Berkeley Function-Calling Leaderboard (BFCL). We assess performance across a broad spectrum of complexities, including Single-Turn (e.g., Parallel, Multiple), Multi-Turn, and Web Search tasks. Detailed setups are provided in Appendix E.2. The results are in Table 1.

Superior Generalization of DAPO-Enhanced Refiner. The results highlight a critical disparity between training methodologies. The SFT-only Refiner frequently degrades performance, which is due to overfitting that hampers generalization to the unseen tasks. Conversely, Refiners trained with DAPO (both Pure and Cold Start, where Cold Start

Model	NESTFUL			Bamboogle*		
	Part. Acc.	Full Acc.	Win Rate	Acc_{Full}	$\text{Acc}_{Toolname}$	$\text{Acc}_{Parameter}$
Qwen2.5-3B-Instruct	0.15/0.10	0.09/0.04	0.11/0.13	0.00	54.27	52.27
+ Refiner (SFT)	0.09/0.08	0.06/0.05	0.13/0.12	10.13	19.93	18.93
+ Refiner (DAPO)	**0.21/0.19**	**0.13/0.10**	**0.21/0.23**	48.13	79.67	79.47
+ Refiner (SFT + DAPO)	0.18/0.16	0.12/0.09	0.17/0.20	**50.07**	**82.93**	**82.13**
Qwen2.5-7B-Instruct	0.24/0.25	0.19/0.18	0.25/0.18	20.87	60.40	60.40
+ Refiner (SFT)	0.08/0.07	0.06/0.05	0.13/0.11	4.40	15.07	14.93
+ Refiner (DAPO)	**0.27/0.28**	**0.21/0.22**	**0.29/0.31**	**49.13**	**79.47**	**79.47**
+ Refiner (SFT + DAPO)	0.25/0.27	0.20/0.21	0.28/**0.31**	31.60	61.27	61.27
Llama-3.2-3B-Instruct	0.13/0.11	0.04/0.03	0.04/0.04	1.60	54.20	45.70
+ Refiner (SFT)	0.09/0.09	0.05/0.05	0.13/0.12	8.53	23.07	21.47
+ Refiner (DAPO)	0.19/**0.21**	0.09/0.12	0.13/0.17	**36.53**	**74.73**	**73.80**
+ Refiner (SFT + DAPO)	**0.21/0.21**	**0.13/0.13**	**0.16/0.18**	14.87	63.00	61.60
Llama-3.1-8B-Instruct	0.22/0.21	0.16/0.15	0.11/0.09	43.07	64.80	60.67
+ Refiner (SFT)	0.06/0.06	0.04/0.04	0.10/0.09	4.40	15.07	14.93
+ Refiner (DAPO)	**0.27/0.27**	**0.19/0.20**	**0.25/0.25**	**60.27**	**72.20**	**72.00**
+ Refiner (SFT + DAPO)	0.25/0.26	0.18/0.19	0.22/0.23	49.40	63.60	63.27
Qwen3-1.7B	0.17/0.15	0.11/0.12	0.12/0.20	16.70	62.27	59.40
+ Refiner (SFT)	0.08/0.08	0.05/0.05	0.13/0.14	5.33	15.27	14.53
+ Refiner (DAPO)	**0.21/0.22**	**0.13/0.15**	**0.19/0.31**	39.67	68.60	66.80
+ Refiner (SFT + DAPO)	0.20/0.20	**0.13**/0.14	**0.19**/0.28	**42.33**	**75.07**	**72.80**
Qwen3-8B	0.29/0.25	0.23/0.21	0.25/0.33	80.33	86.87	86.80
+ Refiner (SFT)	0.06/0.08	0.05/0.06	0.09/0.16	6.33	19.60	19.20
+ Refiner (DAPO)	**0.30/0.29**	**0.24/0.22**	**0.29/0.41**	**86.60**	87.13	87.13
+ Refiner (SFT + DAPO)	**0.30/0.27**	0.23/0.21	**0.33**/0.40	79.13	**90.33**	**90.27**
Hammer2.0-7B	0.29/-	0.22/-	0.27/-	24.80	60.53	55.00
+ Refiner (SFT)	0.16/-	0.11/-	0.25/-	5.33	14.60	13.80
+ Refiner (DAPO)	0.29/-	0.21/-	0.30/-	**44.60**	**77.27**	**76.87**
+ Refiner (SFT + DAPO)	**0.32**/-	**0.25**/-	**0.33**/-	29.40	61.80	61.47
ToolACE-2-8B	0.01/-	0.00/-	0.00/-	10.40	54.27	54.13
+ Refiner (SFT)	0.11/-	0.07/-	0.14/-	11.20	23.60	23.33
+ Refiner (DAPO)	0.13/-	0.08/-	0.15/-	42.20	75.87	75.47
+ Refiner (SFT + DAPO)	**0.15**/-	**0.10**/-	**0.20**/-	**43.93**	**82.93**	**82.53**

Table 2: Results on the NESTFUL benchmark (Sequential Errors) and the Bamboogle* benchmark (Format and Content Correction). For the NESTFUL benchmark, scores are reported as "Raw / CoT".

marginally outperforms Pure) consistently deliver gains across the board. This confirms that RL effectively develops a robust error-correction policy, avoiding being constrained by the rigid formatting of supervised demonstrations.

Consistent Gains in Single-Turn and Web Search. In Single-Turn and Web Search evaluations, our plug-and-play module demonstrates universal efficacy. The DAPO-enhanced Refiners comprehensively boost execution accuracy for both general-purpose models and specialized tool-use models (Hammer2.0-7B excluded from Web Search due to format mismatch). Case analysis reveals that by intercepting foundational errors, such as syntax hallucinations or parameter mismatches, the Refiner upgrades the output quality of various upstream LLMs, demonstrating that even models fine-tuned for tool use benefit from an external correction mechanism to handle edge cases.

Performance in Complex Multi-Turn Tasks. Regarding Multi-Turn scenarios, the Refiner bolsters the capabilities of general-purpose LLMs and Hammer2.0-7B. However, a minor exception is observed with ToolACE-2-8B, which has been specifically optimized for multi-turn tool use. The regression suggests that our Refiner, which has not been exposed to complex multi-turn formats during training, lacks the generalization for this specific scenario. Enhancing the Refiner's adaptability to multi-turn context management remains a key direction for future improvement.

4.3 Refinement of Sequential Errors

Experiment Settings. We evaluate sequential error correction using NESTFUL, which demands logical sequences of tool invocations and intermediate variable management. Performance is assessed via three metrics: **Part. Acc.** (the average ratio of correct individual calls), **Full Acc.** (exact sequence matches), **Win Rate** (the percentage of samples yielding the correct final answer). Detailed setups are shown in Appendix E.3. Table 2 reports the results, presented in a "Raw / CoT" format.

Superiority of DAPO Training. As shown in Table 2, the SFT-enhanced Refiner struggles to generalize and degrades performance on sequence correction. In contrast, the DAPO-enhanced Refiner yields improvements across all metrics. This confirms that the RL training is effective for mastering the complex logic of sequential dependency and variable management.

Consistent Performance Gains. The DAPO-enhanced Refiners consistently boost performance for both standard and CoT-augmented upstream models, often enabling smaller models to outperform larger baselines. Case analysis shows that the Refiner effectively rectifies format errors, inserts missing variables, eliminates redundancies, and reorders sequences to ensure logical execution.

