

PRFC: Learning Progressive Recovery from Function Calling Errors for Tool-Using LLMs

Beibei Yu, Tao Shen, Ling Chen

AAII, University of Technology Sydney, NSW, Australia
Beibei.Yu@student.uts.edu.au, Tao.Shen@uts.edu.au,
Ling.Chen@uts.edu.au

Abstract

Tool-using large language models (LLMs) can make multiple attempts to complete a task, using execution feedback from failed tool calls to identify errors and revise subsequent actions. However, their recovery performance often plateaus in later attempts, as models may repeat similar tool-call errors instead of making effective corrections. To address this limitation, we introduce PRFC, a reinforcement learning method that models error recovery as a sequential decision-making process. PRFC uses a progress-aware reward that assigns credit only when a later attempt improves upon previous failures, encouraging steady improvement across the recovery trajectory. Across four benchmarks with a Qwen3-4B backbone, PRFC improves multi-turn recovery on BFCL-v3 by an absolute 6.4% and achieves the highest mean score across three single-step tasks (API-Bank, T-Eval, and Nestful). These results show that modeling progress across attempts provides an effective learning signal for multi-attempt tool-use recovery. These results suggest that explicitly modeling progress across attempts provides a useful learning signal for tool-use recovery.

1 Introduction

Large language models (LLMs) have demonstrated remarkable capacity as general-purpose reasoning engines, achieving strong performance across a wide range of tasks (Li et al., 2025b). However, many real-world problems cannot be solved through pure text generation alone. To address this limitation, recent work has equipped LLMs with tool-using capabilities, enabling them to interact with external environments including web services, databases, simulators, and domain-specific software (Qin et al., 2024; Liu et al., 2024, 2025). This paradigm extends LLMs from passive sequence predictors into agentic systems. Furthermore, tool use is especially critical for smaller LLMs, whose

limited parametric knowledge must be complemented by external tools (Li et al., 2026; Shen et al., 2024; Erdogan et al., 2024).

A defining characteristic of tool-using agents is that they operate in an interactive feedback loop: each action produces immediate feedback from the tool, which can inform subsequent reasoning and decision making (Zhang et al., 2025b; Dong et al., 2026). In particular, error feedback is central to recovery in tool use. By exposing mismatches between expected and actual outcomes, execution errors provide rich contextual cues for diagnosing failure modes and guiding targeted repairs (Madaan et al., 2023; Shinn et al., 2023).

Despite recent advances, current LLM-based tool-using agents still struggle to effectively leverage tool error feedback for robust recovery. In practice, tool feedback is highly heterogeneous across tools and environments: some tools return structured and informative error messages, while others provide sparse, ambiguous, or even misleading signals (Huang et al., 2025; Zhang et al., 2025b). This variability, combined with incomplete or underspecified feedback, makes it difficult for models to recover from prior failures. Successful recovery, therefore, requires directional exploration and iterative revision across repair attempts.

However, many existing approaches evaluate each repair attempt primarily as an isolated success or failure conditioned on a growing prompt, rather than as part of a directionally improving process. They treat recovery as trial and error, either by imitating error-fix patterns (Polyakov et al., 2025; Cui et al., 2025; Sun et al., 2024) or by optimizing mainly for terminal outcomes (Qian et al., 2025; Yin et al., 2025). Thus, agents frequently fail to extract actionable guidance from prior failures, leading to brittle behaviors in which they repeat the same incomplete call, prematurely conclude that the task cannot be completed, or modify irrelevant arguments rather than the error indicated by the

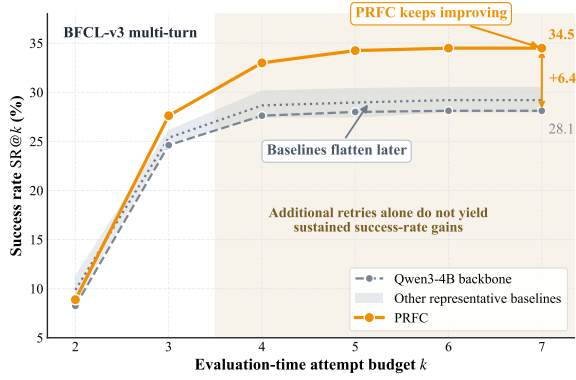


Figure 1: BFCL-v3 multi-turn recovery curves, where $SR@k$ denotes the success rate of tasks solved within at most k repair attempts. Existing methods improve mainly in the first few repair attempts and then saturate, showing that extra retries alone do not yield sustained recovery. PRFC continues improving and reaches the highest $SR@k$.

feedback. Figure 1 further corroborates this limitation. Performance gains concentrate in the first few repair attempts and then plateau, indicating that additional retries alone do not yield sustained recovery. This raises a central question: *how can a model turn error experience into progressive recovery rather than blind retries?*

To address this challenge, we propose PRFC, a reinforcement learning approach that trains tool-using models to improve error recovery from their own trajectories. Rather than treating recovery attempts as independent retries, PRFC optimizes over single-turn trajectories in which the model makes a sequence of repair attempts for the same query. This trajectory-level formulation requires a reward signal that can distinguish productive repairs from repeated failures. We therefore introduce a fine-grained attempt scoring scheme that evaluates the quality of each attempt and provides a local learning signal. However, attempt quality alone does not indicate whether the model is improving across successive repairs. PRFC further compares each attempt with previous failures and assigns credit only when it represents meaningful progress.

PRFC therefore goes beyond assessing attempts in isolation by explicitly modeling cross-attempt progress over the trajectory. Specifically, we add a progress-based shaping signal that rewards a later attempt only when it improves on the best prior failed attempt, as measured by gains in per-attempt scores along the same trajectory. This design encourages sustained recovery rather than diminishing-return retries, guiding the agent toward more targeted repairs while discouraging

repetitive or otherwise unproductive behavior. By explicitly rewarding improvement across attempts, PRFC transforms recovery from unguided trial-and-error into a learnable policy that internalizes persistence and directional exploration.

Finally, we evaluate PRFC on BFCL-v3 and three additional single-step tool-use benchmarks. Despite using only $\sim 1.4K$ training examples, PRFC substantially improves multi-turn recovery on BFCL-v3, increasing task success by +6.4% absolute over the Qwen3-4B backbone while also achieving the best recovery efficiency. On API-Bank, T-Eval, and Nestful, PRFC preserves or modestly improves performance, achieving the highest mean score across the three benchmarks. These results suggest that learning targeted and productive repair strategies is particularly beneficial in interactive settings while preserving general tool-use quality on the additional single-step benchmarks.

2 Related Work

2.1 RL for Tool-Using LLMs

Recent tool-using LLMs increasingly rely on reinforcement learning (RL) to learn from interaction outcomes, moving beyond supervised traces (Li et al., 2025a; Singh et al., 2025; Wei et al., 2025; Qiao et al., 2024). Within this line of research, GRPO-style and related policy optimization methods (Zhang et al., 2025a,c) have been widely adopted. These methods typically optimize terminal rewards over sampled trajectories. To alleviate reward sparsity, subsequent work has explored more fine-grained reward formulations. For example, Qian et al. (2025) propose decomposable scoring schemes that evaluate tool calls along multiple dimensions, thereby providing richer supervision signals. Other studies extend RL-based tool use to multi-turn or conversational function calling (Zhao et al., 2025; Tan et al., 2025; Zeng et al., 2026). Despite these advances, most RL-based tool-using LLMs remain optimized either for final task completion or for the quality of isolated tool calls, leaving cross-attempt progress unmodeled.

2.2 Error Recovery from Feedback in LLMs

Prior work on error recovery in reasoning and tool-use settings has largely focused on inference-time reflection or revision (Yao et al., 2022; Madaan et al., 2023), as well as external verifiers or critics that identify errors and trigger correction (Shinn et al., 2023). However, these paradigms depend

on explicit feedback or well-defined correctness signals. When such signals are sparse, ambiguous, or unavailable, which is common in real-world tool deployment, models struggle to reliably detect failures (Kamoi et al., 2024; Agarwal et al., 2019).

More recent training-based approaches internalize self-correction from execution feedback, either by enhancing tool use or instruction following (Qu et al., 2025; Yuan et al., 2025; Shi et al., 2025; Gou et al., 2024), or by extracting correction heuristics directly from error-fix trajectories (Zhang et al., 2025b; Polyakov et al., 2025). A related line of research bridges imitation learning and reinforcement learning by encouraging agents to explore suboptimal actions and learn from the resulting states without relying on explicit rewards (Huang et al., 2025).

Although these approaches improve robustness, they lack an explicit mechanism to evaluate whether subsequent repairs improve upon prior failures. Tracking this directional progress is crucial, as successful recovery under tool feedback requires advancing beyond previous errors rather than merely sustaining the interaction. PRFC addresses this limitation by formalizing error recovery as a cross-attempt learning objective that explicitly rewards such progress.

3 Methodology

PRFC trains agents to recover from tool errors by optimizing multi-attempt recovery trajectories rather than isolated retries. It constructs feedback-guided rollouts to expose recovery dynamics (§3.1), assigns progress-aware rewards to distinguish productive repairs from stagnant attempts (§3.2), and optimizes the LLM over trajectory returns to reinforce efficient tool use (§3.3).

3.1 Feedback-Guided Multi-Attempt Rollouts

To expose recovery dynamics to the optimizer, we construct training rollouts in which the model can make multiple repair attempts within each user turn. An episode consists of one or more user turns $Q = (q_1, \dots, q_U)$, with single-turn episodes as the special case $U = 1$. A *turn* corresponds to the current user request, and *repair attempts* are the sequential retries made within a turn in response to tool feedback. After each attempt, the model observes the execution result and updates its next action until the request is resolved or the maximum attempt horizon is reached.

Turn-level interaction. Within turn u , the model makes up to T_u attempts. At attempt $t \in \{1, \dots, T_u\}$, the policy π_θ generates an action $a_{u,t} = (a_{u,t}^{\text{think}}, a_{u,t}^{\text{tool}})$ conditioned on the preceding user turns and the interaction history accumulated up to that point, denoted by $h_{u,t-1}$. Each action contains a reasoning trace $a_{u,t}^{\text{think}}$ and a set of executable tool calls $a_{u,t}^{\text{tool}} = \{c_1, \dots, c_n\}$. The executor $\mathcal{E}$ then returns an observation $o_{u,t}$, which is appended as textual feedback to form the updated history, $h_{u,t} = h_{u,t-1} \oplus (a_{u,t}, o_{u,t})$. The next attempt is then conditioned on this updated history.

Trajectory distribution. This process induces a recovery trajectory $\tau = (\tau_1, \dots, \tau_U)$, where $\tau_u = (q_u, a_{u,1}, o_{u,1}, \dots, a_{u,T_u}, o_{u,T_u})$. The policy therefore defines a distribution over episode-level trajectories as

$$\pi_\theta(\tau \mid Q) = \prod_{u=1}^U \prod_{t=1}^{T_u} \pi_\theta(a_{u,t} \mid Q_{\leq u}, h_{u,t-1}). \quad (1)$$

Because training uses the single-turn setting, we drop the turn index below and write q , a_t , h_t , and T for the corresponding single-turn quantities.

