LoopTool: Closing the Data–Training Loop for Robust LLM Tool Calls

Kangning Zhang[1,2*] **Wenxiang Jiao**[2] **Kounianhua Du**[1,2*] **Yong Yu**[1]
Yuan Lu[2] **Weiwen Liu**[1,✉] **Weinan Zhang**[1,✉]
[1]Shanghai Jiao Tong University [2]Xiaohongshu Inc.
{zhangkangning, kounianhuadu, yyu, liuww, wnzhang}@sjtu.edu.cn
wenxiangjiaonju@gmail.com, luyuan3@xiaohongshu.com

Abstract

Augmenting Large Language Models (LLMs) with external tools enables them to execute complex, multi-step tasks. However, tool learning is hampered by the static synthetic data pipelines, where data generation and model training are executed as two separate, non-interactive processes. This approach fails to focus on the model's specific weaknesses adaptively and allows noisy labels to persist, degrading training efficiency. We introduce **LoopTool**, a fully automated, model-aware data evolution framework that closes this loop by tightly integrating data synthesis and model training. LoopTool iteratively evolves both the data and the model through three synergistic modules: (1) *Greedy Capability Probing (GCP)* diagnoses the model's mastered and failed capabilities; (2) *Judgement-Guided Label Verification (JGLV)* uses an open-source judge model to find and correct annotation errors, progressively purifying the dataset; and (3) *Error-Driven Data Expansion (EDDE)* generates new, challenging samples based on identified failures. This closed-loop process is tightly integrated with reinforcement learning training and operates within a cost-efficient, open-source ecosystem, thereby eliminating reliance on costly APIs. Experiments show that LoopTool-8B significantly surpasses its 32B data generator and achieves new state-of-the-art results on the BFCL-v3 and ACEBench benchmarks for its scale. Our work demonstrates that closed-loop, self-refining data pipelines can dramatically enhance the tool-use capabilities of LLMs. The code is accessible in https://github.com/DeepExperience/LoopTool.

1 Introduction

Large Language Models (LLMs) augmented with external tools have become a powerful paradigm for solving complex tasks beyond pure text generation (Qu et al., 2025; Schick et al., 2023; Qin et al., 2023). By invoking APIs, querying databases, and interacting with computational engines, such agents can tackle diverse real-world scenarios (Chen et al., 2025b; Xie et al., 2024; Pan et al., 2025) with high efficiency and adaptability. The development of robust tool-use capabilities, however, hinges on access to accurate, large-scale, and well-aligned training data that matches the model's current competencies (Liu et al., 2025).

A widely adopted approach constructs large-scale tool-calling datasets via automated synthesis pipelines (Qin et al., 2023; Liu et al., 2024; Tang et al., 2023; Liu et al., 2025; Prabhakar et al., 2025), followed by supervised fine-tuning (SFT) or reinforcement learning (Wang et al., 2025; Shao et al., 2024). Despite notable advances, **they almost invariably adopt a static design, wherein data generation and model training are executed as two separate, non-interactive processes**. In such settings, the training data is generated *a priori* without awareness of the evolving state of the model, causing wasted capacity on trivial cases already mastered while leaving harder, underrepresented cases unresolved. Furthermore, the model plays no role in guiding or influencing data generation. This inherent disconnect leads to a persistent mismatch between the model's learning needs and the fixed nature of the available training data, thereby constraining both the efficiency and effectiveness of post-training.

Another major challenge in tool-use data generation is the trade-off between cost-efficiency and data quality. Many pipelines rely on large closed-source models (OpenAI, 2023) for data generation and evaluation. Although these models can produce high-fidelity tool-calling sequences, their use entails high API costs and low generation efficiency, rendering frequent large-scale synthesis impractical. Substituting them with more accessible open-source models often introduces noisy annotations, including incorrect arguments, incomplete function

*This work was done while Kangning Zhang and Kounianhua Du were interns at Xiaohongshu Inc.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 21144–21164
July 2-7, 2026

calls, or outputs misaligned with task requirements. Such errors inject misleading learning signals and may impair model generalization (Liu et al., 2025).

To overcome the limitations of static, costly, and error-prone tool-use data pipelines, we propose *LoopTool*, an automatic, model-aware data evolution framework that couples data synthesis and training in a closed loop. LoopTool begins with an *Automated Tool-Augmented Dialog Generation* stage, where tool specifications are synthesized and combined with multi-agent dialogue generation to produce a diverse seed corpus of realistic tool-oriented conversations, followed by an initial GRPO-based (DeepSeek-AI et al., 2025) post-training round.

Each iteration integrates three synergistic modules. First, *Greedy Capability Probing (GCP)* queries the fine-tuned model on the training corpus via greedy decoding to reveal mastered, borderline, and failure cases, whose predicted tool calls support automated error analysis and focus training on challenging instances. Second, *Judgement-Guided Label Verification (JGLV)* uses the open-source judge Qwen3-32B (Yang et al., 2025) to compare predictions with reference labels, distinguishing genuine model errors from cases where the model output surpasses the original annotation; these "model-better-than-label" examples replace noisy labels and progressively purify the supervision signal. Third, *Error-Driven Data Expansion (EDDE)* turns verified failures into structurally similar yet contextually diverse hard samples that preserve the core functional challenge under varied conditions. Across iterations, LoopTool incorporates corrected annotations, diversified hard examples, and refined seeds into subsequent training rounds, inducing a dynamic curriculum aligned with the model's evolving strengths and concentrating learning on non-trivial, high-value cases while mitigating noisy-label effects.

To balance quality and cost, LoopTool unifies data generation and evaluation within a single open-source model, Qwen3-32B, thereby eliminating reliance on expensive closed-source APIs while maintaining high data quality. Notably, although the final 8B-scale LoopTool model is trained entirely on data generated and judged by Qwen3-32B, it surpasses the 32B generator in tool-use performance, underscoring the amplifying effect of iterative, model-aware data refinement.

In summary, our main contributions are:

- We present **LoopTool**, the first fully automated, model-aware iterative framework that tightly couples reinforcement learning post-training with targeted data synthesis. By continuously diagnosing model weaknesses and generating capability-targeted training data, LoopTool enables dynamic **co-adaptation** of the model and the dataset.
- We propose a closed-loop data refinement and augmentation strategy that purifies labels through comparative judgment (**JGLV**) and transforms verified failures into diverse, high-value training samples (**EDDE**), enhancing tool-use learning without reliance on closed-source models.
- Leveraging fully open-source, self-contained data generation and refinement, our 8B model trained by LoopTool surpasses its 32B generator and achieves state-of-the-art performance on BFCL-v3 (Patil et al., 2025) and ACEBench (Chen et al., 2025a) among models of similar scale.

2 Related Work

Tool-Augmented LLMs. Integrating LLMs with external tools helps mitigate inherent limitations (Qu et al., 2025), enabling API calls (Shen et al., 2023; Qin et al., 2023), knowledge-base access (Lazaridou et al., 2022; Chen et al., 2025b; Zhang et al., 2026; Hao et al., 2026), code execution (Wang et al., 2024; Fu et al., 2026), process reward model (Zheng et al., 2025) and multimodal processing (Hu et al., 2024; Ma et al., 2024; Wang et al., 2026). Early work mainly used supervised fine-tuning (SFT) on human-annotated tool-use data to improve tool selection and argument generation (Schick et al., 2023; Qin et al., 2023; Liu et al., 2024). More recent studies investigate autonomous tool creation and dynamic invocation, aiming to generalize to unseen APIs without predefined schemas. Benchmarks such as tau-bench (Yao et al., 2024; Barres et al., 2025), BFCL (Patil et al., 2025), and ACEBench (Chen et al., 2025a) standardize evaluation of tool selection, argument generation, multi-step reasoning, and multi-turn tool use.

Synthetic Data for Tool Use. The scarcity and cost of high-quality tool-use datasets have driven research into automated synthesis pipelines (Qin et al., 2023; Liu et al., 2025, 2024; Prabhakar et al., 2025; Xu et al., 2025). Methods include

multi-agent simulation (Mitra et al., 2024; Tang et al., 2024), modular task composition (Chen et al., 2025c), and graph-based query–function synthesis (Arcadinho et al., 2024; Yin et al., 2025). Our work builds on this line but differs by introducing a fully automated, model-aware, iterative paradigm in which synthesis is guided by post-training diagnostics and refined via systematic error correction.

Reinforcement Learning for Tool-Use Optimization. Reinforcement learning (RL) increasingly enhances LLM reasoning and decision-making (Ouyang et al., 2022; Rafailov et al., 2024; Meng et al., 2024; Shao et al., 2024). In tool-use settings, GRPO has shown strong performance (Qian et al., 2025; Zhang et al., 2025). We embed RL into an interleaved train–generate loop, enabling the model to iteratively improve through exposure to prior failures and progressively refined supervision. A detailed comparison between LoopTool and recent studies on iterative training (Zweiger et al., 2025; Zhou et al., 2025) is presented in Appendix C.

3 Automated Tool-Augmented Dialog Generation

Before initiating LoopTool, we require a diverse and high-quality *seed dataset* $\mathcal{D}_{\text{seed}}$ to support the first round of post-training. To this end, we introduce an *Automated Tool-augmented Dialog Generation* that synthesizes realistic function-calling interactions by combining curated APIs with simulated multi-agent conversations. While this stage is *not* the core innovation, it establishes the essential foundation for the following iterations.

3.1 Hierarchical Dual-Tree Guided API Synthesis

Our tool set includes real-world APIs collected from public sources (Liu et al., 2025, 2024) and synthetic APIs generated via a *Hierarchical Dual-Tree* method. For each application domain, we construct two complementary hierarchies: (i) a **Context Tree** capturing topical scope and functional granularity, from coarse categories at the root to fine-grained specializations at the leaves; and (ii) a **Constraint Tree** encoding structural and functional requirements for valid APIs, such as naming conventions, parameter types and counts, and output formats. To synthesize an API, we independently sample a leaf path from each tree and merge them into a structured LLM prompt, thereby enforcing both functional intent and structural constraints. Illustrative Context and Constraint Trees are provided in Appendix I.

3.2 Multi-Agent Tool-Use Dialog Generation

Multi-Agent Dialogue Simulation. We construct the seed dataset by simulating tool-use dialogs with four roles. The *Planner Agent* designs coherent dialog flows based on a sampled tool subset and a target number of turns, enabling realistic task decomposition and natural progression toward tool usage. The *User Agent* follows the Planner's outline, issuing requests, refining requirements, and supplying missing information such as parameters. The *Assistant Agent* selects suitable APIs from the assigned subset, infers candidate parameters from context, executes tool calls, or synthesizes user-facing responses. The *Tool Agent* executes tool calls according to the API specifications and generates simulated outputs. The dialog proceeds turn by turn until the predefined length is reached.