Robustness to Flawed Reasoning. Notably, while low-quality CoT reasoning can sometimes degrade raw model performance, our Refiner effectively mitigates this issue. It recovers lost performance and sometimes leverages the CoT context for further gains (e.g., Llama-3.1-8B-Instruct), proving its ability to extract correct execution intent even from noisy or hallucinated reasoning traces.

4.4 Quantitative Analysis of Format and Content Correction

Experiment Settings. Given the well-structured nature of the Bamboogle dataset, which features predefined retrieval categories limited to celebrity birth information, we reformulate its search queries into a tool-calling format, yielding the Bamboogle* benchmark. This modified dataset enables a quantitative assessment, which we dissect by individually analyzing the tool format, name, and parameters: $\mathbf{Acc}_{Toolname}$ (tool selection), $\mathbf{Acc}_{Parameter}$ (tool and parameter), $\mathbf{Acc}_{Full}$ (tool, parameter, and format). Detailed specifications are provided in Appendix E.4. The results are shown in Table 2.

Superiority of DAPO Training. In terms of format and content correction, the DAPO-enhanced Refiners continue to outperform the SFT-enhanced one. As shown in Table 2, the SFT-based Refiner frequently degrades performance, failing to generalize to the specific formatting nuances of the Bamboogle* benchmark. In contrast, both DAPO-based approaches (Pure and Cold Start) achieve substantial gains. This reinforces the finding that the RL objective is crucial for learning a flexible and robust correction policy.

Enhancement of Format and Semantic Accuracy. The deployment of DAPO-trained Refiners leads to substantial improvements in Acc_{Full}, highlighting their efficacy in rectifying syntactic errors. Furthermore, the marked increases in $\text{Acc}_{Toolname}$ and $\text{Acc}_{Parameter}$ demonstrate that the Refiner actively corrects semantic problems. By effectively extracting relevant details from the user query and context, the Refiner corrects errors in tool selection and parameter generation, thereby elevating the overall utility of the upstream LLM.

5 Overview of Supplementary Materials

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, the complete training and evaluation configurations, and the inference efficiency (Appendix A, B, C, D, E, F). Subsequently, we present additional experiments and rigorous ablation studies to demonstrate that the Refiner's strong performance stems from our proposed training strategy and data choices, specifically by: further validating the Refiner on the API-Bank benchmark, comparing performance against an untrained baseline, ablating the overlong penalty, and ablating the use of simplified nested invocation training data (Appendix G, H, I, J). Finally, we provide case studies to substantiate our findings (Appendix M).

6 Conclusion

We introduce a specialized post-processing module, termed the **Invocation Refiner**, designed to rectify tool invocations generated by arbitrary upstream LLMs. Critically, our approach functions as a universal plug-and-play solution, decoupling the correction process from the upstream generation. This design allows for the seamless enhancement of Tool-Integrated Reasoning across diverse base LLMs without the need for model-specific modifications or retraining. By leveraging the DAPO RL algorithm, the Refiner effectively generalizes to identify and correct a wide range of invocation errors, including malformed formats, incorrect parameters, and logical sequencing flaws. Our comprehensive evaluations across multiple tool-use benchmarks demonstrate that this modular approach delivers consistent performance gains for various upstream models, paving the way for building more robust, adaptable, and reliable AI agents through flexible component integration.

Limitations

While our Invocation Refiner demonstrates robust performance across a wide array of tasks, we acknowledge certain limitations that outline directions for future research.

First, our current training curriculum focuses primarily on single-turn interactions. As discussed in Section 4.2, the Refiner's ability to fully leverage historical context in complex multi-turn scenarios remains an area for further enhancement.

Second, the model cascading framework inevitably introduces inference latency. However, as detailed in Appendix F, this overhead remains within a manageable range.

Ethical considerations

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

References

Glaive AI. 2023. glaive-function-calling-v2.

AI@Meta. 2024. Llama 3 model card.

Kinjal Basu, Ibrahim Abdelaziz, Kiran Kate, Mayank Agarwal, Maxwell Crouse, Yara Rizk, Kelsey Bradford, Asim Munawar, Sadhana Kumaravel, Saurabh Goyal, and 1 others. 2024. Nestful: A benchmark for evaluating llms on nested sequences of api calls. *arXiv preprint arXiv:2409.03797*.

Alexander Brinkmann and Christian Bizer. 2025. Self-refinement strategies for llm-based product attribute value extraction. *arXiv preprint arXiv:2501.01237*.

Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. 2023a. Fireact: Toward language agent fine-tuning. *arXiv preprint arXiv:2310.05915*.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023b. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Zehui Chen, Kuikun Liu, Qiuchen Wang, Wenwei Zhang, Jiangning Liu, Dahua Lin, Kai Chen, and Feng Zhao. 2024. Agent-flan: Designing data and methods of effective agent tuning for large language models. *arXiv preprint arXiv:2403.12881*.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, and 1 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, and 1 others. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. 2025. Retool: Reinforcement learning for strategic tool use in llms. *arXiv preprint arXiv:2504.11536*.

Han Han, Tong Zhu, Xiang Zhang, Mengsong Wu, Hao Xiong, and Wenliang Chen. 2024. Nestools: A dataset for evaluating nested tool learning abilities of large language models. *arXiv preprint arXiv:2410.11805*.

Chi Hu, Yimin Hu, Hang Cao, Tong Xiao, and Jingbo Zhu. 2024. Teaching language models to self-improve by learning from language feedback. *arXiv preprint arXiv:2406.07168*.

Jiaming Ji, Boyuan Chen, Hantao Lou, Donghai Hong, Borong Zhang, Xuehai Pan, Tianyi Alex Qiu, Juntao Dai, and Yaodong Yang. 2024. Aligner: Efficient alignment by learning to correct. *Advances in Neural Information Processing Systems*, 37:90853–90890.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. 2025. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*.

Xuefeng Li, Haoyang Zou, and Pengfei Liu. 2025. Torl: Scaling tool-integrated rl. *arXiv preprint arXiv:2503.23383*.

Qiqiang Lin, Muning Wen, Qiuying Peng, Guanyu Nie, Junwei Liao, Jun Wang, Xiaoyun Mo, Jiamu Zhou, Cheng Cheng, Yin Zhao, and 1 others. 2024. Hammer: Robust function-calling for on-device language models via function masking. *arXiv preprint arXiv:2410.04587*.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, and 1 others. 2024. Toolace: Winning the points of llm function calling. *arXiv preprint arXiv:2409.00920*.

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

Lilian Ngweta, Mayank Agarwal, Subha Maity, Alex Gittens, Yuekai Sun, and Mikhail Yurochkin. 2024. Aligners: Decoupling llms and alignment. *arXiv preprint arXiv:2403.04224*.

Shishir G. Patil, Huanzhi Mao, Charlie Cheng-Jie Ji, Fanjia Yan, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. 2025. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*.

Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah A Smith, and Mike Lewis. 2022. Measuring and narrowing the compositionality gap in language models. *arXiv preprint arXiv:2210.03350*.

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

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Xuanhe Zhou, Yufei Huang, Chaojun Xiao, and 1 others. 2024. Tool learning with foundation models. *ACM Computing Surveys*, 57(4):1–40.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Huatong Song, Jinhao Jiang, Yingqian Min, Jie Chen, Zhipeng Chen, Wayne Xin Zhao, Lei Fang, and Ji-Rong Wen. 2025. R1-searcher: Incentivizing the search capability in llms via reinforcement learning. *arXiv preprint arXiv:2503.05592*.