3.2 Progress-Aware Reward

Given a recovery trajectory, the primary learning challenge lies in transforming intermediate improvements into a reward signal that informs the agent of viable repair directions. Providing rewards exclusively for final success results in sparse supervision and fails to distinguish productive revisions from repetitive or misguided attempts. Therefore, we use two complementary signals: a local repair score that measures the quality of each individual attempt in isolation, and a progress-shaping term that rewards an attempt only when it improves upon the best performance observed previously within the same trajectory.

Fine-grained local repair scoring. Many failed tool calls are only partially incorrect; for instance, an agent might select the correct tool but omit a parameter, or provide valid argument keys with an invalid value. Assigning a score of zero to all such attempts would eliminate the directional signal necessary for targeted repair (Wang et al., 2026; Feng et al., 2026). Following (Qian et al., 2025), we decompose each tool call into semantically meaningful components and assign partial credit for correct tool selection, key overlap, and matching parameter values. The resulting score

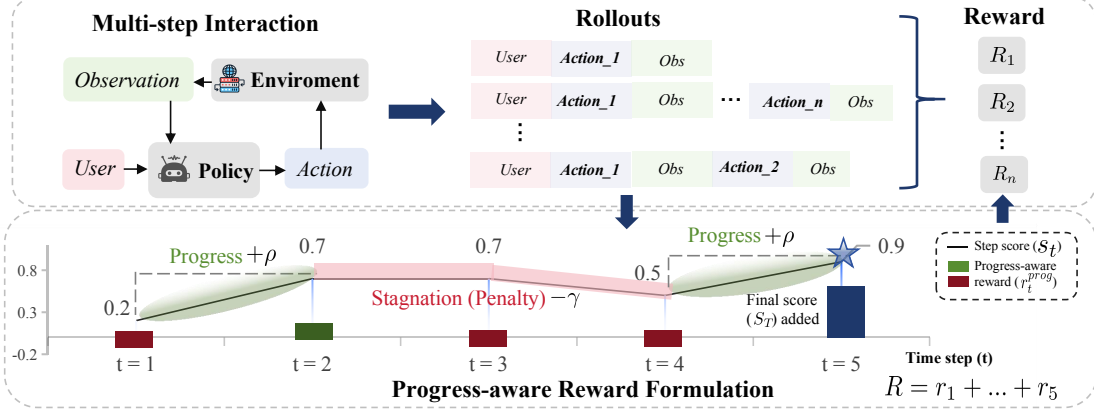


Figure 2: Overview of PRFC: we construct multi-attempt recovery rollouts and optimize the policy with a progress-aware reward that captures incremental improvement across consecutive repair attempts.

$s_t \in [0, 1]$ is normalized across tasks of varying complexity, whereas syntactically invalid actions receive $s_t = 0$. This score provides the local quality signal used in both the score-only ablation and the progress-based reward. Appendix C provides the computation details.

Cross-attempt progress shaping. However, local scoring alone treats each attempt independently and thus does not reveal whether the agent is advancing beyond previous failures. To capture this directional aspect, we introduce *cross-attempt progress shaping*. Instead of comparing an attempt with its immediate predecessor, we evaluate it against the best recovery achieved up to that point. Specifically, we maintain a running maximum $\bar{s}_t = \max\{s_1, \dots, s_t\}$ as the recovery frontier ($\bar{s}_0 = -1$). An attempt advances the frontier only when it exceeds the previous maximum; consequently, oscillations among partially correct states receive no credit. The primary objective is to determine whether an attempt breaks the current plateau. Therefore, we instantiate cross-attempt progress shaping with a binary frontier reward,

$$r_t^{\text{prog}} = \begin{cases} +\rho & \text{if } s_t > \bar{s}_{t-1} \text{ (progress)} \\ -\rho & \text{otherwise (stagnation),} \end{cases} \quad (2)$$

where $\rho > 0$. This design focuses exclusively on whether the frontier advances, thereby avoiding complex reward-scaling choices that would entangle credit assignment with magnitude tuning.

Additionally, we introduce a per-attempt step cost $r_{\text{step}} = -\lambda$ to discourage unnecessarily prolonged repair sequences. The complete immediate reward at attempt t is defined as

$$r_t = r_{\text{step}} + r_t^{\text{prog}} + \mathbf{1}[t = T] \cdot s_T, \quad (3)$$

where $\mathbf{1}[t = T] \cdot s_T$ anchors the trajectory return to the final repair quality. Overall, this reward function favors trajectories that advance the recovery frontier, minimize unnecessary interaction costs, and conclude with a high-quality repair. Consequently, it encourages the model to make effective corrections promptly, rather than engaging in unproductive interactions. We define $R = \sum_{t=1}^T r_t$ and set $\lambda = \rho$ to ensure that the efficiency and progress terms remain on the same scale.

To optimize training efficiency, we implement early stopping when $s_t \leq \bar{s}_{t-1}$ for k consecutive attempts, thereby pruning plateaued rollouts.

3.3 Policy Optimization

We optimize PRFC by maximizing $\mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)]$ using Group Relative Policy Optimization (GRPO) (Shao et al., 2024). For each query q , we sample a group of G trajectories, compute their returns $R_i = \sum_{t=1}^{T_i} r_t^i$, and normalize within the group to obtain advantages:

$$A_i = \frac{R_i - \text{mean}(\{R_j\}_{j=1}^G)}{\text{std}(\{R_j\}_{j=1}^G) + \epsilon}. \quad (4)$$

We then optimize the clipped objective with KL regularization toward the reference policy π_{ref} :

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{q \sim \mathcal{D}, \{\tau_i\}_{i=1}^G \sim \pi_{\theta_{\text{old}}}(\cdot|q)} \left[\frac{1}{G} \sum_{i=1}^G \min \left(\rho_i(\theta) A_i, \text{clip}(\rho_i(\theta), 1 - \epsilon, 1 + \epsilon) A_i \right) - \beta \text{D}_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}}) \right], \quad (5)$$

where $\rho_i(\theta) = \pi_\theta(\tau_i | q) / \pi_{\theta_{\text{old}}}(\tau_i | q)$ is the trajectory-level likelihood ratio. We also report

Data	Num.	Setting	Turn	Step	Feedback
Training data	1.4K	diagnostic tool-use rollouts	S	M	Diagnostic
BFCL-v3 MT	0.8K	agentic interaction	M	M	Native
BFCL-v3 ST	3.4K	live tool use	S	S	–
API-Bank	0.7K	API dialogue	S	S	–
T-Eval	23.3K	comprehensive tasks	S	S	–
Nestful	1.8K	nested API calls	S	S	–

Table 1: Training data and benchmark settings under the terminology of §3.1. “Setting” describes the data or evaluation interface, while “Turn” and “Step” use S/M to denote single/multiple user turns and repair attempts, respectively. “–” indicates a single-step benchmark without intermediate execution feedback. Additional details are provided in Appendix A.

results with GSPO (Zheng et al., 2025) as an alternative policy objective.

4 Experiments

Benchmarks. We evaluate PRFC using the official protocols of each benchmark rather than the diagnostic training environment, as summarized in Table 1. Specifically, the multi-turn split of BFCL-v3 (Patil et al., 2025) serves as our primary evaluation and recovery benchmark because it natively supports repeated, multi-attempt function calling under execution feedback. In contrast, API-Bank (Li et al., 2023), T-Eval (Chen et al., 2024), and Nestful (Basu et al., 2025) are used as single-turn, single-step benchmarks. Rather than testing error recovery, these supplementary tasks function as transfer and preservation assessments to examine whether recovery-oriented training compromises or preserves the quality of first-attempt tool use. Across all environments, we adopt the official metrics to evaluate final task success.

Training data and environment. We construct the training set from the ToolRL dataset (Qian et al., 2025) by filtering for recoverable-but-nontrivial instances, yielding $\sim 1.4\text{k}$ examples that encompass $\sim 7.6\text{k}$ unique tools. Because ToolRL does not provide a live, stateful backend, we train the policy using an offline diagnostic executor $\mathcal{E}$ (see more details in Appendix A.1). This executor compares each predicted call against the ground-truth annotation and generates a natural-language feedback string. The offline executor enables efficient generation of multi-attempt rollouts with lower latency and computational cost than a live backend, while preserving realistic feedback signals that reflect the quality of each attempt.

During evaluation, the learned policy is deployed under the native protocol of each benchmark, ensuring decoupling from the training data. Specifically,

BFCL-v3 interfaces with a live backend that returns execution outputs and exceptions, which provides a realistic environment for evaluating error recovery within complex interactions.

Baselines. To ensure a fair comparison and isolate performance gains from backbone variations, all controlled baselines are reimplemented within a standardized framework (Appendix E) using the identical **Qwen3-4B** backbone (Yang et al., 2025). *Single-attempt baselines* do not use inter-attempt recovery trajectories during training. Within this category, **SFT** serves as the standard supervised baseline. For reinforcement learning, **GRPO** and **GSPO** optimize trajectory-level returns without explicit recovery supervision, employing our unified fine-grained scoring scheme. **ToolRL** (Qian et al., 2025) adopts a similar optimization strategy but adheres to the original scoring formulations.

Conversely, *feedback-guided baselines* use multi-attempt recovery trajectories augmented with execution feedback. **ToolReflection** (Polyakov et al., 2025) applies SFT to filtered recovery traces, conditioning the model on the complete interaction history to predict the final repair action. **Early Experience** (Zhang et al., 2025b) augments imitation learning by sampling alternative actions from intermediate states to generate supplementary feedback-induced transitions. In contrast to these methods, **PRFC** optimizes a progress-shaped reinforcement learning objective, rewarding an attempt exclusively when it surpasses the best prior failure within the same trajectory.

4.1 Main Results

PRFC improves native multi-turn performance. Table 2 presents the primary BFCL-v3 results. Under the same evaluation protocol, PRFC improves the multi-turn split over the Qwen3-4B backbone and single-attempt baselines. This suggests that interactive tool-use interaction benefits from ordered repair sequences with explicit progress credit, rather than isolated action optimization alone. We analyze optimizer robustness separately in Table 7.