Rule-based and LLM-based Verification. All dialogs undergo a two-stage verification pipeline. Rule-based checks validate API syntax, parameter completeness, type consistency, and adherence to schema definitions. LLM-based evaluation, using the open-source Qwen3-32B, assesses each tool call for contextual appropriateness and alignment with user intent. Only dialogs passing both stages are retained in the seed dataset.

4 Iterative Model–Data Co-adaptation

To overcome the limitations of static data generation and support dynamically adaptive model training, we develop an automated iterative framework for tool-augmented LLM learning as shown in Figure 1. LoopTool integrates the GRPO Optimization, Greedy Capability Probing, Judgement-Guided Label Verification, and Error-Driven Data Expansion into a unified closed loop. This iterative cycle enables the model to assess its own capabilities continuously, target its weaknesses, and refine the quality of supervision data.

4.1 GRPO Training for Tool Calling

Data Format. We construct the initial seed tool-calling dialogue dataset $\mathcal{D}_{\text{seed}}$ through the *Automated Tool-Augmented Dialog Generation* in Section 3. Each multi-turn dialog sample is transformed into multiple GRPO training samples, which consist of the tuple: $(\mathcal{T}, c_t, a_t^*)$, where t

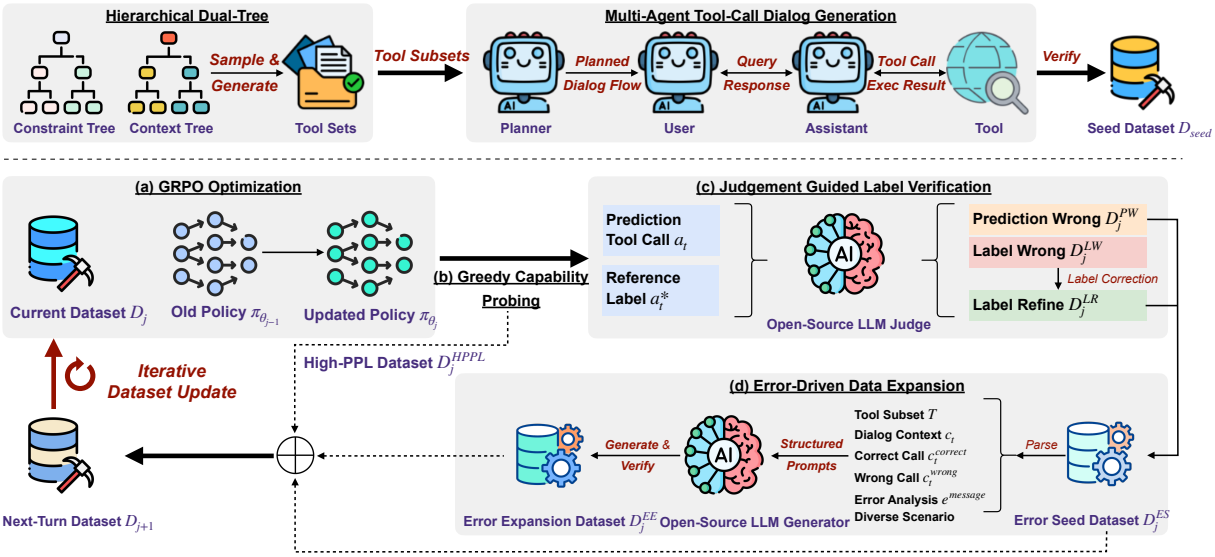

Figure 1: The overall closed-loop automatic pipeline of **LoopTool**, which couples (a) GRPO optimization, (b) Greedy Capability Probing, (c) Judgement-Guided Label Verification, and (d) Error-Driven Data Expansion for iterative tool-use enhancement.

denotes the current turn in the dialogue, $\mathcal{T}$ is the set of available tools at the current step, and c_t means the historical dialogue context, which can be either a single-turn user query or a multi-turn conversation. a_t^* is the tool call step from the conversation corresponding to the last user query. The model's output O_t include two structured components: a reasoning trace wrapped within `<think>` ...`</think>` and the predicted tool invocation a_t inside `<tool_call>` ...`</tool_call>`. A detailed specification of both the single-turn and multi-turn training formats is provided in Appendix J.

Binary Reward Definition. To quantify the quality of model-generated tool calls, we adopt a *Binary Reward* scheme (Qian et al., 2025; Zhang et al., 2025), which serves as a simple yet effective rule-based reward function. For a given context c_t and the model output a_t, the reward is defined as:

$$r(\mathcal{T}, c_t, a_t^*, a_t) = \begin{cases} 1, & \text{ToolMatch}(a_t, a_t^*) \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

Here, $\text{ToolMatch}(\cdot,\cdot)$ denotes a strict AST-level comparison: it returns 1 only when the predicted call and the reference call match in both function name and parsed argument values, and 0 otherwise. This definition is identical to the rule-based reward used during GRPO training and is consistent with the official benchmark scripts.

GRPO Optimization. Given the tool sets $\mathcal{T}$ and historical dialogue c_t, the policy π_θ sample a group of candidate response $\{O_t^1, O_t^2, \ldots, O_t^G\}$ from the old policy $\pi_{\theta_{\text{old}}}$ and their corresponding rewards are $\{r_t^1, r_t^2, \ldots, r_t^G\}$. We optimizes the π_θ through maximizing the following objective:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{\mathcal{D},\{O_t^i\}_{i=1}^G \sim \pi_{\theta_{\text{old}}}} \frac{1}{G} \sum_{i=1}^{G} \Bigg[\min\left(\rho_t^i A_t^i,\ \text{clip}(\rho_t^i, 1-\epsilon, 1+\epsilon) A_t^i\right) - \beta \mathbb{D}_{\text{KL}}(\pi_\theta \| \pi_{\text{old}}) \Bigg],$$
$$\text{where } \rho_t^i = \frac{\pi_\theta(O_t^i \mid c_t, \mathcal{T})}{\pi_{\theta_{\text{old}}}(O_t^i \mid c_t, \mathcal{T})}, \quad A_t^i = \frac{r_t^i - \mu_r}{\sigma_r}. \quad (2)$$

where μ_r and σ_r mean the group-wise mean and standard deviation of rewards. ϵ is the PPO clipping parameter, and β controls the strength of the KL penalty.

4.2 Greedy Capability Probing

GRPO-based post-training often assigns near-zero advantages to both trivially easy and prohibitively hard samples, leading to gradient vanishing (Yu et al., 2025). To improve sample efficiency, we introduce *Greedy Capability Probing* (GCP), an offline diagnostic stage that identifies training instances with substantive learning value.

Given the training set $\mathcal{D}_j$ at iteration j, we perform deterministic greedy decoding with the current policy π_{θ_j} on every instance. For each tool-call sample $(\mathcal{T}, c_t, a_t^*)$, the model generates a prediction $a_t \in O_t$ via greedy search. If $a_t = a_t^*$, the sample is provisionally regarded as *mastered*. Otherwise, the quadruple $(\mathcal{T}, c_t, a_t^*; a_t)$ is forwarded to *Judgement-Guided Label Verification* (JGLV) for correctness assessment. To quantify sample

difficulty, we compute sample-level perplexity:

$$\mathrm{PPL}_{(\mathcal{T},c_t)} = \exp\left(-\frac{1}{L}\sum_{i=1}^{L}\log p_\theta(o_i \mid \mathcal{T}, c_t, o_{1:i-1})\right) \tag{3}$$

where L is the output length and o_i is the i-th output token. Higher perplexity corresponds to lower model confidence and indicates that the sample lies near the decision boundary, making it more informative for further training. In subsequent iterations, GCP selectively carries over a subset of these high-PPL cases $\mathcal{D}_j^{\text{HPPL}}$ into the next iteration.

4.3 Judgement-Guided Label Verification

In each iteration j, for every mismatched case $(\mathcal{T}, c_t, a_t^*; a_t)$ identified by *GCP*, we organize the tool specifications $\mathcal{T}$, dialogue context c_t, reference label a_t^* and model prediction a_t into the open-source Qwen3-32B, which outputs a categorical decision: $y_{\text{judge}} \in \{\texttt{PRED\_WRONG}, \texttt{LABEL\_WRONG}, \texttt{BOTH\_CORRECT}, \texttt{BOTH\_WRONG}\}$ and formatted error analysis e_{message}. Based on the judgment results, we define two key subsets of the evolving dataset: the *Prediction Wrong set* and the *Label Wrong Set*.

$$\begin{aligned}\mathcal{D}_j^{PW} &= \{(\mathcal{T}, c_t, a_t^*; a_t) \mid y_{\text{judge}} = \texttt{PRED\_WRONG}\}\\ \mathcal{D}_j^{LW} &= \{(\mathcal{T}, c_t, a_t^*, a_t) \mid y_{\text{judge}} = \texttt{LABEL\_WRONG}\}\end{aligned} \tag{4}$$

$\mathcal{D}_j^{PW}$ are retained for retraining in the next iteration. For $\mathcal{D}_j^{LW}$, we replace a_t^* with a_t to obtain a refined set $\mathcal{D}_j^{LR}$. For samples classified as BOTH_CORRECT, we retain only those with high-PPL into $\mathcal{D}_j^{\text{HPPL}}$. Samples identified as BOTH_WRONG are directly discarded to avoid propagating noisy supervision. (Refer to Appendix K for judgment prompt and detailed samples.)

Unlike methods that ask LLM to regenerate labels, JGLV casts annotation refinement as a *comparative judgment* problem, where the judge simply decides which of two existing candidates better satisfies the task specification. The evolving policy's own predictions are incorporated as candidates. As training progresses and the policy produces more accurate tool calls, JGLV increasingly replaces incorrect labels with superior model outputs, turning label verification into a self-reinforcing process that continually improves data quality.[1]

4.4 Error-Driven Data Expansion

While GCP and JGLV surface mismatches and correct noisy labels, repeatedly reusing the same errors yields limited gains (see Section 5.4), especially when failures stem from systematic weaknesses. To expand coverage of challenging tool-use scenarios, we introduce *Error-Driven Data Expansion* (EDDE), which converts verified failure cases into structurally similar "hard" samples.

In iteration j, EDDE operates on $\mathcal{D}_j^{ES} = \mathcal{D}_j^{PW} \cup \mathcal{D}_j^{LR}$, where $\mathcal{D}_j^{PW}$ and $\mathcal{D}_j^{LR}$ are the misprediction and label-refined sets from JGLV. For each error seed $(\mathcal{T}, c_t, a_t^*; a_t) \in \mathcal{D}_j^{ES}$, we extract: the tool subset $\mathcal{T}$, dialog context c_t, correct call a_t^{correct}, wrong call a_t^{wrong}, and error analysis e_{message}. A generator LLM is then prompted to produce k new tool-calling samples that preserve the seed's structural difficulty (e.g., argument complexity, multi-step dependencies). To avoid excessive similarity among the augmented samples derived from the same error seed, we additionally introduce scenario diversification constraints $s_{\text{constraint}}$. Specifically, each generation prompt is enriched with varied situational contexts, while preserving the core challenge (Refer to Appendix L for error generation prompt and newly generated samples). All EDDE-generated samples are subjected to the same two-tier validation pipeline outlined in Section 3.2. Samples passing both filters are collected into: $\mathcal{D}_j^{EE} = \mathrm{Verify}\big(\mathrm{Generate}(\mathcal{D}_j^{\text{ES}})\big)$.