Qwen Team. 2025a. Qwen2.5-vl.

Qwen Team. 2025b. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Tu Vu, Mohit Iyyer, Xuezhi Wang, Noah Constant, Jerry Wei, Jason Wei, Chris Tar, Yun-Hsuan Sung, Denny Zhou, Quoc Le, and 1 others. 2023. Freshllms: Refreshing large language models with search engine augmentation. *arXiv preprint arXiv:2310.03214*.

Hongru Wang, Yujia Qin, Yankai Lin, Jeff Z Pan, and Kam-Fai Wong. 2024. 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*, pages 2983–2986.

Ke Wang, Houxing Ren, Aojun Zhou, Zimu Lu, Sichun Luo, Weikang Shi, Renrui Zhang, Linqi Song, Mingjie Zhan, and Hongsheng Li. 2023. Mathcoder: Seamless code integration in llms for enhanced mathematical reasoning. *arXiv preprint arXiv:2310.03731*.

Zhengyuan Yang, Jianfeng Wang, Linjie Li, Kevin Lin, Chung-Ching Lin, Zicheng Liu, and Lijuan Wang. 2024. Idea2img: Iterative self-refinement with gpt-4v for automatic image design and generation. In *European Conference on Computer Vision*, pages 167–184. Springer.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, and 1 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*.

Xiang Yue, Xingwei Qu, Ge Zhang, Yao Fu, Wenhao Huang, Huan Sun, Yu Su, and Wenhu Chen. 2023. Mammoth: Building math generalist models through hybrid instruction tuning. *arXiv preprint arXiv:2309.05653*.

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

Jianguo Zhang, Tian Lan, Ming Zhu, Zuxin Liu, Thai Hoang, Shirley Kokane, Weiran Yao, Juntao Tan, Akshara Prabhakar, Haolin Chen, and 1 others. 2024. xlam: A family of large action models to empower ai agent systems. *arXiv preprint arXiv:2409.03215*.

Yuze Zhao, Jintao Huang, Jinghan Hu, Xingjun Wang, Yunlin Mao, Daoze Zhang, Zeyinzi Jiang, Zhikai Wu, Baole Ai, Ang Wang, Wenmeng Zhou, and Yingda Chen. 2024. Swift:a scalable lightweight infrastructure for fine-tuning. *Preprint*, arXiv:2408.05517.

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 Invocation Refiner:** We present the template and format of the input prompt for our Invocation 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 Invocation Refiner:** We detail the process of creating the training datasets for our Invocation Refiner.

2. Experimental Setup and Training Details:

- Appendix D. **Training Details:** We present the training configurations for our Invocation Refiner, 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 a quantitative correction analysis.

3. Model Performance and Ablation Studies:

- Appendix G. **Performance on the General-Purpose Tool-use Benchmark API-Bank:** We evaluate our Refiner on the API-Bank benchmark, demonstrating that the DAPO-enhanced Refiner consistently enhances the performance of various upstream models by rectifying format and semantic errors, whereas the SFT-only variant struggles with generalization due to distribution shifts.
- Appendix H. **Performance Comparison with an Untrained Refiner:** We compare our Invocation Refiner against an untrained counterpart (Qwen3-1.7B), demonstrating that our Refiner's strong performance is attributable to our RL training for tool invocation correction, rather than the inherent capabilities of the raw model.
- Appendix I. **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 J. **An Ablation Study of the Simplified Nested Invocation Training Data:** We conduct an ablation study on the simplified nested invocation 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:

- Appendix F, **Inference Efficiency:** We evaluate the time overhead introduced by the Refiner, showing that the latency is manageable and can be effectively mitigated through decoupled deployment and selective invocation.
- Appendix M, **Case Studies:** We present representative case studies to substantiate the findings from our comparative evaluation.

A Crafting the Input Prompt for Invocation Refiner

This section delineates the structure of the input prompt for the Invocation Refiner.

The input prompt begins with a system prompt designed to establish the Refiner's role and operational scope. Specifically, it explicitly instructs the Refiner that its core objective is to identify and rectify errors within tool invocations produced by upstream models. Additionally, the system prompt elucidates the input format and enumerates the potential categories of errors the model may encounter. The complete system prompt utilized in our experiments is provided 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.

For each call to the Refiner, we provide the task-specific system prompt, the current user query, and the upstream LLM's generation, which includes its reasoning trace along with the potentially flawed tool invocations. The specific input format is defined 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 current sample, QUERY refers to the user query for the current sample, and UPSTREAM refers to the reasoning trace and 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 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"]}, "required": null}, {"name": "Get Supported Currencies", "description": "Retrieve a list of supported currencies by Coinbase.", "parameters": {"type": "dict", "properties": {}, "required": []}, "required": null}, {"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"]}, "required": null}].

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: *<function_list> [{"name": "Financial_Fundamentals_API", "results": {"roa_ratio": 0.123, "financial_year": "2025"}}] < /function_list>*

B.3 xLAM

The tool invocation format of the xLAM dataset is: *[{"name": functio_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 system prompt is: *You are an expert in composing functions. Based on the question, you will need to make one or more function/tool calls to achieve the purpose. Here is a list of functions in JSON format that you can invoke: [{"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: *<func_call> [{"name": "live_giveaways_by_type", "arguments": {"type": "beta"}}, {"name": "live_giveaways_by_type", "arguments": {"type": "game"}}] < /func_call>.*

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. In addition, 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 system prompt is: *You have access to a list of APIs and the task description. You need to follow the given task description and determine which API to call in sequence according to the order required by the task description. API can be retrieved from the APIs list. You don't need to know the actual return value of the API call, just assign each return value as a string "API_call_{number}" in "responses", such as "API_call_0","API_call_1","API_call_2" and so on. The "number" in "API_call_{number}" should increase by one from 0 globally. Please first determine which APIs to call in sequence based on the task, and then determine the parameter values of each API depending on the specific details of the task. If you decide to call the API, you need to fill in all this API's required parameters which can be found in this API's "required" list. If you think the task does not include the actual value of a necessary parameter in API's "required" list, you can assign the necessary parameter a value of "UNK". The remaining parameters are optional parameters, determine whether to fill them in according to the task. If you think the parameter value to be filled in is the return value of a previous API call, set it as "API_call_x", then the parameter value can be filled in with "API_call_x". Now it is your turn to generate the API call result based on the APIs and task description below. Remember that you only need to generate the API call result, not any additional explanations. APIs:*

[{"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"}}, "required": ["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 Invocation Refiner

C.1 Generation of Erroneous Tool Invocations

To construct a training dataset of erroneous tool invocations for our Invocation Refiner, we systematically generate flawed data by leveraging five LLMs with diverse scales and performance characteristics: Qwen3-1.7B, Qwen3-8B, Qwen3-14B (Team, 2025b), 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 ground-truth 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 ground-truth 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 generates a collection of tool invocations with errors intentionally injected by five distinct LLMs. From this pool, we construct our final dataset by randomly sampling 150 instances per error type for each model. To ensure diversity, each 150-sample subset comprises an equal split of 50 instances from the Glaive, ToolACE, and xLAM datasets. This strategy results in a total of 2,250

samples designed to mimic the erroneous outputs of upstream LLMs within our training framework.