Progress-aware learning outperforms recovery imitation. Among methods using recovery trajectories, PRFC performs better than the feedback-guided baselines ToolReflection and Early Experience. While these baselines rely on imitation-style supervision over multi-attempt trajectories, PRFC optimizes a progress-aware objective that explicitly allocates credit only when a subsequent attempt ad-

Method	Param.	Multi Turn					Single Turn		Hal.	Overall
		Base	Miss Func	Miss Param	Long Context	Overall	Non-Live	Live		
External Leaderboard References										
GPT-5.2-2025-12-11	/	36.50	18.00	27.50	30.50	28.12	81.85	70.39	77.21	59.73
DeepSeek-V3.2-Exp (FC)	/	41.50	39.50	33.50	35.00	37.38	34.85	53.66	65.34	41.65
Qwen3-30B-A3B-Instruct	30B	43.50	10.50	25.00	41.00	30.00	85.77	77.94	80.56	64.18
Qwen3-8B (Prompt)	8B	41.50	38.50	27.00	26.50	33.38	88.56	80.09	78.63	66.93
ToolACE-MT (8B)	8B	57.50	31.50	34.00	38.00	40.25	84.94	71.52	74.31	65.41
Arch-Agent-3B	3B	42.00	37.50	31.00	29.00	34.88	86.67	72.91	71.71	64.43
Our Controlled Experiments										
Qwen3-4B (Backbone)	4B	39.50	32.00	19.00	22.00	28.10	87.40	79.79	78.21	64.67 (ref.)
+ SFT	4B	33.50	34.50	24.50	19.50	27.98	87.79	79.70	78.39	64.75 (↑0.1%)
+ GRPO	4B	35.00	31.50	23.00	19.00	27.10	88.52	80.19	80.96	64.74 (↑0.1%)
+ GSPO	4B	31.50	33.00	26.00	21.50	28.00	84.04	78.61	78.72	65.21 (↑0.8%)
+ ToolRL	4B	26.50	17.00	17.50	22.50	20.89	88.48	79.52	70.28	62.50 (↓3.4%)
+ ToolReflection	4B	36.50	37.00	26.50	22.00	30.50	86.73	83.33	79.19	65.33 (↑1.0%)
+ Early Experience	4B	35.00	34.00	24.00	23.50	29.13	87.73	80.01	72.91	65.24 (↑0.9%)
+ PRFC	4B	43.00	36.00	30.00	29.00	34.50	88.44	80.50	78.72	67.39 (↑4.2%)

Table 2: Performance on BFCL-v3 benchmark. External leaderboard references (top block; Dec. 16, 2025) are scale/data context, not a controlled comparison. All controlled methods (bottom block) share the Qwen3-4B backbone and protocol. PRFC uses GSPO as the default optimizer; optimizer robustness is analyzed in Table 7. The $\uparrow/\downarrow\%$ annotation is the relative Overall change vs. backbone. Hal. denotes hallucination.

Method	API-Bank	T-Eval	Nestful	Overall
Qwen3-4B	59.6	47.7	6.5	37.9
GSPO	62.3	46.6	6.2	38.4
ToolReflection	59.8	47.7	7.1	38.2
Early Experience	60.3	47.6	7.1	38.3
PRFC	61.7	49.4	7.3	39.5

Table 3: Performance on additional comparative benchmarks. Overall is the mean across the three benchmarks.

vances beyond prior failures. This comparison suggests that cross-attempt progress reward provides a more effective learning signal than the mere imitation of observed correction patterns, demonstrating that error recovery benefits from being optimized as a directional process.

Evaluation on supplementary single-step benchmarks. We further evaluate PRFC on API-Bank, T-Eval, and Nestful (Table 3) to examine whether recovery-oriented training affects foundational tool-use capability. Because these tasks evaluate single-step tool use, they serve as a complementary assessment of transfer and preservation rather than direct evidence of progressive recovery. PRFC achieves the highest mean score among the evaluated methods across these three benchmarks and performs well on T-Eval and Nestful, indicating that recovery-oriented optimization maintains general tool-use capability.

4.2 Ablation on Progress-Aware Credit Assignment

The main results evaluate the full PRFC training recipe, but they do not identify whether its progress-

aware shaping component contributes to the gain. This ablation is motivated by the reward design in §3.2: local repair scores indicate the quality of individual attempts, but they do not by themselves specify whether a new attempt advances beyond earlier failures. Given the multi-attempt rollout from §3.1, $\tau_u = (q_u, a_{u,1}, o_{u,1}, \dots, a_{u,T_u}, o_{u,T_u})$, we therefore compare PRFC against controls that use the same trajectory structure but convert scores into supervision differently.

Table 4 maps each control to this design choice. The terminal variants remove the progress-aware shaping component from Eq. 3, namely the frontier reward in Eq. 2 together with the per-attempt step cost, leaving only a terminal quality signal. These variants test whether terminal feedback alone can account for the gain. The *All attempts* variant gives the optimizer access to every intermediate score s_t , but applies each score independently as a direct local target; implementation details of this action-level objective are provided in Appendix B. In contrast, PRFC keeps s_T as the terminal anchor and uses intermediate scores only through progress-aware shaping, which rewards frontier advances while penalizing unproductive attempts.

Terminal-only rewards leave progress unmodeled. The first two variants *Terminal* ($\{0, 1\}$) and *Terminal* (s_T) restrict quality supervision to the terminal attempt $t = T$. Replacing the binary success-or-failure signal with s_T improves performance, suggesting that fine-grained terminal scoring alleviates reward sparsity. However, both variants still

Variant	Traj.	Score	Scope	Prog.	Acc.
Backbone	–	–	–	–	28.1
Terminal ($\{0, 1\}$)	✓	$\{0, 1\}$	$t = T$	–	29.9
Terminal (s_T)	✓	s_T	$t = T$	–	31.4
All attempts	✓	s_t	all t	–	33.5
PRFC	✓	s_T	$t = T$	✓	34.5

Table 4: Ablation on the BFCL-v3 multi-turn subset. Traj. indicates whether multi-attempt trajectories (§3.1) are used. Score and Scope specify which local score is used as an optimization target and where it is applied. Prog. denotes whether the progress-aware shaping component in Eq. 3 is used.

Method	Outcome distribution (%)				Recovery efficiency	
	Acc. ↑	Misguided repair ↓	Repeated failures ↓	Premature abandonment ↓	$\bar{A}$	SPA ↑
Qwen3-4B	28.1	49.6	2.6	6.6	2.58	10.9
GSPO	28.0	50.9	2.2	8.5	2.25	12.5
ToolReflection	31.8	48.0	1.9	7.8	2.47	12.9
Early Experience	29.1	48.9	3.1	5.2	2.60	11.2
PRFC	34.5	43.6	2.5	5.9	2.60	13.3

Table 5: Recovery-relevant outcome distribution and efficiency on the BFCL-v3 multi-turn subset. $\bar{A}$ is the average number of executed attempts until task success or permanent termination, and SPA = $\text{Acc}/\bar{A}$ measures success yield per attempt, jointly reflecting recovery quality and efficiency. Detailed definitions and computation rules for the failure categories are provided in Appendix F.

ignore how the model moves through the recovery trajectory, so they cannot distinguish productive repair from repeated or stagnant attempts.

Progress-aware ordering adds evidence beyond dense per-attempt scoring. We next ask whether the benefit can be explained simply by exposing the optimizer to more score signals. The *All attempts* variant improves over *Terminal* (s_T), suggesting that intermediate attempts contain useful supervisory information beyond the final endpoint. However, because it treats each s_t as an independent local target, it does not encode whether an attempt improves over earlier failures.

PRFC surpasses *All attempts* by a modest margin. This result does not by itself prove the mechanism, but it is consistent with the interpretation that organizing intermediate scores as progress-aware shaping provides benefit beyond treating the same scores as independent local labels.

4.3 Behavioral Analysis under Tool Feedback

The preceding ablation shows that progress-aware credit assignment improves recovery. We next examine how this advantage manifests after failed tool observations. By analyzing post-failure decisions, we can identify whether the model converts feedback into targeted repair.

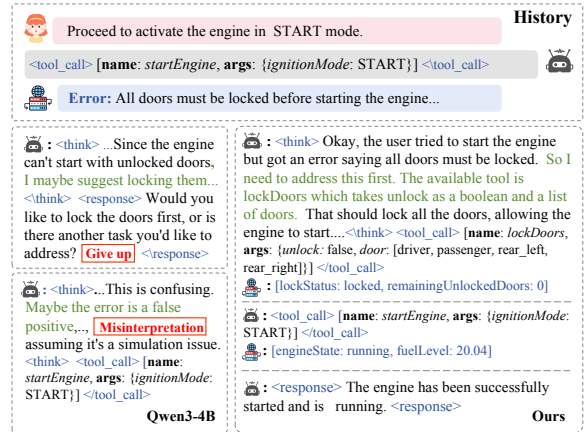


Figure 3: Illustrative repair decisions under tool feedback. The backbone exhibits premature abandonment or misguided repair, while PRFC identifies the unmet precondition and performs the intermediate repair needed to complete the request.

Baseline Failure behavioral outcomes. After a failed observation, the model must decide whether to repeat a previous action, stop, or revise its next tool call. We therefore analyze three post-failure outcomes using the definitions in Appendix F: *Repeated failures*, *Premature abandonment*, and *Misguided repair*. *Repeated failures* captures adjacent non-empty repetitions of the same canonical tool action. *Premature abandonment* captures stopping despite a recoverable failure. *Misguided repair* captures continued tool use that revises the wrong failure-relevant dimension, such as a violated precondition, incorrect state assumption, wrong argument slot, or execution-order dependency.

Misguided repair is the primary recovery bottleneck in Baselines. Table 5 identifies *misguided repair* as the primary recovery bottleneck. The dominant failure mode is not lack of persistence, but revision along the wrong failure-relevant dimension. Most unsuccessful trajectories occur because the model executes an alternative action yet modifies an incorrect aspect of the failed call, thereby failing to advance beyond prior errors. This pattern supports our central premise: effective error recovery requires learning which revisions make progress, not simply persisting after a failure.

PRFC alleviates misguided repair. As detailed in Table 5, PRFC achieves the highest overall accuracy alongside the lowest rate of misguided repair. This behavioral shift provides supporting evidence consistent with the intended effect of the progress-aware reward formulation. By assigning credit exclusively when an attempt advances the

Feedback	Base	Miss Func	Miss Param	Long Context	Overall
No-Progress	38.5	36.0	26.5	26.0	31.8
Detailed	40.0	37.0	31.5	28.0	34.1
Coarse	43.0	36.0	30.0	29.0	34.5

Table 6: Impact of the feedback interface on BFCL-v3 subsets under the same training recipe.

recovery frontier, the policy is trained to revise relevant failure dimensions rather than introducing arbitrary changes that merely prolong the interaction. Figure 3 illustrates this divergence: while the backbone model terminates prematurely or modifies incorrect action components, PRFC identifies unmet preconditions and executes necessary intermediate repairs prior to retrying the target call.

Efficiency supports the same interpretation.

Because success can occasionally emerge from numerous unproductive retries, accuracy alone insufficiently captures recovery quality. We therefore report SPA, which normalizes success by the average number of executed attempts. PRFC achieves the highest SPA (Table 5), suggesting that the performance gains are associated with more productive repair decisions. By contrast, GSPO combines the fewest attempts with the highest premature-abandonment rate, showing that shorter trajectories can reflect early termination rather than efficient recovery. This interpretation is also consistent with Figure 1, which shows that PRFC continues to derive benefits from subsequent attempts, whereas the performance of the evaluated baselines plateaus.