Integration into the Iterative Loop. At the end of iteration j, the training dataset for the next round is constructed by merging multiple sources identified during the current iteration:

$$\mathcal{D}_{j+1} = \mathcal{D}_j^{ES} \cup \mathcal{D}_j^{EE} \cup \mathcal{D}_j^{\text{HPPL}} \cup \mathcal{D}_j^{\text{Seed-new}} \tag{5}$$

where $\mathcal{D}_j^{\text{Seed-new}}$ is a small untrained subset from the initial seed dataset $\mathcal{D}_{\text{seed}}$. This merged dataset $\mathcal{D}_{j+1}$ is then used in the subsequent GRPO training round, with the policy π_{θ_j} serving as the initialization. The full iteration pipeline is summarized in the Algorithm 1.

5 Experiments

5.1 Experiment Setup

Benchmarks and Implementation. We evaluate LoopTool by applying our data generation pipeline and pure RL fine-tuning to Qwen3-8B (Yang et al., 2025)[2]. Experiments are conducted on

[1] Appendix E compares Qwen3-32B and GPT-4o as judge models and validates the reliability of Qwen3-32B.

[2] We also test LoopTool on Llama-3.1-8B-Instruct; see Appendix F.

Rank	Overall Acc	Model	Single-Turn		Multi-Turn	Hallucination	
			Non-Live AST Acc	Live Acc	Overall Acc	Relevance	Irrelevance
1	78.45	xLAM-2-70b-fc-r (FC)	88.44	72.95	**75.00**	66.67	78.91
2	76.43	xLAM-2-32b-fc-r (FC)	89.27	74.23	67.12	88.89	76.74
3	**74.93**	**LoopTool-8B (Ours)**	**89.52**	**84.72**	50.88	61.11	87.67
4	73.57	watt-tool-70B (FC)	84.06	77.74	58.87	**94.44**	76.32
5	72.04	xLAM-2-8b-fc-r (FC)	84.40	66.90	69.12	77.78	64.34
6	71.71	GPT-4o-2024-11-20 (FC)	86.81	78.85	50.00	83.33	81.31
7	70.42	GPT-4o-2024-11-20 (Prompt)	87.67	79.88	43.00	72.22	85.36
8	70.32	GPT-4.5-Preview-2025-02-27 (FC)	86.12	79.34	45.38	66.67	83.64
9	69.25	Qwen3-32B (FC)	88.90	77.83	43.12	72.22	75.79
... (Ranks 10–18 omitted for brevity)							
19	66.34	Qwen3-8B (FC)	88.81	78.54	33.00	77.78	79.08
20	65.19	Qwen3-8B (FC, **self-host**)	87.06	78.50	31.25	77.78	78.74

Table 1: Comprehensive evaluation of the BFCL-v3 (last updated on 2025-06-14). FC denotes that the model is tailored for functional calling. The best results in each category are highlighted in bold, while the second-best are underlined.

Model	Normal						Special	Agent	Overall
	Atom	Single-Turn	Multi-Turn	Similar API	Perference	Summary			
Closed-Source Large Language Models									
GPT-4o	90.0	78.0	68.0	80.0	78.0	82.5	92.7	56.0	**81.1**
Gemini-2.5-Pro-05-06	83.7	73.5	61.0	72.0	58.0	75.1	90.7	52.5	**75.8**
Qwen-Max	88.0	75.0	61.0	74.0	82.0	79.7	74.0	60.0	**75.1**
GPT-4o-Mini	84.3	73.5	59.0	74.0	72.0	76.4	76.7	27.5	**68.9**
Gemini-1.5-Pro	82.3	73.0	61.0	74.0	72.0	75.7	77.3	26.0	**68.5**
Open-Source Large Language Models									
Kimi-k2-0711	87.0	78.5	62.0	70.0	74.0	78.9	81.3	65.0	**77.4**
Qwen2.5-Coder-32B-Instruct	86.0	73.5	59.0	76.0	72.0	77.4	80.0	50.0	**73.9**
LoopTool-8B (Ours)	**86.0**	**76.0**	**58.0**	**74.0**	**78.0**	**78.0**	**80.7**	**43.3**	**73.4**
Qwen3-32B	86.3	76.0	57.0	80.0	70.0	77.3	76.0	46.7	**72.2**
ToolACE-2.5-Llama-3.1-8B	87.7	75.5	62.0	74.0	66.0	78.3	76.0	35.9	**71.1**
DeepSeek-V3	88.0	77.5	63.0	76.0	78.0	80.3	72.7	34.0	**71.1**
Qwen2.5-72B-Instruct	81.3	74.5	64.0	76.0	80.0	76.8	74.0	37.5	**70.0**
Qwen3-8B	80.3	68.5	52.0	70.0	58.0	70.9	78.0	34.2	**67.1**

Table 2: Comprehensive evaluation of ACEBench for English Data (last updated on 2025-07-21). LoopTool-8B (Ours) achieves the best result in the 8B scale and surpasses the Qwen3-32B model.

BFCL-v3 (Patil et al., 2025) and ACEBench (Chen et al., 2025a), two comprehensive executable function-call benchmarks; details and metrics are given in Appendix B.1. In BFCL-v3, *Overall Acc* is the official aggregate score across single-turn, multi-turn, and hallucination categories, while in ACEBench, *Overall* is the official aggregate over Normal, Special, and Agent splits. GRPO is implemented with Verl (Sheng et al., 2025), using batch size 128, learning rate 1×10^{-6}, and two epochs per iteration, initializing from the previous checkpoint and resetting optimizer states. The rollout temperature is fixed at 1.0, with entropy and KL coefficients set to 0. We adopt Clip-Higher (Yu et al., 2025), increasing ϵ_{high} from 0.2 to 0.28 to encourage high-entropy, low-probability tokens, and set $k = 4$ in EDDE. Full hyperparameters are provided in Appendix B.2.

Fairness of iterative controls. For all iterative comparisons, including the static baseline and open-loop controls introduced below, we keep the same number of iterations, the same number of gradient updates per iteration, the same initialization-from-previous-checkpoint schedule, and the same optimizer reset policy. The only difference is how the next-round training data is constructed.

5.2 Overall Performance Analysis

Results on BFCL and ACEBench. Using the official evaluation scripts, we report average accuracy across categories (Tables 1 and 2). On both BFCL-v3 and ACEBench, LoopTool-8B achieves SOTA among open-source 8B models and surpasses several larger ones. On BFCL-v3, LoopTool-8B reaches **74.93%** overall accuracy, outperforming Qwen3-8B by **+8.59** points and achieving the best Single-Turn and Live execution accuracy. It also surpasses the 32B-scale Qwen3 used as data generator and judge, demonstrating the effectiveness of our iterative data evolution. On ACEBench, LoopTool-8B attains **73.4%** overall accuracy, **+6.3** points over Qwen3-8B, with consis-

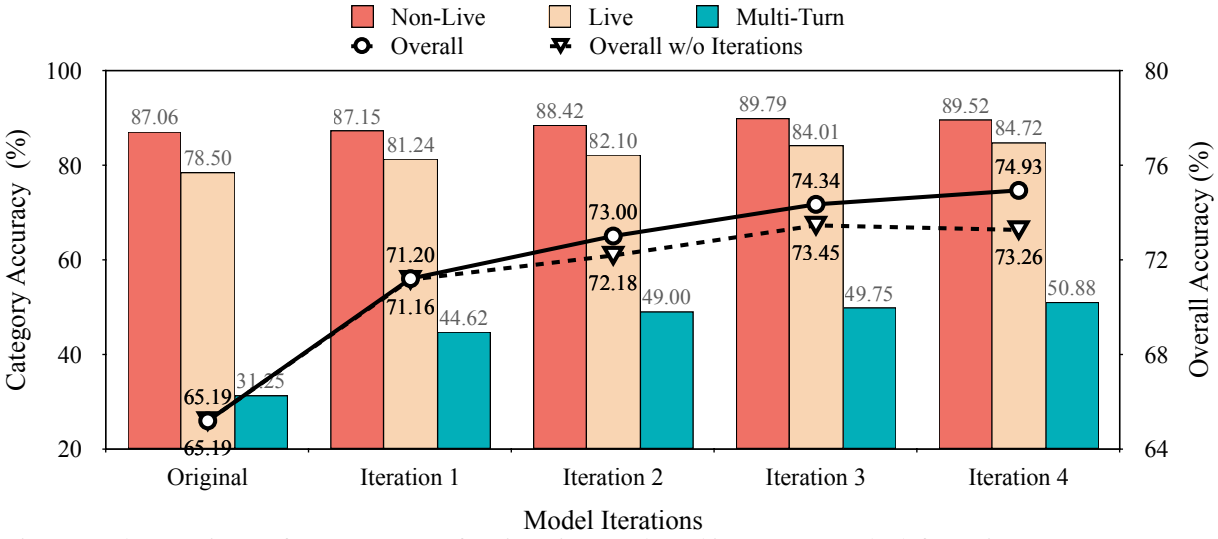

Figure 2: The Iterative Performance across four iterations evaluated in BFCL-v3. The left y-axis represents Category Acc (bar chart), while the right y-axis denotes Overall Acc (line chart). "Overall w/o Iterations" refers to the result obtained under the same number of iteration steps, where we train solely on the initial seed dataset $\mathcal{D}_{\text{seed}}$.

tent gains across categories.

5.3 Iterative Details and Analysis

5.3.1 Closed-Loop Validation

To isolate the contribution of model-aware feedback from simply using more synthetic data, we compare LoopTool with three additional controls: an *open-loop random expansion* baseline that matches LoopTool's data volume but periodically adds randomly generated samples, a variant that uses the same EDDE generator but replaces verified failure seeds with randomly selected ones, and a *static curriculum* that only reorders the fixed seed corpus from easy to hard. LoopTool consistently outperforms all controls, with the largest margin over open-loop expansion at Iteration 4 (**+1.98** points), showing that the gain comes from targeting current failure modes rather than from volume alone. It also surpasses random error seeds by **+1.48** points, indicating that hard-data generation remains ineffective without closed-loop error selection. The full control results are reported in Appendix B.3.

