TALE: Tool-using Agent Learning from Error

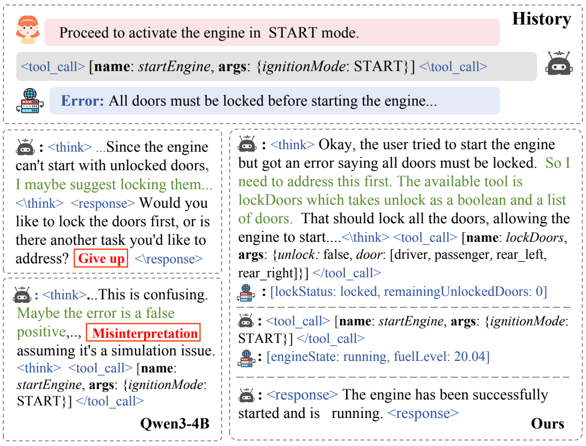
Anonymous ACL submission

Abstract

Large language models equipped with external tools have shown strong potential as general-purpose agents, yet they remain brittle when tool execution fails. Existing approaches largely rely on heuristic self-correction or imitation of error-fix patterns, making it difficult for agents to reliably recover from tool-calling errors, especially in multi-step settings. In this paper, we present TALE, a reinforcement learning framework that enables LLM-based agents to acquire robust self-correction behaviors by explicitly learning from their own tool-calling errors. TALE formulates error recovery as a sequential decision-making problem and introduces a progress-aware reward that captures incremental improvement across consecutive repair attempts, rather than relying solely on sparse success signals. This design encourages directional exploration and discourages repetitive or unproductive behaviors. Extensive experiments on four representative tool-using benchmarks demonstrate that TALE consistently improves task success and correction efficiency, achieving significant gains over strong baselines on challenging settings such as BFCL-v3. Our results show that learning from error experience is a key step toward more reliable and adaptive tool-using agents.

1 Introduction

Large language models (LLMs) have demonstrated remarkable capacity as general-purpose reasoning engines, achieving strong performance across a wide range of tasks. 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., 2023; Liu et al., 2024b,a). This paradigm fundamentally extends LLMs from passive sequence predictors into agentic systems. Notably, tool use is especially critical for smaller LLMs, whose limited parametric knowledge must be complemented by external tools (Li et al., 2025c; Wang et al., 2024; Li et al., 2025b; Shen et al., 2024; Erdogan et al., 2024).



Proceed to activate the engine in START mode.

<tool_call> [**name**: *startEngine*, **args**: {*ignitionMode*: START}] <\tool_call>

Error: All doors must be locked before starting the engine...

: <think> ...Since the engine can't start with unlocked doors, I maybe suggest locking them... <\think> <response> Would you like to lock the doors first, or is there another task you'd like to address? <\response>

: <think>...This is confusing. Maybe the error is a false positive,.., assuming it's a simulation issue. <think> <tool_call> [**name**: *startEngine*, **args**: {*ignitionMode*: START}] </tool_call>

: <think> Okay, the user tried to start the engine but got an error saying all doors must be locked. So I need to address this first. The available tool is lockDoors which takes unlock as a boolean and a list of doors. That should lock all the doors, allowing the engine to start....<\think> <tool_call> [**name**: *lockDoors*, **args**: {*unlock:* false, *door*: [driver, passenger, rear_left, rear_right]}] </tool_call>

: [lockStatus: locked, remainingUnlockedDoors: 0]

: <tool_call> [**name**: *startEngine*, **args**: {*ignitionMode*: START}] </tool_call>

: [engineState: running, fuelLevel: 20.04]

: <response> The engine has been successfully started and is running. <response>

Figure 1: Self-correction failures in current LLMs. Left: Qwen3-4B correctly identifies the unmet precondition but fails to recover. Right: TALE identifies the unmet precondition and executes the required recovery steps.