C.2 Generation of Shuffled Nested Invocations

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 invocations. 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 tool names 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 (Ablation Study in Appendix J).

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

We outline the specific hyperparameter settings and configurations used for the training below.

Supervised Fine-Tuning (SFT). For the SFT phase, we employ a learning rate of 1e-6 with a warmup ratio of 0.05. The Refiner is trained for a single epoch with a global batch size of 32. To accommodate complex reasoning chains and tool definitions, we utilize a maximum context window of 4096 tokens.

Reinforcement Learning with DAPO. For the RL training phase using the DAPO algorithm, we maintain the learning rate at 1e-6 and the maximum context window at 4096 tokens. The model is optimized for a single epoch with a training batch size of 16. Specifically for the DAPO algorithm, we configure the following mechanisms:

- **Asymmetric Clipping:** We set the lower clipping range (`clip_ratio_low`) to 0.2 and the higher clipping range (`clip_ratio_high`) to 0.28 to stabilize policy updates.
- **Generation and Sampling:** For each sample in a batch, 16 candidate responses are generated. To ensure the quality of training signals, we employ a dynamic sampling strategy that filters out samples providing no learning signal. To construct a complete batch of 16 qualified samples, the trainer is permitted to perform up to 48 sampling attempts.
- **Overlong Response Penalty:** To discourage verbosity and ensure efficiency, we implement a length penalty mechanism. Responses exceeding 1024 tokens incur a penalty that increases linearly from 0 (at 1024 tokens) to a maximum of 1 (at 4096 tokens).

Unlike open-ended generation where the action space is vast, the refinement task operates within a constrained semantic neighborhood of the upstream

output. This constrained semantic gap makes it easier for Reinforcement Learning to explore, as the policy optimization can focus on fine-grained adjustments (e.g., format alignment, parameter constraint) rather than learning reasoning from high-level input.

D.2 Training Time and Resource Consumption

We provide a breakdown of the computational resources required for the different training strategies:

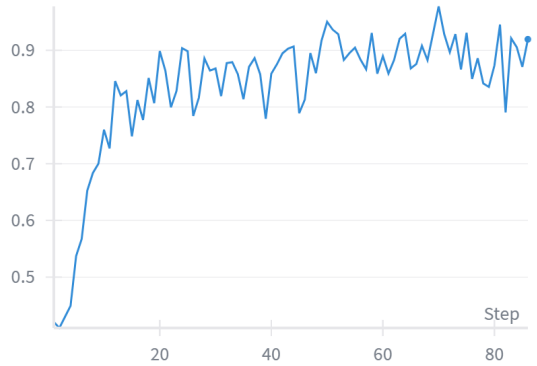
- The SFT phase is conducted on a cluster of four NVIDIA L20 GPUs. The entire process is highly efficient, completing in approximately 40 minutes with a VRAM occupancy of around 18GB per GPU.
- For the reinforcement learning stage using DAPO, we utilize four NVIDIA RTX A5000 GPUs. The training session lasts for 6 hours and consumes approximately 22GB of VRAM on each GPU.

D.3 DAPO Training Results

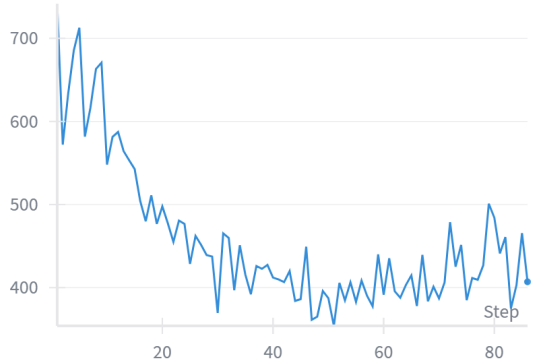
Figure 2 illustrates the training dynamics of our Invocation Refiner under DAPO strategy, plotting the changes in reward and average response length.

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.



(a) Normalized Reward



(b) Response Length

Figure 2: Training dynamics of our Invocation Refiner under DAPO strategy.

E Evaluation Setup

E.1 Deployment of the Invocation Refiner

During the evaluation, we deploy the Invocation Refiner using the ms-swift framework (Zhao et al., 2024). The model is hosted on two NVIDIA L20 GPUs and accelerated using vLLM (Kwon et al., 2023), with the "max_new_tokens" parameter set to 4096. Regarding the inference workflow, once the output from the upstream LLM is obtained, we transmit it to the deployed Invocation Refiner via an API call. The Refiner then processes this input and returns the rectified tool invocation.

E.2 Evaluation Setup for the Berkeley Function-Calling Leaderboard

To rigorously assess our Refiner's versatility across diverse tool-use scenarios, we employ the Berkeley Function-Calling Leaderboard (BFCL) (Patil et al., 2025). Our evaluation covers a broad spectrum of tasks, including Single-Turn tool invocation, Multi-Turn interactions, and Web Search capabilities. We utilize the official evaluation scripts provided by the BFCL repository to ensure standardized comparisons.

E.2.1 Single-Turn Evaluation Setup

The Single-Turn assessment measures the model's ability to generate correct tool calls in a single interaction. The evaluation methodology relies on Abstract Syntax Tree (AST) Evaluation, which verifies correctness across four dimensions: Function Name Matching, Required Parameters Matching, Parameter Type Matching, and Parameter Value Matching. The evaluation is stratified into four task categories:

- **Simple (S):** The toolset contains a single tool; the model is required to invoke this tool once.
- **Multiple (M):** The toolset contains multiple tools; the model must select the most appropriate tool and invoke this tool once.
- **Parallel (P):** The toolset contains a single tool; the model is required to invoke this tool multiple times within a single turn.
- **Multiple-Parallel (MP):** The toolset contains multiple tools; the model must select appropriate tools and perform multiple invocations across them in a single turn.

Furthermore, these tasks are divided into Live and Non-Live datasets. "Live" refers to user-contributed, real-world scenarios, while "Non-Live" consists of expert-curated synthetic tasks.

E.2.2 Multi-Turn Evaluation Setup

The Multi-Turn evaluation assesses the model's capability to maintain state and logic over time. It encompasses Multi-Step interactions, where the model executes multiple internal function calls to address a single user request, and Multi-Turn interactions, which involve extended conversational exchanges between the user and the model. Performance is measured using both state-based and response-based evaluations across four sub-tasks:

- **Base (B):** All necessary information is provided. The model is expected to handle the interaction without ambiguity.
- **Missing Functions (MF):** Tests the model's ability to identify that no available tool can fulfill the request.
- **Missing Parameters (MPm):** Tests the ability to recognize when essential information is missing, requiring the model to ask for clarification rather than making assumptions.
- **Long-Context (LC):** Challenges the model's ability to maintain accuracy and relevance in lengthy, information-dense contexts.

It is worth noting that in multi-turn scenarios, our Refiner is tasked exclusively with correcting the tool invocations at each step. When the upstream model generates a dialogue termination response (i.e., a final answer to the user), the Refiner is bypassed, allowing the conversation to conclude naturally based on the upstream model's output.

E.2.3 Web Search Evaluation Setup

This setup enables the model to utilize a browser tool that integrates a search engine with the ability to visit websites and view their content. It evaluates the model's ability to retrieve, evaluate, and synthesize online information to answer complex questions. Additionally, the retrieval API returns both the URL and a content snippet. To distinguish between simple snippet understanding and deep website content understanding, the evaluation is divided into two categories:

- **Base (B):** The model is provided with the content snippet alongside the URL.
- **No Snippet (NS):** The model receives only the site URL and title, forcing it to retrieve and process the actual page content to answer the query.