5.3.2 Iterative Dataset Distribution

	# Total	# $\mathcal{D}_j^{\text{ES}}$	# $\mathcal{D}_j^{\text{EE}}$	# $\mathcal{D}_j^{\text{HPPL}}$	# $\mathcal{D}_j^{\text{Seed-new}}$
$\mathcal{D}_1$	18304	0	0	0	18304 (100%)
$\mathcal{D}_2$	18304	1919 (10.48%)	6566 (35.87%)	4187 (22.98%)	5632 (30.77%)
$\mathcal{D}_3$	18304	3386 (18.50%)	8066 (44.07%)	4036 (22.06%)	2816 (15.38%)
$\mathcal{D}_4$	18304	3731 (20.38%)	8169 (44.63%)	4996 (27.29%)	1408 (7.69%)

Table 3: Distribution of samples across iterative datasets in our LoopTool framework.

Configuration	Overall Acc	Non-Live Acc	Live Acc	Multi-Turn Acc
Iteration 1 ($\mathcal{D}_1$)	71.20	87.10	81.34	44.62
Iteration 2 ($\mathcal{D}_2$)	**73.00**	**88.42**	82.10	**49.00**
w/o High-PPL	72.31	88.17	81.59	46.25
w/o JGLV	71.30	87.90	82.05	43.88
Remove EDDE	71.50	88.06	81.47	45.00
HighPPL-Replace	72.50	88.10	**82.36**	47.88
Error-Seed Repetition	72.38	88.40	81.87	46.88
Iteration 3 ($\mathcal{D}_3$)	**74.34**	**89.79**	**84.01**	**49.75**
w/o High-PPL	73.50	89.12	82.79	48.90
w/o JGLV	72.61	89.17	82.59	46.25
Remove EDDE	73.12	88.75	82.45	48.75
HighPPL-Replace	73.28	89.40	83.96	46.88
Error-Seed Repetition	73.43	88.15	83.74	48.38

Table 4: We conduct the corresponding ablation experiments in Iteration 2 and Iteration 3, employing the data variants of $\mathcal{D}_2$ and $\mathcal{D}_3$. Overall accuracy and per-category accuracy are reported.

The initial seed dataset $\mathcal{D}_{\text{seed}}$ includes $28k$ tool call samples. The corpus $\mathcal{D}_{j+1}$ at iteration $j+1$ is constructed from four primary sources as illustrated in Eq (5). $\mathcal{D}_j^{\text{Seed-new}}$ means the untrained new seed samples randomly drawn from the seed dataset $\mathcal{D}_{\text{seed}}$. In each iteration, we gradually reduce the proportion of untrained seed samples, ensuring that each training round incorporates newly generalized queries, while gradually converging on increasingly challenging samples. The detailed data statistics are presented in Table 3. Importantly, LoopTool does not train only on hard failures: $\mathcal{D}_j^{\text{HPPL}}$ and $\mathcal{D}_j^{\text{Seed-new}}$ act as a replay-style buffer that preserves borderline and standard cases, which helps prevent catastrophic forgetting while the loop shifts more capacity toward difficult semantic errors.

Metric	Iter. 1	Iter. 2	Iter. 3	Iter. 4
Avg. PPL	1.123	1.287	1.551	1.638
Multi-turn ratio (%)	50.0	62.7	66.5	70.9
Avg. turns / sample	2.1	3.4	4.8	5.2
Syntactic & schema (%)	9.2	4.5	1.8	0.5
Parameter (%)	62.1	54.3	48.7	42.4
Reasoning & context (%)	38.7	41.2	49.5	57.1

Table 5: Evolution of data complexity and failure types. LoopTool progressively increases training difficulty while shifting the remaining failures from structural issues to reasoning-heavy errors.

5.3.3 Performance Analysis of Iterative Training Framework

We compare the proposed iterative training paradigm with static model training. As shown in Figure 2, the LoopTool framework yields consistent gains in tool-calling accuracy across iterations. Starting from the initial model ("Original"), each iteration exploits closed-loop data evolution to identify and correct model deficiencies, leading to steady improvement. In contrast, the static "Overall w/o Iterations" setting produces markedly smaller gains. Without newly synthesized hard cases or refined labels, the model quickly saturates on the limited supervision and exhausts the information content of $\mathcal{D}_{\text{seed}}$. Performance plateaus by Iteration 2 and declines after Iteration 3, indicating overfitting and an increasing mismatch between the fixed training distribution and the model's evolving inference behavior. The computational resource costs of multiple iterations are summarized in Appendix G. The detailed iterative learning curves are analyzed in Appendix H.

Table 5 shows that the iterative loop changes not only data volume but also data quality. The average perplexity, multi-turn ratio, and average number of turns all increase monotonically, indicating that EDDE progressively injects more difficult scenarios. In parallel, the failure profile shifts from structural issues to semantic ones: syntactic and schema errors drop from 9.2% to 0.5%, while reasoning-and-context errors grow to 57.1% among the remaining failures. This pattern suggests an automatic curriculum in which easy formatting errors are removed early and later rounds concentrate on hard planning, dependency, and state-tracking failures. We further extend training beyond the main four rounds and observe that performance peaks at Iteration 4 and then declines slightly, so we use four rounds in the final system. The detailed saturation results are provided in Appendix B.4, while the full control comparisons remain in Appendix B.3.

5.4 Ablation Study

To assess the contributions of each key component in LoopTool, we perform ablation experiments on BFCL-v3. Specifically, we design the following variants: (i) **w/o High-PPL**: Replace $\mathcal{D}_j^{\text{HPPL}}$ with randomly samples that the model predicted correctly; (ii) **w/o JGLV**: Skip verification and treat all mismatches ($a_t \neq a_t^*$) as model errors; keep original labels without refinement. (iii) **Remove EDDE**: Drop $\mathcal{D}_j^{EE}$ entirely; (iv) **HighPPL-Replace**: Replace $\mathcal{D}_j^{EE}$ with an equal number of high-PPL samples selected via GCP; (v) **Error-Seed Repetition**: Remove $\mathcal{D}_j^{EE}$ and duplicate $\mathcal{D}_j^{ES}$ to match data scale. From the results in Table 4, several key observations can be made:

- **Importance of high-PPL samples**. **w/o High-PPL** lead to consistent accuracy drops, especially in Multi-Turn cases. Even replacing EDDE samples with high-PPL ones (**HighPPL-Replace**) sustains performance close to full configurations, confirming that high-PPL cases—though previously predicted correctly—lie near decision boundaries of current policy and drive further refinement, in line with recent works (Liang et al., 2025; Shang et al., 2025).

- **Necessity of JGLV. w/o JGLV** significantly degrades accuracy, confirming that noisy or erroneous labels can mislead training. Without label refinement, such errors persist and even propagate when used by EDDE to generate variants, exacerbating noise in subsequent iterations.

- **Effectiveness of EDDE** The three **w/o EDDE** variants in both Iteration 2 and Iteration 3 consistently reduce overall accuracy. Further, we compare these variants with the full configuration, evaluating accuracy only on the subset of historically misclassified cases, as shown in Figure 3. The results indicate that merely re-training on the original erroneous seeds is insufficient for the model to effectively handle these difficult cases. By contrast, EDDE synthesizes structurally similar, error-informed variants that preserve the core challenges of the original failures while introducing additional diversity. This targeted augmentation enables the model to learn the relevant patterns more robustly, thereby improving its performance on the original hard seeds.

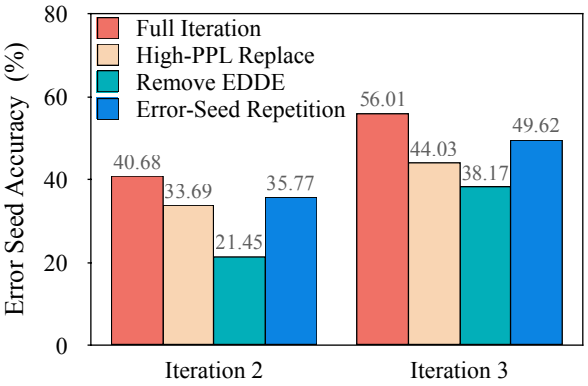

Figure 3: The Prediction Accuracy of Error Seed across iterations.

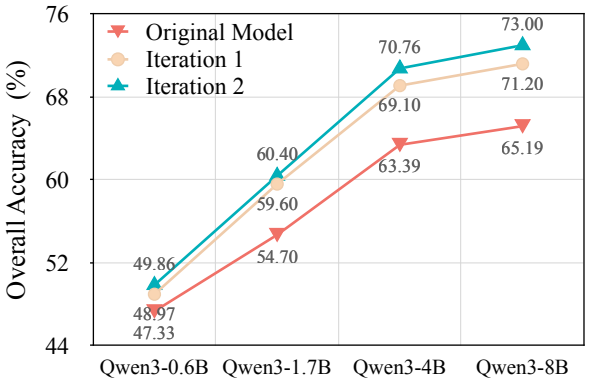

Figure 4: Scaling performance with different model sizes.

5.5 Scaling Performance with Model Size

We evaluate LoopTool across backbone models from 0.6B to 8B parameters, measuring BFCL-v3 accuracy over two training iterations (Figure 4). Larger models consistently achieve higher accuracy in both the initial (*Iteration 1*) and refined (*Iteration 2*) stages, with greater absolute improvements in the second iteration. Specifically, the 0.6B model gains only $+0.70$ points, whereas the 8B model achieves $+1.80$ points. Larger models tend to identify correct trajectories earlier, thereby amplifying the benefits of iterative refinement.

5.6 Generalization Ability Evaluation

Model	Qwen3-8B	LoopTool-8B
MMLU-redux	**87.72**	87.37
IFEval	83.30	**84.70**
LiveCodeBench	42.31	**46.15**
Math-500	91.40	**92.60**
AIME24	60.00	**70.00**
AIME25	56.67	**66.67**

Table 6: Generalization benchmark performance comparison between Qwen3-8B and LoopTool-8B.

Beyond tool-use performance, we assess whether LoopTool-8B preserves or improves generalization to non-tool domains. We compare it with vanilla Qwen3-8B (Yang et al., 2025) on six benchmarks: MMLU-redux (Gema et al., 2025), IFEVAL (Zhou et al., 2023), LiveCodeBench (Jain et al., 2024), Math-500 (Lightman et al., 2023), AIME24, and AIME25 (Art of Problem Solving, 2025), as shown in Table 6. LoopTool-8B matches or surpasses Qwen3-8B on all benchmarks, with clear gains: +1.40 on `IFEval`, +3.84 on `LiveCodeBench`, +1.20 on `Math-500`, and improvements on both `AIME` sets. These results show that the iterative, model-aware data refinement and training paradigm does not overfit to tool-calling, but instead strengthens general reasoning and problem-solving, improving cross-domain generalization.