4.4 Robustness and Sensitivity Analysis

Robustness to optimizer choice. The main comparison reports PRFC with GSPO as the default optimizer. To clarify the role of the optimizer, Table 7 reports both the optimizer-matched RL baseline and PRFC under each optimizer, with changes relative to the same Qwen3-4B backbone shown in parentheses. The single-attempt GRPO and GSPO baselines do not substantially improve multi-turn recovery over the backbone, whereas PRFC improves under both optimizers. This pattern suggests that the gain is associated with the progress-aware objective rather than GSPO alone.

Robustness to training feedback granularity. Since PRFC trains against a diagnostic executor, the resulting gains might stem from memorizing fixed templates rather than acquiring generalizable recovery behavior. To examine this, the informa-

Opt.	Multi-turn		Overall	
	RL	PRFC	RL	PRFC
Backbone	28.10 (ref.)		64.67 (ref.)	
GRPO	27.10 (-1.00)	30.89 (+2.79)	64.74 (+0.07)	66.02 (+1.35)
GSPO	28.00 (-0.10)	34.50 (+6.40)	65.21 (+0.54)	67.39 (+2.72)

Table 7: Optimizer robustness on BFCL-v3. Parentheses report absolute point changes relative to the Qwen3-4B.

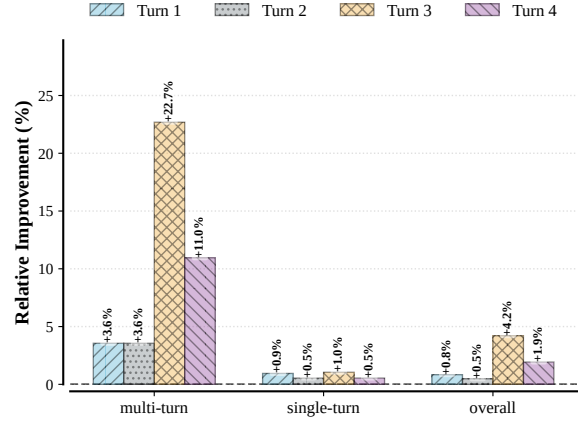


Figure 4: Rollout budget sensitivity. Relative improvement over the backbone on BFCL-v3 as a function of the maximum rollout attempts used during training, $T \in \{1, 2, 3, 4\}$.

tiveness of training-time feedback is ablated. The *Detailed* setting uses standard diagnostic templates, whereas *Coarse* substitutes a generic failure message (“please try again”). In addition, *No-Progress* serves as the RL counterpart to the supervised Tool-Reflection baseline; both use static multi-attempt traces terminating in a final action to isolate the contribution of process-level learning.

Table 6 shows that *Coarse* matches, and slightly exceeds, *Detailed*, while *No-Progress* performs worse. This suggests that PRFC is not simply memorizing the wording of detailed diagnostic templates. Moreover, the weaker performance of *No-Progress* indicates that multi-attempt trajectories alone are insufficient to match PRFC. Since evaluation on BFCL-v3 uses native execution outputs that differ from the offline diagnostic feedback used during training, these results support the view that PRFC learns recovery behavior that transfers beyond the diagnostic training interface.

Training Rollout horizon. We vary the maximum rollout depth T , which controls the number of repair attempts allowed within each training trajectory. As shown in Figure 4, performance improves as T increases from 1 to 3, indicating that additional repair opportunities help the model learn

ρ	Base	Miss Func	Miss Param	Long Context	Overall
0	40.0	37.0	25.5	23.0	31.4
0.1	43.0	36.0	30.0	29.0	34.5
0.2	36.0	36.5	27.5	26.5	31.6

Table 8: Impact of the progress shaping reward scale ρ on BFCL-v3 subsets under the same training recipe, with $\lambda = \rho$.

from multi-attempt recovery trajectories. However, performance declines at $T=4$, suggesting that longer rollouts introduce limited additional benefit and may increase less informative exploration. Based on this trade-off, we set $T=3$ by default.

Cross-attempt progress shaping scale. We vary the progress-reward scale ρ while keeping the training recipe fixed, and set $\lambda = \rho$ to maintain a balanced trade-off between progress rewards and step costs. As shown in Table 8, $\rho=0.1$ achieves the best performance. Setting $\rho=0$ removes the progress signal, while increasing it to $\rho=0.2$ places more weight on intermediate progress and leads to lower performance. These results suggest that a moderate reward scale provides a more effective training signal. We therefore use $\rho=\lambda=0.1$ by default.

5 Conclusion

PRFC improves the reliability of tool-using LLMs by learning feedback-guided recovery from their own errors. By modeling recovery as a sequence of repair attempts and rewarding progress across attempts, PRFC trains the model to use execution feedback to improve subsequent tool calls. Experiments show that PRFC substantially improves multi-turn recovery on BFCL-v3 and remains competitive on additional single-step tool-use benchmarks. These results show that modeling progress across attempts provides an effective learning signal for multi-attempt tool-use recovery.

Limitations

This study focuses on tool-calling environments with explicit execution feedback, where training episodes consist of multiple repair attempts within a single turn. This setting provides a controlled way to study how models learn from repeated failures and subsequent corrections. Future work may extend PRFC to longer-horizon agent interactions involving evolving observations, intermediate subgoals, or delayed feedback.

Our experiments use a 4B backbone under current computational and data constraints. Future

work will evaluate PRFC on larger models and broader agentic tasks to further assess its scalability.

References

- Rishabh Agarwal, Chen Liang, Dale Schuurmans, and Mohammad Norouzi. 2019. Learning to generalize from sparse and underspecified rewards. In *International conference on machine learning*, pages 130–140. PMLR.
- Kinjal Basu, Ibrahim Abdelaziz, Kiran Kate, Mayank Agarwal, Maxwell Crouse, Yara Rizk, Kelsey Bradford, Asim Munawar, Sadhana Kumaravel, Saurabh Goyal, and 1 others. 2025. Nestful: A benchmark for evaluating llms on nested sequences of api calls. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 33526–33535.
- Zehui Chen, Weihua Du, Wenwei Zhang, Kuikun Liu, Jiangning Liu, Miao Zheng, Jingming Zhuo, Songyang Zhang, Dahua Lin, Kai Chen, and 1 others. 2024. T-eval: Evaluating the tool utilization capability of large language models step by step. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9510–9529.
- Yue Cui, Liuyi Yao, Shuchang Tao, Weijie Shi, Yaliang Li, Bolin Ding, and Xiaofang Zhou. 2025. Enhancing tool learning in large language models with hierarchical error checklists. *arXiv preprint arXiv:2506.00042*.
- Guanting Dong, Hangyu Mao, Kai Ma, Licheng Bao, Yifei Chen, Zhongyuan Wang, Zhongxia Chen, Jiazhen Du, Huiyang Wang, Fuzheng Zhang, and 1 others. 2026. Agentic reinforced policy optimization. In *International Conference on Learning Representations*, volume 2026, pages 16981–17017.
- Lutfi Eren Erdogan, Nicholas Lee, Siddharth Jha, Sehoon Kim, Ryan Tabrizi, Suhong Moon, Coleman Richard Charles Hooper, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. 2024. Tinyagent: Function calling at the edge. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 80–88.
- Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. 2026. Group-in-group policy optimization for llm agent training. *Advances in Neural Information Processing Systems*, 38:46375–46408.
- Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Nan Duan, Weizhu Chen, and 1 others. 2024. Critic: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations*, volume 2024, pages 57734–57811.

- Shiting Huang, Zhen Fang, Zehui Chen, Siyu Yuan, Junjie Ye, Yu Zeng, Lin Chen, Qi Mao, and Feng Zhao. 2025. Critictool: Evaluating self-critique capabilities of large language models in tool-calling error scenarios. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 26683–26692.
- Ryo Kamoi, Yusen Zhang, Nan Zhang, Jiawei Han, and Rui Zhang. 2024. When can llms actually correct their own mistakes? a critical survey of self-correction of llms. *Transactions of the Association for Computational Linguistics*, 12:1417–1440.
- Chengpeng Li, Zhengyang Tang, Ziniu Li, Mingfeng Xue, Keqin Bao, Tian Ding, Ruoyu Sun, Benyou Wang, Xiang Wang, Junyang Lin, and 1 others. 2025a. Cort: Code-integrated reasoning within thinking. *arXiv preprint arXiv:2506.09820*.
- Chengpeng Li, Zhengyang Tang, Ziniu Li, Mingfeng Xue, Keqin Bao, Tian Ding, Ruoyu Sun, Benyou Wang, Xiang Wang, Junyang Lin, and 1 others. 2026. Teaching language models to reason with tools. *Advances in Neural Information Processing Systems*, 38:66843–66892.
- 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. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pages 3102–3116.
- Xiaoxi Li, Guanting Dong, Jiajie Jin, Yuyao Zhang, Yujia Zhou, Yutao Zhu, Peitian Zhang, and Zhicheng Dou. 2025b. Search-o1: Agentic search-enhanced large reasoning models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 5420–5438.
- Weiwen Liu, Xu Huang, Xingshan Zeng, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong WANG, and 1 others. 2025. Toolace: Winning the points of llm function calling. In *International conference on learning representations*, volume 2025, pages 41359–41381.
- Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, Rithesh RN, and 1 others. 2024. Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets. *Advances in Neural Information Processing Systems*, 37:54463–54482.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, 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.
- 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 (bfc1): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*.
- Gregory Polyakov, Ilseyar Alimova, Dmitry Abul'khanov, Ivan Sedykh, Andrey Bout, Sergey Nikolenko, and Irina Piontkovskaya. 2025. Tool-reflection: Improving large language models for real-world api calls with self-generated data. In *Proceedings of the 1st Workshop for Research on Agent Language Models (REALM 2025)*, pages 184–199.
- Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tur, Gokhan Tur, and Heng Ji. 2025. Toolrl: Reward is all tool learning needs. *arXiv preprint arXiv:2504.13958*.
- Shuofei Qiao, Honghao Gui, Chengfei Lv, Qianghuai Jia, Huajun Chen, and Ningyu Zhang. 2024. Making language models better tool learners with execution feedback. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3550–3568.
- Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, and 1 others. 2024. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *International Conference on Learning Representations*, volume 2024, pages 9695–9717.
- Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2025. From exploration to mastery: Enabling llms to master tools via self-driven interactions. In *International Conference on Learning Representations*, volume 2025, pages 55675–55697.
- 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*.
- Weizhou Shen, Chenliang Li, Hongzhan Chen, Ming Yan, Xiaojun Quan, Hehong Chen, Ji Zhang, and Fei Huang. 2024. Small llms are weak tool learners: A multi-llm agent. In *Proceedings of the 2024 conference on empirical methods in natural language processing*, pages 16658–16680.
- Zhengliang Shi, Shen Gao, Lingyong Yan, Yue Feng, Xiuyi Chen, Zhumin Chen, Dawei Yin, Suzan Verberne, and Zhaochun Ren. 2025. Tool learning in the wild: Empowering language models as automatic tool agents. In *Proceedings of the ACM on Web Conference 2025*, pages 2222–2237.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652.