For the Hammer2.0-7B model, its output format is incompatible with the termination format of the BFCL Web Search task, preventing it from successfully ending the conversation even when it possesses accurate context. Consequently, we exclude Hammer2.0-7B from the Web Search evaluation.

E.3 Evaluation Setup for the NESTFUL benchmark

Our evaluation utilizes the NESTFUL benchmark (Basu et al., 2024), designed to assess the execution of logical tool invocation sequences in multi-step problem-solving scenarios. A defining challenge of this benchmark is the management of intermediate states: models must capture the output of a tool call in a temporary variable (e.g., var_0) and correctly pass this variable as an argument to subsequent calls. This rigorously tests multi-step reasoning and planning capabilities, mirroring complex tasks such as chained mathematical computations.

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 must then generate a JSON-formatted list of tool calls to satisfy the user's query. Each call specifies the tool name, arguments, and a label parameter (e.g., var_0) for variable assignment.

We rigorously verify the correctness of the generated execution chain using three key metrics:

- **Partial Sequence Accuracy (Part. Acc.):** The average percentage of correct API calls (matching both name and arguments) within the generated sequence, relative to the ground truth sequence.
- **Full Sequence Accuracy (Full Acc.):** The percentage of samples where the model generates the entire sequence of API calls correctly, matching the ground truth in both content and order.
- **Win Rate:** A functional correctness metric measuring the percentage of tasks where the predicted APIs are fully executable and the final computed result exactly matches the ground truth answer. This involves sequentially executing the tool calls and tracking intermediate variable values.

E.4 Evaluation Setup for the Bamboogle* benchmark

This evaluation leverages the Bamboogle dataset (Press et al., 2022), which consists of 17 distinct retrieval tasks centered on celebrity birth dates and birthplaces. To facilitate a granular assessment of tool invocation capabilities, we restructure these tasks into a tool-calling framework, termed Bamboogle*. This involves mapping the original retrieval objectives into a suite of distinct APIs, including foundational APIs for retrieving raw birth data and secondary APIs designed to process birth data for downstream tasks. Consequently, the task transforms from direct question-answering into generating the correct sequence of API calls.

For instance, to complete the retrieval for "What is the capital of the birthplace of *CELEBRITY*?", a related 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: *{"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 environment enables a multi-faceted quantitative assessment of the model's performance. We report three hierarchical metrics to dissect the nature of invocation errors:

- **Acc**$_{Full}$: The percentage of predictions where the tool names, parameters, and format are all correct.
- **Acc**$_{Toolname}$: The percentage of predictions that contain the correct tools, regardless of its parameters or format.

- **Acc**$_{Parameter}$: The percentage of predictions that contain the correct tools and the correct parameters, irrespective of the format.

This framework provides an ideal testbed for our Invocation Refiner, allowing us to isolate its contribution to semantic reasoning (tool or parameter selection) versus syntactic compliance (formatting).

E.5 Evaluation Setup for the API-Bank Benchmark

E.5.1 API-Bank Benchmark Specification

API-Bank (Li et al., 2023) 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 use the API-bank dataset provided by the official ToolRL repository (Qian et al., 2025), which 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.5.2 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 structure 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 instruction. 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 ensures a more accurate evaluation.

F Inference Efficiency

To investigate the computational overhead introduced by our Refiner, we conduct a time consumption analysis on the Bamboogle* benchmark. We compare the total inference time of Llama-3.1-8B-Instruct, Qwen3-1.7B, and Qwen3-8B before and after integrating the Invocation Refiner.

In this evaluation, each upstream model is deployed on a single NVIDIA L20 GPU. The Refiner is similarly deployed on a separate single NVIDIA L20 GPU using the ms-swift framework (Zhao et al., 2024), serving as an API endpoint that upstream models call to request invocation rectification. To ensure optimal inference speed, all models utilize vLLM (Kwon et al., 2023) for acceleration.

Model	Time (minute)	
Llama-3.1-8B-Instruct	76	-
Llama-3.1-8B-Instruct + Refiner	116	1.53×
Qwen3-1.7B	159	-
Qwen3-1.7B + Refiner	217	1.36×
Qwen3-8B	353	-
Qwen3-8B + Refiner	421	1.19×

Table 3: Time consumption on the Bamboogle* benchmark before and after introducing the Refiner.

As shown in Table 3, the introduction of the Refiner results in total time consumption increasing to 1.53×, 1.36×, and 1.19× for Llama-3.1-8B-Instruct, Qwen3-1.7B, and Qwen3-8B, respectively.

Model	Overall Acc.	Δ	Level 1	Level 2	Level 3
Qwen2.5-3B-Instruct	51.59	-	59.65	32.84	36.64
+ Refiner (SFT)	61.31	+18.84%	64.16	62.69	51.91
+ Refiner (DAPO)	**68.84**	**+33.44%**	**73.18**	**67.16**	**56.49**
+ Refiner (SFT + DAPO)	64.49	+25.00%	68.92	62.69	51.91
Qwen2.5-7B-Instruct	62.48	-	70.68	49.25	44.27
+ Refiner (SFT)	67.50	+8.03%	71.18	62.69	58.78
+ Refiner (DAPO)	**73.70**	**+17.96%**	**78.70**	**67.16**	**61.83**
+ Refiner (SFT + DAPO)	71.36	+14.21%	76.69	62.69	59.54
Llama-3.2-3B-Instruct	40.54	-	44.86	29.85	32.82
+ Refiner (SFT)	61.31	+51.23%	64.91	58.21	51.91
+ Refiner (DAPO)	64.15	+58.24%	66.17	**68.66**	55.73
+ Refiner (SFT + DAPO)	**69.18**	**+70.65%**	**71.93**	65.67	**62.60**
Llama-3.1-8B-Instruct	67.17	-	71.93	62.69	54.96
+ Refiner (SFT)	65.33	-2.74%	67.17	61.19	**61.83**
+ Refiner (DAPO)	68.51	+1.99%	71.93	**67.16**	58.78
+ Refiner (SFT + DAPO)	**69.18**	**+2.99%**	**74.44**	**67.16**	54.20
Qwen3-1.7B	60.13	-	61.40	43.28	64.89
+ Refiner (SFT)	66.33	+10.31%	67.17	62.69	65.65
+ Refiner (DAPO)	**70.18**	**+16.71%**	**73.43**	**65.67**	62.60
+ Refiner (SFT + DAPO)	69.35	+15.33%	72.18	58.21	**66.41**
Qwen3-8B	70.85	-	73.43	65.67	65.65
+ Refiner (SFT)	71.52	+0.95%	73.18	70.15	67.18
+ Refiner (DAPO)	**74.37**	**+4.97%**	**77.19**	64.18	**70.99**
+ Refiner (SFT + DAPO)	72.36	+2.13%	74.69	**71.64**	65.65
Hammer2.0-7B	70.69	-	73.68	62.69	**65.65**
+ Refiner (SFT)	63.99	-9.48%	66.92	53.73	60.31
+ Refiner (DAPO)	**73.03**	**+3.31%**	76.94	**68.66**	63.36
+ Refiner (SFT + DAPO)	72.36	+2.36%	**78.20**	67.16	57.25
ToolACE-2-8B	60.13	-	62.91	38.81	62.60
+ Refiner (SFT)	63.65	+5.85%	67.17	47.76	61.07
+ Refiner (DAPO)	69.35	+15.33%	72.43	**62.69**	**63.36**
+ Refiner (SFT + DAPO)	**70.35**	**+17.00%**	**74.94**	**62.69**	60.31

Table 4: Results on the API-Bank benchmark (General-Purpose).