6 Conclusion

We introduce **LoopTool**, a fully automated, model-aware pipeline that integrates data synthesis, label refinement, and GRPO-based post-training into a closed loop to enhance tool-augmented LLMs. This unified process yields progressively cleaner and more challenging data without dependence on costly closed-source APIs, leveraging a single open-source model for both judgment and generation. Built entirely on open-source models and tools, our 8B-scale LoopTool model, trained solely on data generated and judged by Qwen3-32B, surpasses its 32B teacher and achieves SOTA performance on BFCL-v3 and ACEBench among 8B-scale models, while also improving general abilities on non-tool benchmarks.

Limitations

LoopTool runs as an offline, strictly iterative framework, where data probing, refinement, and expansion for the next round only begin after the previous GRPO training stage completes, limiting responsiveness and preventing fully online or streaming co-evolution. Our experiments are primarily conducted on BFCL-v3 and ACEBench with function-calling tasks and simple binary rewards, leaving open questions about performance in broader tool ecosystems (e.g., highly stateful, domain-specific tools), under richer reward schemes, and under explicit safety and robustness constraints.

Acknowledgments

The work is supported by National Natural Science Foundation of China (62502310, 62322603). The Shanghai Jiao Tong University team is partially supported by Changan Automobile & Chongqing Natural Science Foundation Joint Fund for Innovation and Development (CSTB2023NSCQ-LZX0136).

References

Samuel Arcadinho, David Aparício, and Mariana Almeida. 2024. Automated test generation to evaluate tool-augmented llms as conversational ai agents. *arXiv preprint arXiv:2409.15934*.

Art of Problem Solving. 2025. AIME: AIME problems and solutions, 2025. Accessed: 2025.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. 2025. τ^2-bench: Evaluating conversational agents in a dual-control environment. *Preprint*, arXiv:2506.07982.

Chen Chen, Xinlong Hao, Weiwen Liu, Xu Huang, Xingshan Zeng, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Yuefeng Huang, Wulong Liu, Xinzhi Wang, Defu Lian, Baoqun Yin, Yasheng Wang, and Wu Liu. 2025a. Acebench: Who wins the match point in tool usage? *Preprint*, arXiv:2501.12851.

Mingyang Chen, Tianpeng Li, Haoze Sun, Yijie Zhou, Chenzheng Zhu, Haofen Wang, Jeff Z. Pan, Wen Zhang, Huajun Chen, Fan Yang, Zenan Zhou, and Weipeng Chen. 2025b. Research: Learning to reason with search for llms via reinforcement learning. *Preprint*, arXiv:2503.19470.

Mingyang Chen, Haoze Sun, Tianpeng Li, Fan Yang, Hao Liang, Keer Lu, Bin Cui, Wentao Zhang, Zenan Zhou, and Weipeng Chen. 2025c. Facilitating multi-turn function calling for llms via compositional instruction tuning. *Preprint*, arXiv:2410.12952.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 181 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Lingyue Fu, Hao Guan, Bolun Zhang, Haowei Yuan, Yaoming Zhu, Jun Xu, Zongyu Wang, Lin Qiu, Xunliang Cai, Xuezhi Cao, Weiwen Liu, Weinan Zhang, and Yong Yu. 2026. Corecodebench: Decoupling code intelligence via fine-grained repository-level tasks. *Preprint*, arXiv:2507.05281.

Aryo Pradipta Gema, Joshua Ong Jun Leang, Giwon Hong, Alessio Devoto, Alberto Carlo Maria Mancino, Rohit Saxena, Xuanli He, Yu Zhao, Xiaotang Du, Mohammad Reza Ghasemi Madani, Claire Barale, Robert McHardy, Joshua Harris, Jean Kaddour, Emile Van Krieken, and Pasquale Minervini. 2025. Are we done with MMLU? In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 5069–5096, Albuquerque, New Mexico. Association for Computational Linguistics.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Zhezheng Hao, Hong Wang, Jian Luo, Jianqing Zhang, Yuyan Zhou, Qiang Lin, Can Wang, Hande Dong, and Jiawei Chen. 2026. Recreate: Reasoning and creating domain agents driven by experience. *arXiv preprint arXiv:2601.11100*.

Yushi Hu, Weijia Shi, Xingyu Fu, Dan Roth, Mari Ostendorf, Luke Zettlemoyer, Noah A Smith, and Ranjay Krishna. 2024. Visual sketchpad: Sketching as a visual chain of thought for multimodal language models. *Preprint*, arXiv:2406.09403.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *Preprint*, arXiv:2403.07974.

Angeliki Lazaridou, Elena Gribovskaya, Wojciech Stokowiec, and Nikolai Grigorev. 2022. Internet-augmented language models through few-shot prompting for open-domain question answering. *Preprint*, arXiv:2203.05115.

Xiao Liang, Zhongzhi Li, Yeyun Gong, Yelong Shen, Ying Nian Wu, Zhijiang Guo, and Weizhu Chen. 2025. Beyond pass@1: Self-play with variational problem synthesis sustains rlvr. *Preprint*, arXiv:2508.14029.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. *Preprint*, arXiv:2305.20050.

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

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao,

Zhiwei Liu, Yihao Feng, Rithesh Murthy, Liangwei Yang, Silvio Savarese, Juan Carlos Niebles, Huan Wang, Shelby Heinecke, and Caiming Xiong. 2024. Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets. *Preprint*, arXiv:2406.18518.

Zixian Ma, Weikai Huang, Jieyu Zhang, Tanmay Gupta, and Ranjay Krishna. 2024. m&m's: A benchmark to evaluate tool-use for multi-step multi-modal tasks. *Preprint*, arXiv:2403.11085.

Yu Meng, Mengzhou Xia, and Danqi Chen. 2024. Simpo: Simple preference optimization with a reference-free reward. *Preprint*, arXiv:2405.14734.

Arindam Mitra, Luciano Del Corro, Guoqing Zheng, Shweti Mahajan, Dany Rouhana, Andres Codas, Yadong Lu, Wei ge Chen, Olga Vrousgos, Corby Rosset, Fillipe Silva, Hamed Khanpour, Yash Lara, and Ahmed Awadallah. 2024. Agentinstruct: Toward generative teaching with agentic flows. *Preprint*, arXiv:2407.03502.

OpenAI. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. *Preprint*, arXiv:2203.02155.

Jingyu Pan, Guanglei Zhou, Chen-Chia Chang, Isaac Jacobson, Jiang Hu, and Yiran Chen. 2025. A survey of research in large language models for electronic design automation. *Preprint*, arXiv:2501.09655.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, 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*.

Akshara Prabhakar, Zuxin Liu, Ming Zhu, Jianguo Zhang, Tulika Awalgaonkar, Shiyu Wang, Zhiwei Liu, Haolin Chen, Thai Hoang, Juan Carlos Niebles, Shelby Heinecke, Weiran Yao, Huan Wang, Silvio Savarese, and Caiming Xiong. 2025. Apigen-mt: Agentic pipeline for multi-turn data generation via simulated agent-human interplay. *Preprint*, arXiv:2504.03601.

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

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *Preprint*, arXiv:2307.16789.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-rong Wen. 2025. Tool learning with large language models: a survey. *Frontiers of Computer Science*, 19(8).

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *Preprint*, arXiv:2305.18290.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Preprint*, arXiv:2302.04761.

Ning Shang, Yifei Liu, Yi Zhu, Li Lyna Zhang, Weijiang Xu, Xinyu Guan, Buze Zhang, Bingcheng Dong, Xudong Zhou, Bowen Zhang, Ying Xin, Ziming Miao, Scarlett Li, Fan Yang, and Mao Yang. 2025. rstar2-agent: Agentic reasoning technical report. *Preprint*, arXiv:2508.20722.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *Preprint*, arXiv:2402.03300.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Preprint*, arXiv:2303.17580.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 1279–1297. ACM.

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

Shuo Tang, Xianghe Pang, Zexi Liu, Bohan Tang, Rui Ye, Tian Jin, Xiaowen Dong, Yanfeng Wang, and Siheng Chen. 2024. Synthesizing post-training data for llms through multi-agent simulation. *arXiv preprint arXiv:2410.14251*.

Shijian Wang, Jiarui Jin, Runhao Fu, Zexuan Yan, Xingjian Wang, Mengkang Hu, Eric Wang, Xiaoxi Li, Kangning Zhang, Li Yao, Wenxiang Jiao, Xuelian Cheng, Yuan Lu, and Zongyuan Ge. 2026. Museagent: A multimodal reasoning agent with stateful experiences. *Preprint*, arXiv:2603.27813.

Shuhe Wang, Shengyu Zhang, Jie Zhang, Runyi Hu, Xiaoya Li, Tianwei Zhang, Jiwei Li, Fei Wu, Guoyin Wang, and Eduard Hovy. 2025. Reinforcement learning enhanced llms: A survey. *Preprint*, arXiv:2412.10400.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better llm agents. *Preprint*, arXiv:2402.01030.

Jian Xie, Kai Zhang, Jiangjie Chen, Tinghui Zhu, Renze Lou, Yuandong Tian, Yanghua Xiao, and Yu Su. 2024. Travelplanner: A benchmark for real-world planning with language agents. *Preprint*, arXiv:2402.01622.

Zhangchen Xu, Adriana Meza Soria, Shawn Tan, Anurag Roy, Ashish Sunil Agrawal, Radha Poovendran, and Rameswar Panda. 2025. Toucan: Synthesizing 1.5m tool-agentic data from real-world mcp environments. *Preprint*, arXiv:2510.01179.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

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

Fan Yin, Zifeng Wang, I Hsu, Jun Yan, Ke Jiang, Yanfei Chen, Jindong Gu, Long T Le, Kai-Wei Chang, Chen-Yu Lee, and 1 others. 2025. Magnet: Multi-turn tool-use data synthesis and distillation via graph translation. *arXiv preprint arXiv:2503.07826*.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, and 16 others. 2025. Dapo: An open-source llm reinforcement learning system at scale. *Preprint*, arXiv:2503.14476.

Shaokun Zhang, Yi Dong, Jieyu Zhang, Jan Kautz, Bryan Catanzaro, Andrew Tao, Qingyun Wu, Zhiding Yu, and Guilin Liu. 2025. Nemotron-research-tool-n1: Exploring tool-using language models with reinforced reasoning. *Preprint*, arXiv:2505.00024.

Wenyuan Zhang, Xinghua Zhang, Haiyang Yu, Shuaiyi Nie, Bingli Wu, Juwei Yue, Tingwen Liu, and Yongbin Li. 2026. Expseek: Self-triggered experience seeking for web agents. *Preprint*, arXiv:2601.08605.

Congming Zheng, Jiachen Zhu, Zhuoying Ou, Yuxiang Chen, Kangning Zhang, Rong Shan, Zeyu Zheng, Mengyue Yang, Jianghao Lin, Yong Yu, and Weinan Zhang. 2025. A survey of process reward models: From outcome signals to process supervisions for large language models. *Preprint*, arXiv:2510.08049.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. 2023. Instruction-following evaluation for large language models. *Preprint*, arXiv:2311.07911.