- Joykirat Singh, Raghav Magazine, Yash Pandya, and Akshay Nambi. 2025. Agentic reasoning and tool integration for llms via reinforcement learning. *arXiv preprint arXiv:2505.01441*.
- Jimin Sun, So Yeon Min, Yingshan Chang, and Yonatan Bisk. 2024. Tools fail: Detecting silent errors in faulty tools. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 14272–14289.
- Weiting Tan, Xinghua Qu, Ming Tu, Meng Ge, Andy T Liu, Philipp Koehn, and Lu Lu. 2025. Process-supervised reinforcement learning for interactive multimodal tool-use agents. *arXiv preprint arXiv:2509.14480*.
- Guoqing Wang, Sunhao Dai, Guangze Ye, Zeyu Gan, Wei Yao, Yong Deng, Xiaofeng Wu, and 1 others. 2026. Information gain-based policy optimization: A simple and effective approach for multi-turn search agents. In *International Conference on Learning Representations*, volume 2026, pages 104502–104529.
- Zhepei Wei, Wenlin Yao, Yao Liu, Weizhi Zhang, Qin Lu, Liang Qiu, Changlong Yu, Puyang Xu, Chao Zhang, Bing Yin, and 1 others. 2025. Webagent-r1: Training web agents via end-to-end multi-turn reinforcement learning. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 7920–7939.
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.
- Fan Yin, Zifeng Wang, I-Hung Hsu, Jun Yan, Ke Jiang, Yanfei Chen, Jindong Gu, Long Le, Kai-Wei Chang, Chen-Yu Lee, and 1 others. 2025. Magnet: Multi-turn tool-use data synthesis and distillation via graph translation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 32600–32616.
- Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Kan Ren, Dongsheng Li, and Deqing Yang. 2025. Easytool: Enhancing llm-based agents with concise tool instruction. 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 951–972.
- Xingshan Zeng, Weiwen Liu, Lingzhi Wang, Liangyou Li, Fei Mi, Yasheng Wang, Lifeng Shang, Xin Jiang, and Qun Liu. 2026. Toolace-mt: Non-autoregressive generation for agentic multi-turn interaction. In *International Conference on Learning Representations*, volume 2026, pages 3122–3145.
- Guibin Zhang, Hejia Geng, Xiaohang Yu, Zhenfei Yin, Zaibin Zhang, Zelin Tan, Heng Zhou, Zhongzhi Li, Xiangyuan Xue, Yijiang Li, and 1 others. 2025a. The landscape of agentic reinforcement learning for llms: A survey. *arXiv preprint arXiv:2509.02547*.
- Kai Zhang, Xiangchao Chen, Bo Liu, Tianci Xue, Zeyi Liao, Zhihan Liu, Xiyao Wang, Yuting Ning, Zhaorun Chen, Xiaohan Fu, and 1 others. 2025b. Agent learning via early experience. *arXiv preprint arXiv:2510.08558*.
- Shaokun Zhang, Yi Dong, Jieyu Zhang, Jan Kautz, Bryan Catanzaro, Andrew Tao, Qingyun Wu, Zhiding Yu, and Guilin Liu. 2025c. Nemotron-research-tool-n1: Tool-using language models with reinforced reasoning. *arXiv preprint arXiv:2505.00024*, 2(10).
- Weikang Zhao, Xili Wang, Chengdi Ma, Lingbin Kong, Zhaohua Yang, Mingxiang Tuo, Xiaowei Shi, Yitao Zhai, and Xunliang Cai. 2025. Mua-rl: Multi-turn user-interacting agent reinforcement learning for agentic tool use. *arXiv preprint arXiv:2508.18669*.
- Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. 2025. Group sequence policy optimization. *ArXiv*, abs/2507.18071.

A Training Environment and Benchmark Details

Table 1 provides the compact summary used in the main text. This appendix gives additional details on the training environment, benchmark protocols, and the boundary between them.

Training environment. The training data are drawn from ToolRL (Qian et al., 2025), which consists of single-turn function-calling instances with annotated ground-truth calls. PRFC converts these instances into single-turn, multi-step recovery trajectories using the offline diagnostic executor $\mathcal{E}$. At each repair attempt, $\mathcal{E}$ compares the predicted tool call with the ground-truth call and returns one of the templated diagnostic observations described in Appendix A.1. The observation is appended to the interaction history as textual tool feedback, and the policy decides whether to continue and what action to attempt next. This diagnostic interaction is used only for training-time recovery supervision.

Benchmark protocols. Benchmark evaluation uses each benchmark’s native protocol rather than $\mathcal{E}$. BFCL-v3 (Patil et al., 2025) provides both multi-turn and single-turn subsets. In the multi-turn subset, each episode contains multiple user turns, and each turn permits up to BFCL’s default `MAXIMUM_STEP_LIMIT=20` function-call steps. Feedback comes from native benchmark execution; in particular, BFCL multi-turn runs against live stateful Python classes such as `GorillaFileSystem` and `TwitterAPI`, exposing native backend outputs or exceptions without ground-truth access. In our setup, the BFCL-v3 single-turn split, API-Bank (Li et al., 2023), T-Eval (Chen et al., 2024), and Nestful (Basu et al., 2025) are treated as single-turn, single-step benchmarks and are scored with their official metrics.

Train–benchmark boundary. The training and evaluation environments share no common tool registries, error strings, or state semantics. Training uses approximately 7.6k ToolRL tools, while BFCL-v3 evaluation uses a disjoint set of stateful APIs (e.g., `GorillaFileSystem` and `TwitterAPI`). During training, the policy observes only the natural-language feedback provided in the interaction context, not the underlying ground-truth annotations. No external ground-truth signals are available at evaluation time. Thus, the only information transferred from training to evaluation is encoded in the learned policy parameters.

A.1 Diagnostic Executor for Train-Time Feedback

We implement a diagnostic executor $\mathcal{E}$ to provide structured offline feedback during training. The executor is built from ToolRL ground-truth annotations (Qian et al., 2025) and is used only during training, not for BFCL-v3 evaluation. It functions as a train-time supervision mechanism rather than as a model of deployment-time tool feedback.

Instead of executing real API calls, it evaluates model-predicted tool calls (Pred) by comparing them with benchmark-provided ground truth (GT). When a mismatch is detected, the executor returns a diagnosis together with a natural-language feedback message, wrapped in `<tool_response>`, that specifies the failure type; this feedback is then appended to the interaction history for subsequent repair attempts. This design yields reproducible multi-step supervision while remaining clearly distinct from the native benchmark backends used at evaluation time. Following the error taxonomy of Huang et al. (2025), we categorize tool-use failures into four common error types, each associated with a fixed feedback template.

Tool Selection Error. The model invokes a valid but incorrect tool/function for the user request.

Tool Selection Error

Query: Help me calculate the square root of 144.
GT: `Calculator.execute(expression="sqrt(144)")`
Pred: `SearchEngine.search(query="square root of 144")`
Feedback: There are issues with your tool calls: you attempted to call tool(s), but selected the wrong function(s). Please review your response and ensure all required tools are correctly invoked to fulfill the query.

Unnecessary Tool Call (or Hallucination). The user query is underspecified (i.e., required arguments are not provided), so the correct behavior is to request clarification rather than issuing a tool call or guessing missing fields.

Unnecessary Tool Call

Query: Check the current stock price.
GT: I can help, but the request is missing required information. Which company (or ticker symbol) would you like me to look up?
Pred: `FinanceTools.get_quote(symbol="AAPL")`
Feedback: There are issues with your tool calls: one tool call is unnecessary. Please review your response and ensure all required tools are correctly invoked to fulfill the query.

Parameter Key Error. The model selects the correct tool/function but uses incorrect argument keys

(e.g., missing required keys or using undefined keys).

Parameter Key Error

Query: Search for papers about LLM agents.
GT: Search(keywords="LLM agents", limit=5)
Pred: Search(query="LLM agents", max_results=5)
Feedback: There are issues with your tool calls: one tool call is missing required parameters or has incorrect parameter values. Please review your response and ensure all required tools are correctly invoked to fulfill the query.

Parameter Value Error. The model uses correct argument keys but provides ill-formatted or semantically incorrect argument values.

Parameter Value Error

Query: Set the timer for 0.5 seconds.
GT: Timer.set(duration=0.5) (*Float*)
Pred: Timer.set(duration="0.5s") (*String*)
Feedback: There are issues with your tool calls: one tool call is missing required parameters or has incorrect parameter values. Please review your response and ensure all required tools are correctly invoked to fulfill the query.

B StepGRPO

B.1 Instance-Wise Action Grouping

StepGRPO is the implementation of the per-attempt credit control used in Table 4; it is not an external baseline. We introduce this name only for reproducibility of the step-level GRPO objective. This control variant assigns credit to individual actions rather than whole recovery trajectories. For a query q , we sample a group of G trajectories $\{\tau_i\}_{i=1}^G \sim \pi_{\theta_{\text{old}}}(\cdot | q)$, where trajectory τ_i contains T_i tool-use actions (recovery attempts). Let $y_{i,t}$ denote the token span of the t -th action, and let $s_{i,t} \in [0, 1]$ denote its fine-grained repair-attempt score computed using the scoring function in §3.2. StepGRPO pools all step scores from the sampled trajectories for the same query, normalizes them into per-action advantages, and applies a GRPO update to each action span independently. It therefore uses every step score as an attempt-level learning signal while deliberately removing the trajectory-level progress coupling that distinguishes PRFC. In this way, StepGRPO serves as a targeted control for isolating the benefit of progress-aware learning.

Instance-wise action set. We flatten all actions from the sampled trajectories for the same query:

$$\mathcal{A}(q) = \{(i, t) \mid 1 \leq i \leq G, 1 \leq t \leq T_i\}, \quad (6)$$

and define the total number of actions under q as

$$M_q = |\mathcal{A}(q)| = \sum_{i=1}^G T_i. \quad (7)$$

Instance-wise advantage. Instead of using trajectory returns R_i , we compute a group-normalized advantage directly from the per-action scores within $\mathcal{A}(q)$:

$$A_{i,t}^{\text{ACT}} = \frac{s_{i,t} - \text{mean}(\{s_{j,u}\}_{(j,u) \in \mathcal{A}(q)})}{\text{std}(\{s_{j,u}\}_{(j,u) \in \mathcal{A}(q)}) + \epsilon}. \quad (8)$$

This objective encourages the policy to increase the likelihood of actions whose local quality exceeds that of other actions sampled for the same query, thereby providing dense local learning signals even when trajectories terminate early or fail overall. At the same time, because $A_{i,t}^{\text{ACT}}$ depends only on the local repair-attempt score $s_{i,t}$ and query-level normalization, it removes the explicit across-attempt progress signal used in PRFC.