These results indicate that the Refiner incurs a manageable amount of additional time overhead.

Crucially, this overhead can be further mitigated in practical applications. First, the Refiner does not need to be active for every turn of the conversation. It can be invoked selectively via API only when the upstream model produces a tool invocation that requires verification or repair. Second, as a decoupled and independent module, the Refiner benefits from high scalability. It can be deployed across multiple GPUs or servers to process requests in parallel, thereby significantly reducing the latency bottleneck in high-throughput scenarios.

G Performance on the General-Purpose Tool-use Benchmark: API-Bank

Experiment Settings. To further evaluate our Refiner's capability in refining general-purpose tool invocation, we use the API-Bank benchmark. We adopt the evaluation script from ToolRL (Qian et al., 2025), with detailed settings provided in Appendix E.5. The quantitative results are summarized in Table 4. Δ indicates the percentage improvement over a Raw baseline.

Superior Generalization of DAPO-Enhanced Refiners. A critical observation from Table 4 is the performance disparity between the SFT and DAPO training strategies. The SFT-only Refiner exhibits limited generalization. It yields marginal improvements or even degrades performance. This decline stems from the distribution shift between the Refiner's training data and the unseen tools or formats in API-Bank. SFT tends to overfit to the specific formats of the training set, leading to "correction" errors on novel distributions. In contrast, the DAPO-enhanced Refiners (both Pure and Cold Start) demonstrate superior robustness, consistently enhancing performance across all upstream models. Notably, for Hammer2.0-7B, where SFT fails, the Pure DAPO Refiner achieves a 3.31% improvement. This confirms that the RL-based objective encourages the model to learn the fundamental logic

Model	Bamboogle*			NESTFUL		
	Acc.	$\text{Acc}_{Toolname}$	$\text{Acc}_{Parameter}$	Part. Acc.	Full Acc.	Win Rate
Qwen2.5-3B-Instruct						
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 (DAPO)	**48.13**	**79.67**	**79.47**	**0.19**	**0.10**	**0.23**
Qwen2.5-7B-Instruct						
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 (DAPO)	**49.13**	**79.47**	**79.47**	**0.28**	**0.22**	**0.31**
Llama-3.2-3B-Instruct						
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 (DAPO)	**36.53**	**74.73**	**73.80**	**0.21**	**0.12**	**0.17**
Llama-3.1-8B-Instruct						
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 (DAPO)	**60.27**	**72.20**	**72.00**	**0.27**	**0.20**	**0.25**
Qwen3-1.7B						
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 (DAPO)	**39.67**	**68.60**	**66.80**	**0.22**	**0.15**	**0.31**
Qwen3-8B						
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 (DAPO)	**86.60**	**87.13**	**87.13**	**0.29**	**0.22**	**0.41**

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

of instruction following rather than merely mimicking training samples, thereby ensuring better generalization.

Significant Gains on Weaker Models via Format Correction. As shown in Table 4, the improvement is particularly pronounced for weaker upstream 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 acts as a rigorous syntax validator, effectively rectifying these structural failures. Conversely, for stronger or specialized models that inherently maintain better formatting, the Refiner primarily fine-tunes the output, resulting in more modest but steady gains.

Analysis of Semantic Error Correction. Beyond syntax, we observe that the Refiner also demonstrates a strong capability in rectifying semantic errors, such as hallucinated tool names or invalid parameter values. By cross-referencing the upstream model's output with the tool definitions and user query, the Refiner effectively identifies and corrects mismatches that would otherwise lead to execution failures. This ability contributes to the performance gains, verifying that the Refiner serves not just as a syntax validator, but as a robust logic checker that aligns the final tool calls with the intended task requirements.

Impact of Chain-of-Thought Quality on Refinement. 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. Comparing Qwen3-1.7B and Llama-3.1-8B-Instruct highlights this dynamic: although Llama-3.1-8B-Instruct is a larger model with a stronger raw baseline (67.17%), its gains from refinement are relatively small. In contrast, Qwen3-1.7B, despite its smaller size, often generates high-quality CoT traces even when the final invocation is flawed. The Refiner effectively leverages this coherent reasoning path to correct the tool calls, leading to a substantial 16.71% gains. This underscores that the Refiner functions better when it can align its output with a logically sound reasoning trace provided by the upstream model.

H Performance Comparison with an Untrained Refiner

In this section, we conduct an ablation study to isolate the contribution of our specific training methodology. By deploying the off-the-shelf Qwen3-1.7B

Model	Bamboogle*			API-bank	NESTFUL		
	Acc.	$\text{Acc}_{Toolname}$	$\text{Acc}_{Parameter}$		Part. Acc.	Full Acc.	Win Rate
Llama-3.2-3B-Instruct							
CoT	1.60	54.20	45.70	39.03	0.11	0.03	0.04
w/o Overlong Penalty	35.13	73.60	70.67	55.28	0.19	0.10	0.16
Ours, w/ Refiner (DAPO)	**36.53**	**74.73**	**73.8**	**64.15**	**0.21**	**0.12**	**0.17**
Llama-3.1-8B-Instruct							
CoT	43.07	64.80	60.67	67.17	0.21	0.15	0.09
w/o Overlong Penalty	**63.80**	68.13	68.00	57.45	0.23	0.16	0.24
Ours, w/ Refiner (DAPO)	60.27	**72.20**	**72.00**	**68.51**	**0.27**	**0.20**	**0.25**
Qwen3-1.7B							
CoT	16.70	62.27	59.40	60.13	0.15	0.12	0.20
w/o Overlong Penalty	38.67	**70.66**	**69.00**	60.30	0.20	0.14	0.30
Ours, w/ Refiner (DAPO)	**39.67**	68.60	66.80	**70.18**	**0.22**	**0.15**	**0.31**
Qwen3-8B							
CoT	80.33	86.87	86.80	70.85	0.25	0.21	0.33
w/o Overlong Penalty	**86.60**	**87.33**	**87.33**	60.47	0.26	0.20	0.37
Ours, w/ Refiner (DAPO)	**86.60**	87.13	87.13	**74.37**	**0.29**	**0.22**	**0.41**

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

as a "Raw Refiner" without any further training, we disentangle the improvements attributable to our RL training (DAPO) from the inherent reasoning capabilities of the raw model itself.

Following the protocol of our main evaluation, we assess performance on two distinct benchmarks: the modified Bamboogle dataset (Bamboogle*) for the format and content accuracy (Acc_{Full}, $\text{Acc}_{Toolname}$, $\text{Acc}_{Parameter}$), and the NESTFUL benchmark for sequential tool-use logic. The comparative results are detailed in Table 5.

Insufficiency of Untrained Base Models. Quantitative analysis reveals that the inherent capabilities of the base model are insufficient for robust error correction. As shown in Table 5, while the Raw Refiner can offer sporadic improvements on weaker upstream models (e.g., Llama-3.2-3B-Instruct on Bamboogle*), it lacks consistency and often degrades performance on stronger baselines. For instance, on Qwen3-8B, the Raw Refiner causes a significant drop in Bamboogle* accuracy from 80.33% to 56.53%, due to its inability to strictly adhere to the specific tool definitions. Similarly, on NESTFUL, the Raw Refiner struggles to systematically improve sequential logic, yielding marginal or negative gains compared to the CoT baseline.