Yifei Zhou, Sergey Levine, Jason E Weston, Xian Li, and Sainbayar Sukhbaatar. 2025. Self-challenging language model agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Adam Zweiger, Jyothish Pari, Han Guo, Ekin Akyürek, Yoon Kim, and Pulkit Agrawal. 2025. Self-adapting language models. *Preprint*, arXiv:2506.10943.

A The use of Large Language Models (LLMs)

In the research process, we employed the open-source Language model as both the Judge Model and the data Generator within our proposed Loop-Tool framework. During manuscript preparation, we used general-purpose LLMs exclusively for grammar checking, phrasing refinement, and clarity improvements in the English text. All conceptual contributions, experiment designs, analyses, and claims in this paper are the responsibility of the authors.

B Experimental Details

B.1 BenchMarks

BFCL The Berkeley Function-Calling Leaderboard (BFCL-V3) (Patil et al., 2025) constitutes a broad and systematic framework designed to rigorously evaluate the function-calling proficiency of large language models (LLMs) across a diverse spectrum of programming languages, application domains, and intricate real-world scenarios. The benchmark encompasses tasks ranging from multiple and parallel function invocations to multi-turn and multi-step function-call interactions. In total, BFCL-V3 comprises 4,951 test instances—3,951 single-turn cases and 1,000 multi-turn samples-carefully curated to reflect dynamic, authentic use cases. The assessment methodology in BFCL incorporates several complementary metrics:

- **Abstract Syntax Tree (AST) Evaluation**: This metric examines the structural correspondence between the abstract syntax tree of the model-generated output, the ground-truth reference, and the formal function specification. It evaluates the correctness of function identification, the inclusion and accuracy of obligatory parameters, and the precision of both parameter types and associated values.

- **Executable Function Evaluation**: Here, the produced API call is executed, and its runtime output is compared directly against the expected ground-truth result, thereby measuring practical execution accuracy.
- **Multi-turn State-based Evaluation**: The evaluation focus on comparing the backend system's state after all function calls are executed at the end of each turn of the conversation. It capture the correctness of model executions that modify the internal state via write and delete.
- **Multi-turn Response-based Evaluation**: It compares the model's execution path against the minimial viable execution result paths as labeled in ground truth. The minimial viable execution result paths refer to a list of function calls that must be executed in order to produce desired response as user requests.
- **Irrelevance**: This criterion quantifies the model's capacity to avoid generating function calls when presented with extraneous or unrelated user queries. The irrelevance score is determined by dividing the number of accurate non-function-call responses by the total test set size.
- **Relevance**: Relevance gauges the model's adeptness at producing function calls that align contextually with the user's query, irrespective of parameter value accuracy. This score is computed as the proportion of appropriate function-call responses within the entire evaluation set.

ACEBench ACEBench (Chen et al., 2025a) is designed to evaluate tool-use capabilities with fine-grained categorization which could be divided into three primary categories: Normal, Special, Agent."Normal" evaluates tool usage in basic scenarios;"Special" evaluates tool usage in situations with ambiguous or incomplete instructions;"Agent" evaluates tool usage through multi-agent interactions to simulate real-world, multi-turn dialogues:

- **Normal Evaluation** compares the model's function call output with the ground truth using AST parsing.
- **Special Evaluation** mainly assesses the ability of model in problem identification. Specifically, the model must: (1) detect and alert missing parameters, (2) accurately locate erroneous parameters, and (3) recognize task-function mismatches.
- **Agent Evaluation** focus on the model's proficiency in utilizing tools during human-agent interactions, employing gpt-4o as a user simulator, incluing End-to-End Accuracy and Process Accuracy.

B.2 Hyper-Parameters

The detailed hyperparameters of GRPO training are illustrated in Table 7.

Category	Hyperparameter
Data Configuration	Train Batch Size: 128
	Validation Batch Size: 128
	Max Prompt Length: 4096
	Max Response Length: 1024
Optimization	Learning Rate: 1e-6
	PPO Mini Batch Size: 128
	KL Loss Used: False
	Entropy Loss Used: False
	Clip Ratio Low: 0.2
	Clip Ratio High: 0.28
Rollout Configuration	Rollout Mini Batch Size: 2
	GPU Memory Utilization: 0.5
	Number of Rollouts: 12
Training & Logging	Save Frequency (epoch): 1
	Test Frequency (epoch): 1

Table 7: Configuration for Iterative GRPO training.

B.3 Closed-loop controls

To verify that the gains of LoopTool do not come merely from additional synthetic data, we evaluate three iterative controls under the same number of iterations, the same number of gradient updates per round, the same checkpoint initialization, and the same optimizer reset policy as the full LoopTool system.

Method	Iter. 2	Iter. 3	Iter. 4
LoopTool (Ours)	**73.00**	**74.34**	**74.93**
Open-loop random expansion	72.04	72.80	72.95
Random error seeds	72.35	73.10	73.45
Static curriculum	72.40	72.55	72.68
Static training on $\mathcal{D}_{seed}$	72.18	73.45	73.26

Table 8: Full iterative control results on BFCL-v3.

B.4 Saturation and iteration dynamics

To determine whether the iterative loop has reached a useful stopping point, we extend LoopTool-8B

beyond the four rounds used in the main experiments. As shown in Table 9, performance peaks at Iteration-4 and then declines slightly in Iterations 5–6 across overall, non-live, live, and multi-turn metrics. This suggests that later rounds provide diminishing returns and may over-concentrate the training distribution on difficult residual cases. We therefore use four iterations as the final setting.

LoopTool-8B	Overall Acc	Non-Live Acc	Live Acc	Multi-Turn Acc
Iteration-4	**74.93**	**89.52**	**84.72**	**50.88**
Iteration-5	74.62	88.12	83.79	50.12
Iteration-6	74.24	87.96	83.65	50.34

Table 9: Extended saturation analysis beyond the main four training rounds.

Table 10 further shows how the loop changes the training distribution over the four effective iterations. The average perplexity, multi-turn ratio, and average number of turns increase monotonically, indicating that EDDE progressively injects more complex samples. Meanwhile, syntactic and schema errors shrink sharply, while reasoning-and-context errors become the dominant remaining failure type. This trend supports the interpretation that LoopTool forms an automatic curriculum: early iterations remove easier structural failures, and later iterations focus more on semantic planning, dependency tracking, and context reasoning.

Metric	Iter. 1	Iter. 2	Iter. 3	Iter. 4
Avg. Perplexity (PPL)	1.123	1.287	1.551	1.638
Multi-turn ratio (%)	50.0	62.7	66.5	70.9
Avg. turns per sample	2.1	3.4	4.8	5.2
Syntactic & schema (%)	9.2	4.5	1.8	0.5
Parameter errors (%)	62.1	54.3	48.7	42.4
Reasoning & context (%)	38.7	41.2	49.5	57.1

Table 10: Evolution of training-data complexity and error distribution across iterations.

C Comparison with the Iterative Algorithm

SEAL (Zweiger et al., 2025) trains LLM to self-adapt through self-edits from the current context, then performs a lightweight inner-loop update on these self-edits to obtain an adapted model. Self-Challenging Agents (SCA) (Zhou et al., 2025) trains tool-using LLM agents using only self-generated tasks. The model first acts as a challenger that explores the environment and proposes executable "Code-as-Test" instances consisting of an instruction, a programmatic verifier, an example solution, and adversarial failure cases, enabling automatic filtering for feasible and non-trivial tasks. The same model then acts as an executor and is optimized with RL using the verifier-derived rewards. LoopTool differs from SEAL and SCA in both the loop granularity and the role of data. It is explicitly dataset-centric rather than instance- or task-centric, using a multi-module closed loop to persistently curate and refine a growing training corpus over many iterations. This design operationalizes cost-effective, fully open-source data improvement at benchmark scale, enabling progressive capability gains that can even allow a smaller trainee model to outperform its larger data generator.

D The Algorithm of LoopTool

We present the complete procedure of our **LoopTool** framework in Algorithm 1 , which couples *GRPO-based post-training*, *Greedy Capability Probing (GCP)*, *Judgement-Guided Label Verification (JGLV)*, and *Error-Driven Data Expansion (EDDE)* into a unified closed-loop data evolution process.

E Qwen3-32B VS GPT-4o as Judge Model

In JGLV, the performance of Qwen3-32B as the evaluator is highly consistent with GPT-4o. We conducted an additional comparison in which Qwen3-32B and GPT-4o were separately used as evaluators in JGLV on all samples in Iterations 2–4, with the result shown in the Table 11. The evaluation output distributions from the two models were highly consistent ($> 97\%$) for every iteration. The result demonstrates that Qwen3-32B has sufficient capability and stability to serve as the judge for JGLV without introducing significant bias or drift in the iterative process. This is likely because the evaluation task, which follows a fixed and explicit instruction format, is relatively simple compared to open-ended generation. Under such well-specified criteria, Qwen3-32B can reliably follow the instructions and thus perform consistently with GPT-4o.

F The performance of LoopTool on Llama

LoopTool is essentially a **model-agnostic** framework for iterative evolution of data and models. To further verify the effectiveness of the LoopTool framework for other model architectures, we conducted additional experiments using Llama-3.1-8B-Instruct (Grattafiori et al., 2024) as

Algorithm 1: LoopTool: Iterative Model-Aware Data Evolution Framework

Input: Initial seed dataset $\mathcal{D}_{\text{seed}}$ from Automated Tool-Augmented Data Construction; Initial model parameters π_{θ_0}.

Output: Final optimized tool-calling model π_{θ_J} after J iterations.