B.2 StepGRPO Objective

We define the action-level likelihood ratio over the action token span as:

$$\rho_{i,t}(\theta) = \frac{\pi_{\theta}(y_{i,t} \mid q, y_{i,<t})}{\pi_{\theta_{\text{old}}}(y_{i,t} \mid q, y_{i,<t})}, \quad (9)$$

where the numerator and denominator are computed from token likelihoods restricted to the span $y_{i,t}$ under the same prefix context $y_{i,<t}$.

The action-level objective follows the trajectory-level GRPO update in Sec. 3.3 but applies clipping and advantages at the level of individual action spans:

$$\begin{aligned} \mathcal{J}_{\text{StepGRPO}}(\theta) = & \mathbb{E}_{q \sim \mathcal{D}, \{\tau_i\}_{i=1}^G \sim \pi_{\theta_{\text{old}}}(\cdot | q)} \left[\frac{1}{M_q} \right. \\ & \sum_{(i,t) \in \mathcal{A}(q)} \min \left(\rho_{i,t}(\theta) A_{i,t}^{\text{ACT}}, \right. \\ & \left. \text{clip}(\rho_{i,t}(\theta), 1 - \epsilon, 1 + \epsilon) A_{i,t}^{\text{ACT}} \right) \\ & \left. - \beta \text{D}_{\text{KL}}(\pi_{\theta} \parallel \pi_{\text{ref}}) \right]. \quad (10) \end{aligned}$$

Action-level credit assignment. In practice, $A_{i,t}^{\text{ACT}}$ is broadcast to all tokens within the span $y_{i,t}$ so that the policy gradient assigns credit to the entire action realization. This treatment is consistent with viewing each repair attempt as an atomic decision: all tokens in $y_{i,t}$ jointly determine the action’s syntactic validity (e.g., a well-formed tool call) and semantic correctness (e.g., the correct function and

arguments). Broadcasting therefore provides a stable action-level learning signal while preserving the token-level likelihood factorization required by GRPO optimization.

C Fine-Grained Repair-Attempt Scoring

The repair-attempt score is used as a local proxy for repair quality. Here we give the exact scoring function and aggregation details. When evaluating tool call generation, an action can produce multiple predicted calls $\mathcal{C} = \{c_1, \dots, c_n\}$, while the ground truth requires $\mathcal{C}^* = \{c_1^*, \dots, c_m^*\}$.

For a predicted tool call $c = (n, P)$ with tool name n and parameter map P , and a ground-truth call $c^* = (n^*, P^*)$, we define the pairwise matching score

$$S(c, c^*) = \underbrace{\mathbf{1}[n = n^*]}_{\text{name}} + \underbrace{\frac{|K \cap K^*|}{|K \cup K^*|}}_{\text{key}} + \underbrace{\sum_{k \in K \cap K^*} \mathbf{1}[v_k = v_k^*]}_{\text{value}}, \quad (11)$$

where K and K^* denote the predicted and ground-truth parameter-key sets, respectively, and v_k is the corresponding value for key k . This decomposition gives credit for correct tool selection, partial key overlap, and correctly matched parameter values, so partially correct calls still contribute a directional learning signal.

To fairly assess the quality of a set of calls, we establish a greedy matching $\mathcal{M}$ between predicted and ground-truth calls by maximizing total similarity. The algorithm iteratively selects the highest-scoring pair (c_i, c_j^*) according to $S(c_i, c_j^*)$, adds this pair to $\mathcal{M}$, removes both calls from consideration, and repeats until no calls remain in either set. This produces the aggregate actual score:

$$s_{\text{actual}} = \sum_{(c_i, c_j^*) \in \mathcal{M}} S(c_i, c_j^*) \quad (12)$$

To enable meaningful comparisons across instances with varying numbers of calls and parameters, we normalize s_{actual} by computing maximum achievable scores for both predicted and ground-truth sets. For a tool call $c = (n, P)$ with parameter set P of size $|P|$, the maximum possible score is $s_{\text{max}}(c) = 1 + 1 + |P| = 2 + |P|$. This decomposition reflects the scoring function’s three components: 1 point for correct tool name matching ($s_n = 1$), 1 point from perfect key overlap ($s_k = 1$ when $K = K^*$), and up to $|P|$ points for correct parameter values. The maximum achievable scores

for the predicted and ground-truth sets are then:

$$s_{\text{max}}^{\text{pred}} = \sum_{j=1}^n (2 + |P_j|), \quad s_{\text{max}}^{\text{gt}} = \sum_{i=1}^m (2 + |P_i^*|) \quad (13)$$

We define precision as the ratio of actual score to maximum predicted score, measuring the quality of generated calls:

$$\mathcal{P} = \frac{s_{\text{actual}}}{s_{\text{max}}^{\text{pred}}} \quad (14)$$

Recall is defined as the ratio of actual score to maximum ground-truth score, capturing how well the predictions cover the required functionality:

$$\mathcal{R} = \frac{s_{\text{actual}}}{s_{\text{max}}^{\text{gt}}} \quad (15)$$

The final repair-attempt score combines both metrics via their harmonic mean (F1 score):

$$s_t = \frac{2\mathcal{P}\mathcal{R}}{\mathcal{P} + \mathcal{R}} \quad (16)$$

This balanced metric penalizes both over-generation (low precision) and under-generation (low recall), providing a comprehensive assessment of tool call quality. Actions with syntax errors receive $s_t = 0$, ensuring that malformed outputs do not obtain misleading partial credit.

D Additional Details on Feedback Robustness

The main text reports the aggregate robustness results under detailed and coarse feedback. Here we provide the training-dynamics view.

Training dynamics. Figure 5 compares optimization under the two interactive feedback interfaces. Detailed diagnostics reach high Score@1 earlier, consistent with higher sample efficiency when error attribution is reliable and well covered by the training distribution. In contrast, coarse feedback improves more gradually yet attains slightly better final performance in our setting, which may reflect greater robustness when diagnostic templates do not fully cover the space of tool-induced failures under the current data and tool registry. Unlike §4.2, which holds the scoring signal fixed to isolate credit assignment, this ablation varies the informativeness of feedback to test robustness under heterogeneous and underspecified tool responses.

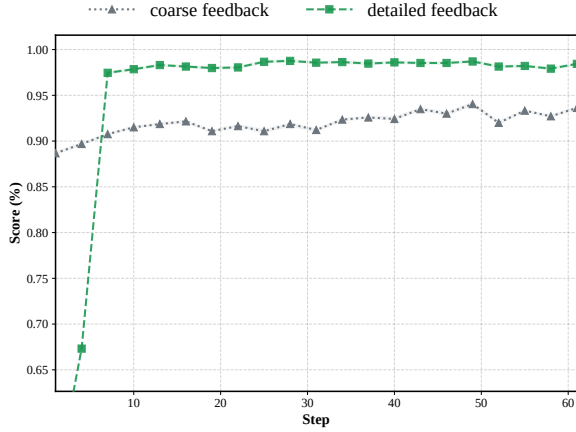


Figure 5: Training curves of PRFC under different feedback interfaces, measured by Score@1.

E Training Details

Data filtering. The training data are drawn from ToolRL (Qian et al., 2025), which provides single-turn function-calling instances with annotated ground-truth tool calls. Because first-attempt accuracy under this setup is around 90%, we focus on recoverable-but-nontrivial instances. We estimate difficulty by running $G=20$ rollouts with the reference policy π_{ref} and retain instances with empirical failure rate $0.01 < f < 0.6$. This filtering yields $\sim 1.4\text{k}$ examples spanning $\sim 7.6\text{k}$ unique tools.

Reinforcement learning. All reinforcement learning experiments are implemented with veRL and SGLang. At each optimization step, we sample a batch of 256 prompts and generate 4 rollouts per prompt from the current policy. Unless otherwise specified, all RL runs are trained for 15 epochs on 2 NVIDIA A800 GPUs. To encourage exploration during rollout, we set the decoding temperature to 1.0. Table 9 summarizes the hyperparameter configuration used for both GRPO and GSPO.

Supervised fine-tuning (SFT). All methods are initialized from the same Qwen3-4B backbone. We perform SFT on the corresponding training split using a standard instruction-tuning objective, where the model is trained to predict the target tool-use trajectory conditioned on the full prompt. We truncate prompts to at most 2048 tokens and cap generated responses at 4096 tokens. SFT is implemented with LoRA to reduce memory footprint; unless otherwise specified, we apply LoRA to the attention and MLP projection layers of the backbone. Training uses a global batch size of 256 and a learning rate of 1×10^{-4} , with mini-batches of size 32. Table 10

Category	Value
Data Configuration	
Train batch size	256
Validation batch size	128
Max prompt length	2048
Max response length	4096
Optimization	
Learning rate	1×10^{-6}
PPO mini-batch size	32
KL loss	Enabled
KL coefficient	0.001
Rollout Configuration	
Rollout engine	SGLang
GPU memory utilization	0.45
Number of rollouts	4
Training & Logging	
Checkpoint frequency (steps)	10
Evaluation frequency (steps)	20
Total epochs	15
Policy loss	{GRPO, GSPO}

Table 9: Hyperparameter configuration for RL training.

Category	Value
Data Configuration	
Train batch size	256
Max prompt length	2048
Max response length	4096
Optimization	
Learning rate	1×10^{-4}
Mini-batch size	32
LoRA Configuration	
LoRA rank (r)	32
LoRA alpha (α)	16

Table 10: Hyperparameter configuration for SFT training.

reports the SFT hyperparameters.

Implementation details of Early Experience.

To ensure a fair comparison, we re-implemented Early Experience on our training split and reconstructed its training data accordingly. Following Implicit World Modeling, the training corpus is derived from the base trajectories and consists of two components: (i) Refined expert data. We reformat the original training trajectories into the world-modeling format, where the model predicts the next state conditioned on the interaction history and the previous-step action. (ii) Exploration data augmentation. For each state, we run the original policy for 50 rollouts to collect diverse candidate responses. We then deduplicate these outputs to retain unique candidates, sample alternative actions from this

set, and execute them in the diagnostic executor (see Appendix A.1) to obtain the corresponding next states. This procedure yields approximately 2.7K additional (s, a', s') triplets, augmenting the dataset with diverse execution feedback.

We adopt the same two-stage training pipeline as the original work: a warm-up stage optimized with the implicit world modeling objective, followed by the main training phase. We keep the number of warm-up steps and the imitation-learning epochs identical to the hyperparameters reported in the original implementation.

Implementation Details of ToolReflection.

We implement ToolReflection as a trajectory-supervised recovery-imitation baseline. The training data for our internal dataset split are generated using a multi-attempt rollout protocol. Specifically, we execute the current policy within the diagnostic executor to collect multi-attempt interaction trajectories.

To follow the core idea of ToolReflection, which learns from model-generated failures and subsequent corrections, we retain only trajectories that contain a successful recovery process. Specifically, a trajectory is kept when the model encounters an API error or suboptimal execution feedback, revises its tool-use decision in a later step, and eventually completes the task successfully.