Critical Role of Specialized RL Optimization. In sharp contrast, our DAPO-enhanced Refiner consistently delivers superior performance across all metrics and upstream models. It comprehensively outperforms the Raw Refiner, achieving substantial gains in invocation format, tool selection, parameter accuracy, and execution order. These results unequivocally demonstrate that the efficacy of our approach is not merely a product of model cascading, but is fundamentally driven by the specialized RL training, which aligns the Refiner with the nuanced requirements of tool invocation correction.

I An Ablation Study of the Overlong Penalty

To validate the necessity of the overlong penalty mechanism within our DAPO training framework, we conduct an ablation study by removing this constraint from the reward function. We hypothesize that without this penalty, the Refiner lacks a "stop condition" for its reasoning process, making it prone to generating excessively verbose outputs that exceed the context window. Specifically, we compare the Refiner trained with the full DAPO objective ("w/ Refiner (DAPO)") and the ablated Refiner ("w/o Overlong Penalty") on the modified Bamboogle dataset (Bamboogle*), the API-Bank benchmark, and the NESTFUL benchmark.

The results, detailed in Table 6, provide strong empirical support for our hypothesis, revealing a distinct performance divergence based on task complexity:

Necessity in Complex Scenarios (API-Bank & NESTFUL). On benchmarks requiring intricate reasoning, the overlong penalty proves indispensable. On API-Bank benchmark, the full Refiner significantly outperforms the ablated Refiner. For instance, on Qwen3-8B, the accuracy drops precip-

Model	NESTFUL			API-bank	Bamboogle*		
	Part. Acc.	Full Acc.	Win Rate		Acc.	$Acc_{Toolname}$	$Acc_{Parameter}$
Qwen2.5-3B-Instruct							
CoT	0.10	0.04	0.13	40.54	0.00	54.27	52.27
w/ Refiner (w/o Simplified Order Data)	0.18	0.09	0.22	65.83	40.86	72.93	72.60
Ours, w/ Refiner (DAPO)	**0.19**	**0.10**	**0.23**	**68.84**	**48.13**	**79.67**	**79.47**
Qwen2.5-7B-Instruct							
CoT	0.25	0.18	0.18	68.34	20.87	60.40	60.40
w/ Refiner (w/o Simplified Order Data)	0.26	0.20	0.28	71.36	21.6	59.80	59.80
Ours, w/ Refiner (DAPO)	**0.28**	**0.22**	**0.31**	**73.70**	**49.13**	**79.47**	**79.47**
Llama-3.2-3B-Instruct							
CoT	0.11	0.03	0.04	39.03	1.60	54.20	45.70
w/ Refiner (w/o Simplified Order Data)	0.19	0.11	0.14	63.17	26.73	59.67	58.00
Ours, w/ Refiner (DAPO)	**0.21**	**0.12**	**0.17**	**64.15**	**36.53**	**74.73**	**73.80**
Llama-3.1-8B-Instruct							
CoT	0.21	0.15	0.09	67.17	43.07	64.80	60.67
w/ Refiner (w/o Simplified Order Data)	**0.28**	**0.20**	**0.28**	**70.18**	58.73	67.00	66.73
Ours, w/ Refiner (DAPO)	0.27	**0.20**	0.25	68.51	**60.27**	**72.20**	**72.00**
Qwen3-1.7B							
CoT	0.15	0.12	0.20	60.13	16.70	62.27	59.40
w/ Refiner (w/o Simplified Order Data)	0.19	**0.15**	0.25	**73.70**	**42.53**	**72.27**	**70.33**
Ours, w/ Refiner (DAPO)	**0.22**	**0.15**	**0.31**	70.18	39.67	68.60	66.80
Qwen3-8B							
CoT	0.25	0.21	0.33	70.85	80.33	86.87	86.80
w/ Refiner (w/o Simplified Order Data)	0.26	0.21	0.35	73.87	83.93	86.40	86.40
Ours, w/ Refiner (DAPO)	**0.29**	**0.22**	**0.41**	**74.37**	**86.60**	**87.13**	**87.13**

Table 7: Ablation study on the simplified nested invocation training data (w/o Simplified Order Data).

itously from 74.37% to 60.47% when the penalty is removed. 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. Similarly, on the NESTFUL benchmark, which demands sequential planning and variable management, the full Refiner consistently achieves higher Win Rates. The ablated model tends to over-analyze intermediate steps, diluting the context and leading to logic errors in the final execution sequence.

Neutral Impact on Simple Scenarios (Bamboogle*). In contrast, on the Bamboogle* benchmark, the impact of the penalty is mixed but generally minimal. As shown in Table 6, the ablated model sometimes matches or marginally surpasses the full model. This can be attributed to the nature of Bamboogle*, which consists of 17 similar subtasks focused on generating search queries from celebrity birth dates or birthplaces (see Section E.4). The similarity of these tasks necessitates a less complex reasoning process, which remains well within the context limits.

Conclusion. 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.

J An Ablation Study of the Simplified Nested Invocation Training Data

The introduction of the simplified nested invocation 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 prin-

ciples of tool coordination rather than overfitting to a specific data schema.

- Finally, this format enables targeted negative sampling. By deliberately placing tools with sequential dependencies into a single concurrent step, we create error cases which teach the model the crucial distinction between parallelizable and sequential operations.

In this section, we conduct an ablation study on the simplified nested invocation training data, investigating the performance change of the Refiner when this data is excluded from the training process. Specifically, we train a variant of the DAPO-enhanced Refiner excluding the simplified nested invocation data. We perform evaluations on the modified Bamboogle dataset (Bamboogle*), API-bank, and NESTFUL benchmarks (with CoT). The results are presented in Table 7.

Enhanced Sequential Reasoning. On the NESTFUL benchmark, which specifically targets the logic of tool sequencing, the full Refiner consistently outperforms the ablated variant across most upstream models. This confirms that the simplified nested invocation data effectively imparts the logic of tool dependencies, translating directly to better performance in multi-step reasoning tasks.

Benefits to General Tool Use. Crucially, the benefits of this data extend beyond sequential tasks to general tool-use benchmarks. On Bamboogle* and API-Bank, the full Refiner demonstrates superior stability and accuracy, particularly when paired with weaker upstream models. As seen with Qwen2.5-3B-Instruct and Llama-3.2-3B-Instruct, the full Refiner significantly widens the performance gap over the ablated version on Bamboogle*. We attribute this to the improved generalization fostered by the simplified nested invocation data. The ablated model, lacking exposure to the structural variance, struggles to correct structurally ambiguous reasoning chains produced by weaker upstream LLMs. In contrast, the full Refiner, having learned a more robust internal representation of tool coordination, proves more resilient in rectifying diverse error patterns.

Conclusion. Ultimately, the inclusion of simplified nested invocation training data leads to a more stable Refiner, enhancing its proficiency in both general tool-use correction and specific tool order correction.

K The Use of AI Assistants

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

L The License for Artifacts

The data sources used in this paper are in the public domain and licensed for research purposes. Their licenses or terms of use can be found in their associated publications, Hugging Face repositories, or GitHub repositories.