Initialize: $j \leftarrow 1$, $\mathcal{D}_1 \leftarrow \text{Subset}(\mathcal{D}_{\text{seed}})$.
while $j \leq J$ **do**
  `// Step 1: GRPO-based Post-training`
  Train policy $\pi_{\theta_{j-1}}$ on $\mathcal{D}_j$ using GRPO in Eq.(2) with binary reward $r(\cdot)$, obtaining updated parameters π_{θ_j}.
  `// Step 2: Greedy Capability Probing (GCP)`
  foreach $(\mathcal{T}, c_t, a_t^*) \in \mathcal{D}_j$ **do**
    Generate a_t via deterministic greedy decoding from π_{θ_j};
    if $a_t \neq a_t^*$ **then**
      Send $(\mathcal{T}, c_t, a_t^*; a_t)$ to JGLV for evaluation;
    Compute $\text{PPL}_{(\mathcal{T}, c_t)}$ by Eq.(3) and retain high-PPL samples and $a_t = a_t^*$ into $\mathcal{D}_j^{HPPL}$;
  `// Step 3: Judgement-Guided Label Verification (JGLV)`
  foreach *mismatched case* $(\mathcal{T}, c_t, a_t^*; a_t)$ **do**
    Obtain judgement result $y_{\text{judge}} \in \{\texttt{PRED\_WRONG}, \texttt{LABEL\_WRONG}, \texttt{BOTH\_CORRECT}, \texttt{BOTH\_WRONG}\}$ via Qwen3-32B;
    if $y_{\text{judge}} = \texttt{PRED\_WRONG}$ **then**
      Add to $\mathcal{D}_j^{MR}$;
    else if $y_{\text{judge}} = \texttt{LABEL\_WRONG}$ **then**
      Replace $a_t^* \leftarrow a_t$ and add to $\mathcal{D}_j^{LR}$;
    else if $y_{\text{judge}} \in \{\texttt{BOTH\_CORRECT}, \texttt{BOTH\_WRONG}\}$ **then**
      Discard sample;
  `// Step 4: Error-Driven Data Expansion (EDDE)`
  Construct error seed set $\mathcal{D}_j^{ES} \leftarrow \mathcal{D}_j^{PW} \cup \mathcal{D}_j^{LR}$;
  foreach *error seed in* $\mathcal{D}_j^{ES}$ **do**
    Generate k new samples with scenario diversification constraints;
  Validate generated set via rule-based + LLM-based evaluation to obtain $\mathcal{D}_j^{EE}$;
  `// Step 5: Dataset Update for Next Iteration`
  Select untrained subset $\mathcal{D}_j^{\text{Seed-new}} \subset \mathcal{D}_{\text{seed}}$;
  Construct next-round dataset by Eq.(5):

$$\mathcal{D}_{j+1} = \mathcal{D}_j^{ES} \cup \mathcal{D}_j^{EE} \cup \mathcal{D}_j^{HPPL} \cup \mathcal{D}_j^{\text{Seed-new}}$$

  $j \leftarrow j + 1$;
return π_{θ_J}

the main updated policy, while keeping training budget, number of steps, and all LoopTool configurations identical to the Qwen-based runs. We still employ the Qwen3-32B model as both the Generator and the Evaluator. The results are illustrated in the Table 12. Results show clear iterative improvements: overall accuracy rises from 49.72% (Original) to 61.00% (Iteration-3), with substantial metric-level gains such as +8.51 points in Multi-Turn accuracy and +35.37 points in Irrelevance. These results provide direct evidence that the LoopTool framework offers substantial benefits to a different model family (Llama), demonstrating that the improvements are not specific to

Iteration	Model	# Both Correct	# Both Wrong	# Pred Wrong	# Label Wrong	Consistency Rate (%)
Iteration-1	Qwen3-32B	14899	1486	1152	767	97.3
	GPT-4o	14884	1474	1158	788	
Iteration-2	Qwen3-32B	13917	1001	2257	1129	98.8
	GPT-4o	13882	1037	2231	1154	
Iteration-3	Qwen3-32B	13649	924	2734	997	97.6
	GPT-4o	13617	919	2758	1010	

Table 11: Comparison of Qwen3-32B and GPT-4o across iterations as judge models in JGLV.

Model	Overall	Non-Live Acc	Live Acc	Multi-Turn Acc	Irrelevance
Original	49.72	84.48	61.13	9.62	48.46
Iteration-1	53.99	85.46	71.97	11.75	75.29
Iteration-2	56.39	86.17	74.14	14.12	82.55
Iteration-3	61.00	86.73	77.74	18.13	83.93

Table 12: Performance comparison of Llama-3.1-8B-Instruct across iterations.

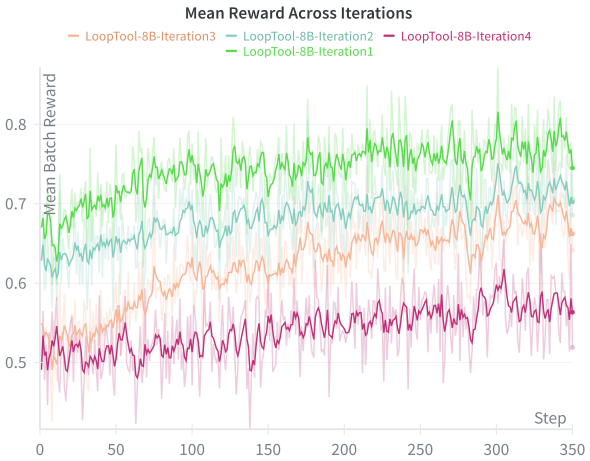

Figure 5: The visualization of reward score across four iterations.

Qwen-based models.

G The Computation Cost across Multiple Iterations

We have quantified actual GPU-hour usage for data iteration and GRPO training, with all experiments run on the same 8 NVIDIA H800 hardware. We deploy the Qwen3-32B model service locally via vLLM to accomplish the JGLV and EDDE modules. The computational cost is equivalently converted into comparable GPU Hours. The result is illustrated in the Table 13. **This shows that the per-iteration data evolution cost is only 6% of the GRPO training cost.** In contrast, without LoopTool's closed-loop data refinement, static-seed training rapidly converges and begins to overfit (as seen by "Overall w/o Iterations" in Figure 2), leaving additional compute under-utilized.

Iteration	GCP	JGLV	EDDE	GRPO Training
Iteration-1	1.05	2.83	6.49	171.96
Iteration-2	1.10	3.15	7.97	181.07
Iteration-3	1.14	3.49	8.07	188.45

Table 13: Computational Cost Comparison. All values are measured in GPU Hours on 8 NVIDIA H800 hardware.

H The Learning Curves in Iterative Learning

Figure 5 presents a detailed depiction of the reward curves for **LoopTool-8B** across multiple iterative training cycles. These curves consistently exhibit two notable characteristics: (1) **There is a stable increase in reward scores across iterations.** In all four training iterations, the model's average binary reward score curves maintain a consistent upward trajectory, free from oscillations or divergence. This stable improvement mirrors the benchmark results shown in Figure 2, thereby confirming a sustained enhancement in the model's tool-use capability; (2) **The reward trends reflect the escalating difficulty of the data.** As illustrated in Figure 5, the overall data complexity progressively increases from one iteration to the next. Although the absolute reward scores tend to be lower in later iterations due to this heightened difficulty, the curves within each training phase still display a steady upward progression, indicating effective learning despite more challenging conditions.

I The example of Hierarchical Dual SubTrees

The example subtrees of the Context Tree and Constraint Tree are illustrated in Figure 6 and Figure 7, respectively.

J The Training Sample for GRPO

The Instruction Prompt used in all GRPO samples is illustrated in Figure 8. The Single-Turn and

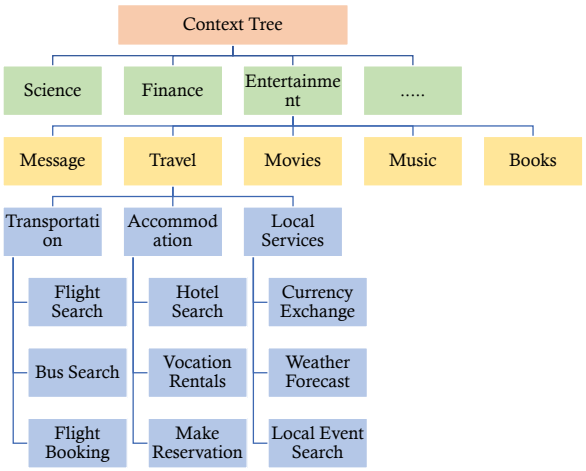

Figure 6: The example subtree of Context Tree.

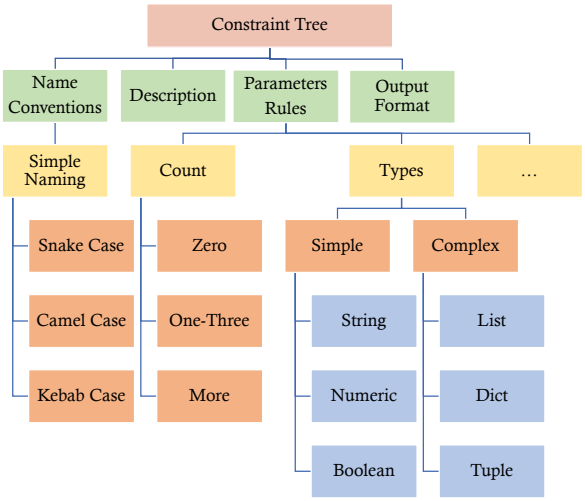

Figure 7: The example subtree of Constraint Tree.

Multi-Turn samples are illustrated in Figure 9 and Figure 10.

K The Label Verification Prompt

The Prompt used in Judge-Guide Label Verification (JGLV) is concluded in Figure 11. Sample examples with $y_{judge} =$ `PRED_WRONG` and $y_{judge} =$ `REF_WRONG` are respectively presented in Figures 12 and 13.

L The Error Generation Prompt and New Error Samples

The system and user prompts for Error-Driven Data Expansion (EDDE) are illustrated in Figures 14 and 15, respecitively. The generated sample case is shown in Figure 16.

The Full Instruction Content

You are an expert in composing functions. You are given a question and a set of possible functions. Based on the question, you will need to make one or more function/tool calls to achieve the purpose.
If none of the functions can be used, point it out. If the given question lacks the parameters required by the function, also point it out. If the result of tool calls has fulfilled the user's request, summarize the answer.
Important Notes

1. When the tool call has fulfilled the user's request, please provide a. concise summary in plain text without extra tool calls. If no tool is suitable, state that explicitly. If the user's input lacks required parameters, ask for clarification
2. During each tool invocation, it is important to carefully examine the corresponding tool's description and constraints. Ensure that the required fields of the tool are strictly satisfied, and that parameter types conform to the definitions. If a function call uses the default parameter value, it is not necessary to specify the value during the call.
3. If the user's request cannot be completed through a one-time function call, or if the parameters of subsequent function calls depend on the results of previous calls, then decompose it into multi-step calls. You only need to return the result of the first step. The use of fictitious parameters or placeholder is strictly prohibited.
4. In multi-turn dialogs, if you encounter an error and the task remains unfinished, retry with more necessary tool calls until completion. Based on the tool feedback, reflect on whether the understanding or selection of the tool is wrong, what tool calling step is missing, and how to achieve the task goal from now on.

The current time is {current_time}.
Tools
You may call one or more functions to assist with the user query.
You are provided with function signatures within <tools></tools> XML tags:
<tools>{tool_sets}</tools>
For each function call, return a json object with function name and arguments within <tool_call></tool_call> XML tags:\n<tool_call>\n{\"name\": <function-name>, \"arguments\": <args-json-object>}\n</tool_call>."

Figure 8: The general instruction prompt employed in all GRPO samples. The variables *current_time* and *tool_sets* are placeholders.