For each retained trajectory, the model is conditioned on the full interaction history, including all preceding actions, and trained with supervised fine-tuning to predict the final corrective action. ToolReflection therefore provides a non-RL baseline that learns from successful recovery trajectories without using the progress-aware reward.

F Decision-Level Failure Metrics

The main text reports three recovery-oriented failure metrics on BFCL-v3 multi-turn. These metrics are intended to characterize the decision made after tool feedback, rather than to replace the official BFCL success metric. We compute them only for analysis after evaluation. Only episodes that fail under the official BFCL metric enter the four-way partition below; successful episodes contribute zero to all four failure rates.

Evaluation unit and deduplication. The unit of analysis is an episode. Each episode may contain multiple user turns, and each turn may contain multiple repair attempts after the model observes native

benchmark feedback. An episode is counted for category C if any turn satisfies that category’s rule. The reported rate for category C is

$$\text{Rate}(C) = \frac{1}{|\mathcal{E}|} \sum_{e \in \mathcal{E}} \mathbf{1}[C(e)], \quad (17)$$

where $\mathcal{E}$ is the set of evaluated BFCL-v3 multi-turn episodes. In the main table, $|\mathcal{E}| = 800$ for each method, corresponding to four multi-turn subsets with 200 episodes each. BFCL score files may contain multiple diagnostic records for the same episode id, so we deduplicate by episode id before computing any category rate.

Canonicalizing tool actions. To determine whether two repair attempts repeat the same action, we keep only executable tool calls. For each attempt, we parse the function name and arguments, sort argument keys, and serialize the resulting ordered tool-call sequence. Natural-language reasoning and formatting differences outside executable tool calls are ignored. If an attempt contains no executable tool call, it is treated as empty and therefore cannot trigger repeated-failures assignment.

Assigning one failure category per failed episode.

We assign exactly one category to each failed episode using the following procedure. **Step 1.** If any user turn contains two adjacent repair attempts with the same non-empty canonical tool action, we assign *repeated failures*. **Step 2.** Otherwise, if any turn stops after weak or non-diagnostic but recoverable feedback and returns a natural-language failure or clarification response instead of tool action, we assign *premature abandonment*. **Step 3.** Otherwise, if the official BFCL evaluator reports an instance-state mismatch, an execution-response mismatch, or forced termination, we assign *misguided repair*. **Step 4.** Any remaining failed episode is assigned to *residual other*. This ordering makes the categories mutually exclusive and separates exact adjacent repetition from broader revision failures before later checks are applied.

Repeated failures. We count a failed episode as repeated failures if some user turn contains two adjacent repair attempts with the same non-empty canonical tool action. We require adjacency because returning to an earlier action after gathering new evidence can be reasonable, whereas exact back-to-back repetition is a stronger sign that the immediately preceding feedback was not absorbed.

Premature abandonment. We count a failed episode as premature abandonment if some turn already contains at least one tool attempt, receives weak or non-diagnostic but recoverable feedback, and then ends with a natural-language failure or clarification response instead of another tool action. Typical weak feedback includes empty match lists, no-result messages, and file-not-found errors; legitimate clarification for genuinely missing unrecoverable information, especially in missing-parameter settings, is excluded.

Misguided repair. We count a failed episode as misguided repair only after repeated failures and premature abandonment have been excluded. The episode is assigned here if the official BFCL evaluator reports an instance-state mismatch, an execution-response mismatch, or forced termination; we use these official outcomes as observable proxies because BFCL does not expose the training-time per-attempt score s_t . State and execution-response mismatches indicate that the model continued interacting but revised the wrong failure-relevant dimension, producing the wrong environment state or the wrong sequence of tool observations. Forced termination is placed here rather than in repeated failures because, once exact adjacent repetition has been removed, exhausting the interaction budget is better interpreted as non-convergent revision than as a simple duplicate-action loop.

Residual other. All remaining failed episodes are assigned to residual other after the first three rules have been applied. This bucket typically includes empty responses, malformed outputs, inference failures, and other evaluator errors that do not cleanly instantiate a repeat, stop, or revise decision after feedback.

G System Prompt and Case Study

System prompt. All experiments are conducted with the Qwen3 series and its official Jinja-based chat template. We retain Qwen3’s native #tool prompt, which standardizes tool-use formatting by (i) providing available function signatures inside <tools> tags and (ii) requiring the agent to emit each invocation as a JSON object wrapped by <tool_call> tags. If no provided function is applicable, or if required arguments are missing, the agent is instructed to explicitly state the limitation. The dialogue history starts from the initial user query (serialized in <user>), and is

extended across steps by appending the agent’s intermediate reasoning <think>, the predicted action <tool_call>, and the diagnostic feedback <tool_response>.

System Prompt

System

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.

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:

<tool_call>

....

</tools>

For each function call, return a json object with function name and arguments within <tool_call></tool_call> XML tags:

<tool_call>

{"name": <function-name>, "arguments": <args-json-object>}

</tool_call>

Case study: misguided repair vs. targeted recovery. This example illustrates the dominant failure mode analyzed in Section 4.3: *misguided repair*. Both the backbone and PRFC first locate the target file with find (./project/analysis_report.csv), so the remaining challenge is not initial tool selection but how to use execution feedback to complete the repair. After receiving No such file or directory, the backbone continues interacting but revises the wrong part of the action. It alternates between destination paths such as ./archive and ../archive while leaving the execution context and file-state assumptions unchecked. Thus, its repeated failed attempts are a symptom of misguided repair: the agent is persistent, but the persistence does not advance the interaction.

PRFC instead turns the same interaction history into a targeted repair sequence. It first satisfies the precondition implied by the user request by moving into the file’s directory (cd project), treats mkdir: File exists as a non-fatal observation, and then uses the tool-specific error to adjust the destination argument from archive/ to archive. This behavior matches the main-text claim that PRFC improves recovery by choosing corrective actions that advance the task, rather than by merely increasing the number of attempts.

Case study: premature abandonment vs. evidence-seeking recovery. This example illustrates a second recovery-relevant failure mode from Section 4.3: *premature abandonment*. In this file-retrieval task, both models enter the project directory and initially search for “Kelly’s test report” using `find`. The resulting empty match list is weak feedback: it does not distinguish a truly missing file from a mismatch between the user’s description and the actual filename. The backbone treats this first negative observation as sufficient evidence to stop, without inspecting the directory or testing plausible filename variants. The task is therefore abandoned even though additional evidence could still recover the answer.

PRFC treats the same weak feedback as a cue for targeted evidence gathering. It lists the directory contents with `ls`, maps the user’s description to the plausible candidate `test_report.docx`, and verifies the hypothesis with `cat`. This contrast supports the efficiency analysis in the main text: successful recovery requires actions that add information and move the task forward, not arbitrary retries or early termination after non-diagnostic feedback.

Case Study (Backbone): Misguided repair under execution feedback

User: Find analysis_report.csv and upon locating it, move it to archive/ in the same directory for safekeeping.

Step 0: Plan formulation for the query

Assistant: **<think>** Okay, the user wants me to find a file called analysis_report.csv and move it to an 'archive' directory. Let me break this down.....So the plan is: 1. Find the file using find with name analysis_report.csv. 2. If found, check if archive exists. If not, create it. 3. Move the file to archive using mv.....So the first step is to find the file. Let's start with that. **</think>**

<tool_call> {"name": "find", "arguments": {"path": ".", "name": "analysis_report.csv"}} **</tool_call>**

Environment: **<tool_response>** {"matches": ["/project/analysis_report.csv"]} **</tool_response>**

Step 1: Successful execution of the initial step

Assistant: **<think>** Okay, the user wants to move the analysis_report.csv file to an 'archive' directory. First, I found the file at "/project/analysis_report.csv". Now, I need to check if the 'archive' directory exists. If it doesn't, I should create it using the mkdir tool. Then, use the mv command to move the file into the archive directory.....First, create the archive directory if necessary. Then move the file. **</think>**

<tool_call> {"name": "mkdir", "arguments": {"dir_name": "archive"}} **</tool_call>**

Environment: **<tool_response>** None **</tool_response>**

Step 2: Attempting to execute the second step of the plan

Assistant: **<think>** Okay, the user wants to move the analysis_report.csv file to the 'archive' directory. I first used the find tool to locate the file, which was found at "/project/analysis_report.csv". Then, I tried to create the 'archive' directory using mkdir, but the response was None, which might mean the directory was already created or the operation was successful. Next, I need to move the file. The correct path for the archive directory is "/archive", so I'll use the mv tool. The source is "/project/analysis_report.csv" and the destination is "/archive/analysis_report.csv". Let's execute that. **</think>**

<tool_call> {"name": "mv", "arguments": {"source": "/project/analysis_report.csv", "destination": "/archive/analysis_report.csv"}} **</tool_call>**

Environment: **<tool_response>** {"error": "mv: cannot move '/project/analysis_report.csv': No such file or directory"} **</tool_response>**

Step 3: Attempting error recovery based on execution feedback

Assistant: **<think>** Okay, the user is trying to move the analysis_report.csv file to the 'archive' directory. Let me check what's going on here.....Therefore, the source is "/project/analysis_report.csv", and the destination should be "/archive/analysis_report.csv". Let me adjust the mv command to use the correct path. So the source is correct, but the destination should be "/archive/analysis_report.csv" instead of "/archive/analysis_report.csv". That might resolve the error. Let me try that. **</think>**

<tool_call> {"name": "mv", "arguments": {"source": "/project/analysis_report.csv", "destination": "/archive/analysis_report.csv"}} **</tool_call>**

Environment: **<tool_response>** {"error": "mv: cannot move '/project/analysis_report.csv': No such file or directory"} **</tool_response>**

Step 4: Failed recovery through misguided path revision

Assistant: **<think>** Okay, the user is trying to move the analysis_report.csv file to an 'archive' directory. Let me see what's going on here.....Alternatively, maybe the 'archive' directory is in the same directory as the project. So the destination is "/archive/analysis_report.csv". But the current working directory is the project directory, so the destination would be "/archive/analysis_report.csv". Let me try that again. **</think>**

<tool_call> {"name": "mv", "arguments": {"source": "/project/analysis_report.csv", "destination": "/archive/analysis_report.csv"}} **</tool_call>**

Environment: **<tool_response>** {"error": "mv: cannot move '/project/analysis_report.csv': No such file or directory"} **</tool_response>**

Step 11: Persistent misguided repair loop

Assistant: **<think>** Okay, the user is trying to move the analysis_report.csv file to an 'archive' directory. Let me see what's going on here.....Alternatively, maybe the 'archive' directory is in the same directory as the project. So the destination is "/archive/analysis_report.csv". But the current working directory is the project directory, so the destination would be "/archive/analysis_report.csv". Let me try that again. **</think>**

<tool_call> {"name": "mv", "arguments": {"source": "/project/analysis_report.csv", "destination": "/archive/analysis_report.csv"}} **</tool_call>**

Environment: **<tool_response>** {"error": "mv: cannot move '/project/analysis_report.csv': No such file or directory"} **</tool_response>**

Termination: Forced quit after 20 steps (no progress)

Case Study (PRFC): Effective Recovery from Execution Errors

User: Find analysis_report.csv and upon locating it, move it to archive/ in the same directory for safekeeping.