M Case Studies

M.1 Correction of Formatting Errors

As shown in Figure 3, Case 1 demonstrates our Refiner's proficiency in fixing formatting errors. Here, the Refiner first validates the semantic correctness of the upstream model's tool invocation but then pinpoint a formatting issue. The error is then resolved through a precise modification targeted solely at the structure.

M.2 Preserving a Correct Tool Invocation

As shown in Figure 4, Case 2 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.

M.3 Correction of Content Errors

As shown in Figure 5, Case 3 demonstrates how our Refiner addresses erroneous content in tool calls. As shown, our Refiner correctly identifies an error in the tool invocation (i.e., in this case, the presence of an extraneous parameter), and accurately rectifies the call.

M.4 Correction of Sequential Errors

As shown in Figure 6, Case 4 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 a helpful multi-turn dialogue assistant capable of leveraging tool calls to solve user tasks and provide structured chat responses.\n\n**Available Tools**\nIn your response, you can use the following tools:\n1. GetOccupationSalary: API for querying the salary of a given occupation.\nParameters: {'occupation': {'type': 'str', 'description': 'The occupation to query.'}}\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```plaintext\n<think> Your thoughts and reasoning </think>\n<tool_call>\n{\"name\": \"Tool name\", \"parameters\": {\"Parameter name\": \"Parameter content\", \"... ...\": \"... ...\"}}\n{\"name\": \"... ...\", \"parameters\": {\"... ...\": \"... ...\", \"... ...\": \"... ...\"}}\n...\n</tool_call>\n<response> AI's final response </response>\n```\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>Calculate the monthly salary after taxes for a Data Scientist</user>\n\n<user> Based on our conversation above, please only make one tool call to solve my need.</user>\n

Ground Truth: : <tool_call>\n{\"name\": \"GetOccupationSalary\", \"parameters\": {\"occupation\": \"Data Scientist\"}}\n</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>\n{\"name\": \"GetOccupationSalary\", \"parameters\": {\"occupation\": \"Data Scientist\"}}\n<response>AI will fetch the salary information for a Data Scientist and then calculate the salary after taxes. Please wait.</response>\n<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>\n{\"name\": \"GetOccupationSalary\", \"parameters\": {\"occupation\": \"Data Scientist\"}}\n</tool_call>

Figure 3: Case study 1, correction of formatting errors.

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**\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```plaintext\n<think> Your thoughts and reasoning </think>\n<tool_call>\n{\"name\": \"Tool name\", \"parameters\": {\"Parameter name\": \"Parameter content\", \"... ...\": \"... ...\"}}\n{\"name\": \"... ...\", \"parameters\": {\"... ...\": \"... ...\", \"... ...\": \"... ...\"}}\n...\n</tool_call>\n<response> AI's final response </response>\n```\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?</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.</response>\n\n<user>The appointment ID is 34567890 and the new date is March 26th with Dr. Lee.</user>\n<response>Alright. I'll modify your appointment now.</response>\n\n<user> Based on our conversation above, please only make one tool call to solve my need.</user>

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

Figure 4: Case study 2, preserving a correct tool invocation.

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**\nIn your response, you can use the following tools:\n1. AccountInfo: API for retrieving and updating user account information.\nParameters: {'username': {'type': 'str', 'description': 'Name of the user.'}, 'password': {'type': 'str', 'description': 'Password of the user.'}}\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```plaintext\n<think> Your thoughts and reasoning </think>\n<tool_call>\n{\"name\": \"Tool name\", \"parameters\": {\"Parameter name\": \"Parameter content\", \"... ...\": \"... ...\"}}\n{\"name\": \"... ...\", \"parameters\": {\"... ...\": \"... ...\", \"... ...\": \"... ...\"}}\n...\n</tool_call>\n<response> AI's final response </response>\n```\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>Update John's personal information and address. Password is 123456. Address is 123 Main St.</user>\n\n<user> Based on our conversation above, please only make one tool call to solve my need.</user>\n

Ground Truth: : <tool_call>\n{\"name\": \"AccountInfo\", \"parameters\": {\"username\": \"John\", \"password\": \"123456\"}}\n</tool_call>

Upstream LLM's Thought: Okay, let's see. The user wants to update John's personal information and address. They provided the password as 123456 and the new address as 123 Main St.\n\nFirst, I need to check which tool to use. The available tool is AccountInfo, which is for retrieving and updating user account info. The parameters required are username, password, and possibly other fields like address.\n\nWait, the parameters listed for AccountInfo are username and password. But the user is mentioning an address. Hmm, maybe the tool allows updating address through the AccountInfo API. Maybe the address is part of the account information. But according to the tool's parameters, the only parameters are username and password. That's confusing. \n\nWait, maybe the tool's parameters are just the username and password for authentication, but the actual update fields (like address) are part of the account info that can be modified. So, the final answer is to call the AccountInfo tool with username John, password 123456, and address 123 Main St. Even though the tool's parameters are listed as only username and password, the user's request requires including the address. Therefore, the tool call should include those parameters.

Upstream LLM's Answer: <tool_call>\n{\"name\": \"AccountInfo\", \"parameters\": {\"username\": \"John\", \"password\": \"123456\", \"address\": \"123 Main St\"}}\n</tool_call>

Refiner's Thought: Okay, let's see. The user wants to update John's personal information and address. The password is 123456, and the address is 123 Main St. The tool-calling instruction provided is using the AccountInfo tool with username John, password 123456, and address 123 Main St. \n\nWait, the AccountInfo tool's parameters are username and password. The address is supposed to be part of the parameters. Let me check the tool's parameters again. The AccountInfo tool's parameters are username and password. The address is not part of the parameters. So the user is trying to include the address in the parameters, but according to the tool's definition, it's not allowed. \n\nSo the correct approach would be to only use the username and password. The address is not part of the parameters for AccountInfo. Therefore, the tool-calling instruction is incorrect because it includes the address in the parameters. The address should be part of the user's request, but the tool doesn't accept it. So the correct tool call would only include username and password. \n\nTherefore, the corrected tool-calling instruction should only include username and password, and the address should be handled elsewhere, maybe in a separate tool or by the user. So the correct instruction would be to call AccountInfo with username John and password 123456, and the address is not included in the parameters. So the original instruction is incorrect because it includes the address in the parameters. The correct one would exclude the address.

Refiner's Answer: <tool_call>\n{\"name\": \"AccountInfo\", \"parameters\": {\"username\": \"John\", \"password\": \"123456\"}}\n</tool_call>

Figure 5: Case study 3, correction of content errors.

The Original System Prompt: You are an expert in composing functions. You are given a question and a set of possible functions.\nBased on the question, you will need to make one or more function/tool calls to achieve the purpose.\nIf 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.\n\nIf you decide to invoke any of the function(s), you MUST put it in the format of \"<think> Your thoughts and reasoning </think>\n [{{\"name\": \"func_name1\", \"arguments\": {{\"argument1\": \"value1\", \"argument2\": \"value2\"}}, \"label\": \"$var_1\"}}, ... (more tool calls as required)]\"\n\nYou SHOULD NOT include any other text in the response.\n\nHere is a list of functions in JSON format that you can invoke.\n\n{FUNCTION_STR}\n\nHere are some examples:\n{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*14 = 56, and for the 10 students is 10*16 = 160. The total age for all 15 students is 15*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*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: Case study 4, correction of sequential errors.