The Single-Turn GRPO Sample

Input: <|im_start|>user\n I am planning a marketing campaign targeting men within age 25-30 in New York. I need to know the estimated size of my target audience. Also, I want to retrieve the media plan with the Campaign ID 'CAM12345'. Can you help with these? <|im_end|>
Label: <tool_call>\n {"name: "getAudienceSize", "arguments": {"age_range": "25-30", "gender": "male", "location": "New York"}} \n </tool_call>
<tool_call>\n {"name": "getMediaPlan", "arguments": {"campaign_id": "CAM12345"}} \n</tool_call>

Figure 9: The example of Single-Turn GRPO samples.

The Multi-Turn GRPO Sample

Input: <|im_start|>user\n I have an image at 'images/satellite_view.png' and I want to understand what the dominant features are in that image. Can you analyze it using a detailed question about its content? <|im_end|>
<|im_start|>assistant\n <tool_call>\n{"name": "pipeline", "arguments": {"image_path": "images/satellite_view.png", "question": "What are the dominant features visible in this satellite view, and how are they distributed across the image?"}}\n</tool_call> <|im_end|>
<|im_start|>user\n <tool_response>\n{"caption": "The satellite view reveals a diverse landscape dominated by dense forest regions, intersected by rivers flowing in a northwesterly direction. Urban clusters are sparsely distributed along the riverbanks, with major infrastructure visible in the central part of the image. Open land patches and agricultural fields are scattered towards the southern quadrant."}\n</tool_response> <|im_end|>
<|im_start|>assistant\n The satellite image analysis … <|im_end|>
<|im_start|>user\n I'd like to calculate the area under the curve for the polynomial function defined by the coefficients [3, -4, 1] between the limits of 1 and 5. Also, can you search for the book 'Calculus: Early Transcendentals' in the library located in New York?" <|im_end|>

Label: <tool_call>\n {"name": "mathematics.calculate_area_under_curve", "arguments": {"polynomial": [3, -4, 1], "limits": [1, 5]}} \n</tool_call>

<tool_call>\n {"name": "library.search_book", "arguments": {"book_name": "Calculus: Early Transcendentals", "city": "New York"}} \n</tool_call>

Figure 10: The example of Multi-Turn GRPO samples.

The Prompt For Judgement-Guided Label Verification (JGLV)

System: You are an expert evaluator specialized in assessing function call responses. Your task is to compare two different function call responses and determine their correctness. Here are the available tools in the conversation:
<tools> {tool_str}\n </tools>
Evaluation Criteria:
1. Correctness: Whether the function calls properly address the user's request
2. Parameter Accuracy: Whether all parameters are correct and appropriate
3. Function Selection: Whether the chosen functions are suitable for the task
4. Completeness: Whether the response fully satisfies the user's needs.
Please provide objective and thorough evaluations based on these criteria.

User: ## **Task**
Please evaluate two function call responses for the following conversation. The time of the conversation is **{date}.**

Original Conversation:
{conversation_text}

Responses to Compare
Response 1
{response1}

Response 2
{response2}

Output Format
Strictly respond with the following formats (no additional text):
"[RESPONSE1_INCORRECT/RESPONSE2_INCORRECT/BOTH_CORRECT/BOTH_INCORRECT]
Error Analysis: [Only briefly describe the cause of the incorrect response, without comparing it to the correct response - max 2 sentences] Correct Approach: [Explain the right way to handle this task - max 2 sentences]."

The analysis should be brief and focus on the key differentiating factor.

Figure 11: The Prompt used in Judge-Guide Label Verification for Judgement Model. The red text corresponds to variables that are placeholders.

The PRED_WRONG Example Identified by JGLV

Input: I have a tweet in English that I want to analyze for sentiment. Also, I have a number 1456 that I want to spell out in French. Additionally, I want to check the metadata of my company's website URL and lastly, I need a list of available spell-check rule sets for Dutch. Here is my tweet: "I just love the holidays! Best time of the year!\". And my company's website URL is "https://examplecompany.com".

Label (Correct):
<tool_call>\n{"name": "Text Sentiment Analysis", "arguments": {"text": "I just love the holidays\"SpellOut\", "arguments": {"data": 1456, "lang": "fr", "ruleset":! Best time of the year!"}}\n</tool_call>
<tool_call>\n{\"name\": \"SpellOut\", \"arguments\": {\"data\": 1456, \"lang\": \"fr\", \"ruleset\": \"cardinal\"}}\n</tool_call>
<tool_call>\n{"name": "parse", "arguments": {"url": "https://examplecompany.com"}}\n</tool_call>
<tool_call>\n{"name": "Get Rule Sets", "arguments": {"lang": "nl"}}\n</tool_call>

Response (Wrong):
<tool_call>\n{"name": "Text Sentiment Analysis", "arguments": {"text": "I just love the holidays\"SpellOut\", "arguments": {"data": 1456, "lang": "fr", "ruleset":! Best time of the year!"}}\n</tool_call>
<tool_call>\n{"name": "Get Rule Sets", "arguments": {"lang": "fr"}}\n</tool_call>
<tool_call>\n{"name": "parse", "arguments": {"url": "https://examplecompany.com"}}\n</tool_call>
<tool_call>\n{"name": "Get Rule Sets", "arguments": {"lang": "nl"}}\n</tool_call>

Error Message (Generated From Judge Model):
Error Analysis: Response 2 incorrectly uses "Get Rule Sets" with `lang: "fr"` (French) instead of `lang: "nl"` (Dutch) for the Dutch spell-check rule sets request.
Correct Approach: Use "SpellOut" with `lang: "fr"` for the French number spelling and "Get Rule Sets" with `lang: "nl" to retrieve Dutch rule sets as specified.

Figure 12: The example with y_{judge} = PRED_WRONG identified by JGLV.

The REF_WRONG Example Identified by JGLV

Input: Can you help me calculate the fluctuation in the exchange rate of the Euro against the Dollar between January 1, 2020, and January 1, 2021? Also find the fluctuation for the Pound against the Dollar for the same time period.

Label (Wrong):
<tool_call>\n{\"name\": \"Fluctuation Endpoint\", \"arguments\": {\"start_string\": \"2020-01-01\", \"end_string\": \"2021-01-01\", \"base\": \"usd\", \"symbols\": \"eur\"}}\n</tool_call>
<tool_call>\n{\"name\": \"Fluctuation Endpoint\", \"arguments\": {\"start_string\": \"2020-01-01\", \"end_string\": \"2021-01-01\", \"base\": \"usd\", \"symbols\": \"gbp\"}}\n</tool_call>

Response (Correct):
<tool_call>\n{\"name\": \"Fluctuation Endpoint\", \"arguments\": {\"start_string\": \"2020-01-01\", \"end_string\": \"2021-01-01\", \"base\": \"eur\", \"symbols\": \"usd\"}}\n</tool_call>
<tool_call>\n{\"name\": \"Fluctuation Endpoint\", \"arguments\": {\"start_string\": \"2020-01-01\", \"end_string\": \"2021-01-01\", \"base\": \"gbp\", \"symbols\": \"usd\"}}\n</tool_call>

Error Message (Generated From Judge Model):
Error Analysis: Response 1 incorrectly sets USD as the base currency and EUR/GBP as symbols, which calculates USD's fluctuation against EUR/GBP instead of EUR/GBP's fluctuation against USD. The base and symbols parameters were reversed.

Correct Approach: Use EUR and GBP as the base currencies with USD as the symbol in separate calls to measure their respective fluctuations against the Dollar.

Figure 13: The example with y_{judge} = REF_WRONG identified by JGLV.

The System Prompt for Error-Driven Data Expansion (EDDE)

System: You are an expert data engineer specialized in creating high-quality training samples for tool-calling language models. Your mission is to generate NEW training samples that help models learn correct tool usage patterns.
Context Information

Current Date: {current_date}
Available Tools:
<tools> {tools_content} </tools>

Your Task
You will analyze an error case and create a completely NEW sample that:

1. Demonstrates CORRECT tool usage in a similar scenario
2. Uses different parameters (names, amounts, symbols, etc.)
3. Maintains similar complexity and potential error patterns
4. Follows exact format requirements

Critical Requirements

1. Generate a realistic but DIFFERENT scenario;
2. Show the CORRECT approach to handle such requests;
3. Use proper conversation format with role markers.
4. Output format must be: INPUT: [content] \n OUTPUT: [content];
5. INPUT could be a single turn or multiple turn request; OUTPUT must use <tool_call>\n...\n</tool_call> format
6. NO additional explanations or text outside the INPUT/OUTPUT sections

Figure 14: The system prompt for Error-Driven Data Expansion (EDDE).

The User Prompt for Error-Driven Data Expansion (EDDE)

User: According to the original sample, generate a NEW training sample:

Original Sample: **{converted_input}**
Correct Response: **{correct_response}**
Incorrect Response: **{incorrect_response}**
Error Analysis: **{error_analysis}**

Format Requirements
Your response must ONLY contain:
INPUT: [New conversation with proper role markers like USER, ASSISTANT]
OUTPUT: [Correct tool calls in <tool_call>\n...\n</tool_call> format]

Generation Instructions
Create a NEW sample that addresses the same type of error but with maximum diversity:
- Apply the scenario focus: **{scenario['focus']}**
- Follow parameter strategy: **{scenario['parameters']}**
- The new sample should help the model recognize the error pattern from the original case and learn the correct approach for similar scenarios
- The correct invocation of new samples does not need to use exactly the same tools as the original samples, and can further involve more complex user requests.

Format Requirements

Your response must ONLY contain:

INPUT: [New conversation with proper role markers like USER, ASSISTANT]
OUTPUT: [Correct tool calls in <tool_call>\n...\n</tool_call> format]

Some Output Examples:

{sample_1}

{sample_2}
...

Figure 15: The user prompt for Error-Driven Data Expansion (EDDE).

The Generated Single-Turn GRPO Sample from EDDE

Original Sample

Input: Log me in with username \"Customer Support\" and password \"P@ssw0rd2023\". Then, show me all tickets I've created

Label (Correct): <tool_call>\n{"name": "ticket_login", "arguments": {"username": "Customer Support", "password": "P@ssw0rd2023"}}\n</tool_call>\n<tool_call>\n{"name": "get_user_tickets", "arguments": {}}\n</tool_call>"

Response (Wrong): <tool_call>\n{"name": "ticket_login", "arguments": {"username": "Customer Support", "password": "P@ssw0rd2023\"}} \n</tool_call>

New Generated Sample

Input: Please sign me in with username "jane.doe" and password "Secure#2024". After that, show me any tickets I have that are currently marked as "open".

Label: <tool_call>\n{"name": "ticket_login", "arguments": {"username": "jane.doe", "password": "Secure#2024"}}\n</tool_call>\n<tool_call>\n{"name": "get_user_tickets", "arguments": {"status": "open"}}\n</tool_call>

Figure 16: The new sample generated by EDDE according to the error in the model response.