Step 0: Plan formulation for the query

Assistant: `<think>` Okay, the user wants me to find a file called analysis_report.csv and move it to an 'archive' directory. Let me break this down.....So the plan is: 1. Call find with name 'analysis_report.csv'. 2. If the file is found, call mv with source as the found file and destination as 'archive/'. But the 'mv' tool requires the source and destination. The source is the file found by 'find', and the destination is 'archive/'. So the first step is to use the 'find' tool. Let's proceed with that.

`</think>`

`<tool_call> {"name": "find", "arguments": {"path": ".", "name": "analysis_report.csv"}} </tool_call>`

Environment: `<tool_response> {"matches": ["/project/analysis_report.csv"]} </tool_response>`

Step 1: Successful execution of the initial step and formulating the next plan based on history

Assistant: `<think>` Okay, the user wants to move the found file to the 'archive' directory. The first step was to find the file, which was located at "/project/analysis_report.csv". Now, I need to move this file to the 'archive' directory.....But the 'mv' command's destination is relative to the current directory, which is 'project' after the 'cd' command. Therefore, moving 'analysis_report.csv' to 'archive/' would place it in the 'archive' directory within 'project'. Therefore, the correct tool calls are: 1. 'cd' to 'project'. 2. 'mkdir' 'archive'. 3. 'mv' 'analysis_report.csv' 'archive/'.....Therefore, the final answer is to move the file to 'archive/' in the 'project' directory. `</think>`

`<tool_call> {"name": "cd", "arguments": {"folder": "project"}} </tool_call>`

Environment: `<tool_response> {"current_working_directory": "project"} </tool_response>`

Step 2: Execution error due to existing directory

Assistant: `<think>` Okay, the user wants to move the analysis_report.csv file to the 'archive' directory. So far, I've found the file in the 'project' directory and changed the current working directory to 'project'. Now, I need to create the 'archive' directory if it doesn't exist and then move the file there.....Since I'm in the 'project' directory, the destination is relative. So the 'mv' command should work as expected. Let me make sure the parameters are correct. The 'mkdir' tool needs the 'dir_name' parameter, which is 'archive'. Then, the 'mv' tool needs 'source' as 'analysis_report.csv' and 'destination' as 'archive/'. I think that's all. Let's proceed with creating the directory and moving the file. `</think>`

`<tool_call> {"name": "mkdir", "arguments": {"dir_name": "archive"}} </tool_call>`

Environment: `<tool_response> {"error": "mkdir: cannot create directory 'archive': File exists"} </tool_response>`

Step 3: Failed recovery attempt with effective exploration and tool adjustment based on feedback

Assistant: `<think>` Okay, the user tried to create the 'archive' directory but got an error saying "File exists". That means the 'archive' directory already exists. So I don't need to create it again. Now, I can proceed to move the 'analysis_report.csv' file into the 'archive' directory. Since I'm already in the 'project' directory, the 'mv' command should be straightforward. The source is 'analysis_report.csv' and the destination is 'archive/'. Let me call the 'mv' tool with those parameters. If the move is successful, the file should be in the 'archive' directory as intended. `</think>`

`<tool_call> {"name": "mv", "arguments": {"source": "analysis_report.csv", "destination": "archive/"}} </tool_call>`

Environment: `<tool_response> {"error": "mv: no path allowed in destination. Only file name and folder name is supported for this operation."} </tool_response>`

Step 4: Successful error recovery after two rounds of tool and parameter adjustments based on feedback

Assistant: `<think>` Okay, let's see. The user is trying to move the analysis_report.csv file to the 'archive' directory. The previous steps were: find the file, change to the 'project' directory, try to create 'archive' (but it already exists), and then attempt to move the file. But the last step failed with an error about no path allowed in the destination. Hmm, the error message says "no path allowed in destination. Only file name and folder name is supported for this operation." So the 'mv' tool might not accept a path like 'archive/' as the destination.....So the correct destination is 'archive', not 'archive/'. Let me try that. The source is 'analysis_report.csv' and the destination is 'archive'. That should move the file into the 'archive' directory without using a path. Let's try that. `</think>`

`<tool_call> {"name": "mv", "arguments": {"source": "analysis_report.csv", "destination": "archive"}} </tool_call>`

Environment: `<tool_response> {"result": "analysis_report.csv moved to 'archive/analysis_report.csv'"} </tool_response>`

Step 5: Task completion and confirmation

Assistant: `<think>` Okay, the user wanted to move the analysis_report.csv file to the 'archive' directory. Let's recap the steps taken so far. First, the file was found in the 'project' directory. Then, the current working directory was changed to 'project'. The 'archive' directory was attempted to be created, but it already existed. Then, the 'mv' command was used to move the file, but there was an error about paths not being allowed in the destination. After adjusting the destination to just 'archive' instead of 'archive/', the move was successful, and the file was moved to 'archive/analysis_report.csv'. Now, the user's goal is achieved. The file is safely stored in the 'archive' directory. The final answer should confirm that the file was moved successfully. `</think>`

The file analysis_report.csv has been successfully moved to the archive directory.

Environment: `<tool_response>` Empty response from the model. Proceed to next turn</tool_response>

Case Study (Backbone): Premature abandonment under rigid reasoning

User: Please cd into project folder and find Kelly's test report somewhere in the directory and read the content to me.

Step 0: Plan formulation and initial navigation

Assistant: `<think>` Okay, the user wants me to change into the "project" directory and find Kelly's test report. Let me break this down step by step..... So I'll proceed with the steps: cd into project, find the file, then cat it. `</think>`

`<tool_call>` {"name": "cd", "arguments": {"folder": "project"}} `</tool_call>`

Environment: `<tool_response>` {"current_working_directory": "project"} `</tool_response>`

Step 1: Attempting to locate specific file

Assistant: `<think>` Okay, the user wants me to find Kelly's test report in the project directory. Let me use the find tool for that. The path should be the current directory, which is "project", and the name is "Kelly's test report". Wait, the find tool's parameters are path and name. Since I'm already in the project directory, the path is ".", and the name is "Kelly's test report". Let me call the find tool with those arguments. If the find tool returns a match, I can then use the cat tool to read the content. If it doesn't find the file, I'll need to inform the user. But let's proceed with the search first. `</think>`

`<tool_call>` {"name": "find", "arguments": {"path": ".", "name": "Kelly's test report"}} `</tool_call>`

Environment: `<tool_response>` {"matches": []} `</tool_response>`

Step 2: Premature abandonment without exploration or iterative repair attempts

Assistant: `<think>` Okay, the user asked to find Kelly's test report in the project directory. I used the find tool with the path set to the current directory (project) and the name "Kelly's test report". (The response came back with an empty matches list, meaning the file wasn't found.....Since the find tool didn't find it, I should inform the user that the file wasn't located.) But I can also suggest possible next steps, like checking for typos or searching deeper into the directory. Let me make sure to mention that the search was thorough but didn't find the file, and offer assistance in case they have more details. `</think>`

(The file "Kelly's test report" could not be found in the current directory or its subdirectories.) Possible reasons: 1. The file name is slightly different (e.g., case sensitivity, typos). 2. The file is in a nested subdirectory. 3. The file does not exist in the current workspace. Would you like to search for a different name, check subdirectories, or verify the directory structure?

Environment: `<tool_response>` Empty response from the model. Proceed to next turn `</tool_response>`

Case Study (PRFC): Adaptive File Search and Retrieval

User: Please cd into project folder and find Kelly's test report somewhere in the directory and read the content to me.

Step 0: Plan formulation and initial navigation

Assistant: <think> Okay, the user wants me to navigate into the "project" folder and find Kelly's test report. Let me break this down step by step..... Let me structure the tool calls: first cd into project, then find for the file, then cat to read it. Each step depends on the previous one. If any step fails, I need to handle that, but the user hasn't mentioned any errors yet, so I'll proceed. </think>

<tool_call> {"name": "cd", "arguments": {"folder": "project"}}</tool_call>

Environment: <tool_response> {"current_working_directory": "project"}</tool_response>

Step 1: Attempting to locate specific file

Assistant: <think> Okay, the user wants me to find Kelly's test report in the project directory. Let me use the find tool for that. I'll set the path to the current directory (.) and the name to "Kelly's test report". Since the user said "somewhere in the directory", I should search recursively. Wait, the find tool's parameters require path and name. The path is currently "project", so I'll set path to "." and name to "Kelly's test report". Let me make sure the parameters are correct. The response should give me the matching paths. Once I have the file path, I can then use the cat tool to read the content. Let's start with the find command. </think>

<tool_call> {"name": "find", "arguments": {"path": ".", "name": "Kelly's test report"}} </tool_call>

Environment: <tool_response> {"matches": []} </tool_response>

Step 2: Strategy adjustment due to missing file

Assistant: <think> Okay, the find command didn't return any matches for "Kelly's test report". Hmm, maybe the file name is different or the file isn't in the project directory. Let me check if the user meant a different name or if there's a typo. Alternatively, maybe the file is in a subdirectory. I should try searching again with a more general approach or check the current directory contents again. Wait, the user said "somewhere in the directory", so perhaps I should search all subdirectories. Let me try the find command again, maybe with a different parameter or check if the file exists in the current directory. If it's not found, I might need to inform the user that the file isn't present. </think>

<tool_call> {"name": "ls", "arguments": {"a": true}} </tool_call>

Environment: <tool_response> {"current_directory_content": ["file1.txt", "file2.txt", "test_report.docx"]} </tool_response>

Step 3: Verifying potential candidate file

Assistant: <think> Okay, the user asked for Kelly's test report, and after using the find tool, it wasn't found. Then I used ls with the -a flag to list all files, including hidden ones. The response shows files like file1.txt, file2.txt, and test_report.docx. Since the user mentioned "Kelly's test report", maybe the actual file is named test_report.docx. I should check if that's the case. Let me use the cat tool to read the content of test_report.docx to see if it's the desired file. If it is, then proceed to read it. If not, inform the user that the file wasn't found. </think>

<tool_call> {"name": "cat", "arguments": {"file_name": "test_report.docx"}}</tool_call>

Environment: <tool_response> {"file_content": "Kelly Total Score: 96"}</tool_response>

Step 4: Task completion and Final answer generation

Assistant: <think> Okay, the user wanted Kelly's test report. I first tried to find the file using the find tool but didn't get any results. Then I listed the current directory contents with ls -a and saw a file called test_report.docx. I assumed that might be Kelly's report and used cat to read it. The content showed "Kelly Total Score: 96", which seems to be the test results. Since the user didn't specify the exact file name, this must be the correct document. I'll present the content as the answer. </think>

The content of Kelly's test report is: **Kelly Total Score: 96**. This appears to be the test results for Kelly. Let me know if you need anything else!