A defining characteristic of tool-using agents is their operation within an interactive environment: each action produces immediate feedback from the tool, which is then incorporated into the agent's subsequent reasoning and decision making. In particular, error feedback plays a central role in enabling self-correction for tool use. By exposing mismatches between intended 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 these advances, current LLM-based tool-using agents still struggle to effectively leverage tool error feedback for robust self-correction. 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. This variability, combined with incomplete or underspecified feedback, makes it difficult for models to infer how their actions should be revised. Thus, agents frequently fail to extract actionable guidance from prior failures, leading to brittle behaviors such as repeatedly issuing the same incorrect calls or prematurely abandoning tasks that are in fact recoverable. Figure 1 illustrates these failure patterns in a representative example.

Recent empirical studies further corroborate this limitation. For example, Huang et al. (2025) shows that even advanced LLMs struggle to recover from tool-calling errors, with recovery success rates capped at 29.47% on the BFCL-v3 benchmark (Patil et al., 2024) and dropping to only 17.39% on API-Bank (Li et al., 2023). These results suggest that self-correction under tool feedback remains a fundamental bottleneck rather than a solved capability. Critically, many existing approaches treat self-correction as pattern matching, i.e., imitating observed error-fix trajectories, rather than as a principled learning problem (Polyakov et al., 2025; Cui et al., 2025; Sun et al., 2024). This raises a key question: **how can a tool-using agent systematically learn from its own error experiences to improve future actions?**

To address this challenge, we propose **TALE** (Tool-using Agent Learning from Error experience), a learning framework that enables tool-using agents to acquire robust self-correction behaviors by explicitly learning from their own error trajectories. Rather than treating tool execution feedback as isolated signals or mimicking fixed error-fix patterns, TALE formulates self-correction as a sequential decision-making problem, where each recovery attempt constitutes an exploration step guided by structured feedback and cumulative progress.

At the core of TALE is a progress-aware reward formulation: First, instead of relying solely on binary success signals, we adopt a fine-grained tool-using stepwise scoring scheme that decomposes each tool call into semantically meaningful components (Qian et al., 2025). Then, TALE goes beyond step-level feedback by explicitly modeling trajectory-level progress. We introduce a progress-based shaping signal that rewards improvements across consecutive attempts, measured by relative gains in fine-grained tool-using scores along a trajectory. This design encourages agents to pursue recovery actions that move them closer to successful execution, while discouraging unproductive behaviors such as repeating identical errors or prematurely abandoning partially correct solutions.

By explicitly incentivizing monotonic improvement, TALE transforms self-correction from unguided trial-and-error into a learnable policy that internalizes persistence and directional exploration. We optimize this objective using reinforcement learning, enabling agents to discover effective recovery strategies through direct interaction with tool feedback rather than through rote imitation.

Finally, we demonstrate the effectiveness of TALE through extensive experiments on four representative tool-using benchmarks spanning multi-turn function calling and API-based reasoning. Across all settings, TALE consistently improves both task success and correction efficiency. On the challenging BFCL-v3 benchmark, TALE achieves a +6.4% absolute improvement in task success over strong baselines, while reducing the average number of attempts required to reach a correct solution. These results indicate that TALE not only enables agents to recover from errors more reliably, but also allows them to do so more efficiently by learning targeted and productive repair strategies.

2 Related work

2.1 RL for Tool-Using Agents

Equipping LLMs with the ability to invoke external tools has become a central paradigm for building agentic systems (Li et al., 2025a; Singh et al., 2025; Wei et al., 2025). Early approaches primarily relied on supervised fine-tuning over curated tool-use traces, which limits generalization to unseen tools or failure modes. More recent work has shifted toward reinforcement learning (RL), enabling agents to learn from outcome-based feedback obtained through interaction with tools and environments (Zhang et al., 2025c; Qiao et al., 2024).

Within this line of research, GRPO-style and related policy optimization methods (Zhang et al., 2025a) have been widely adopted. These methods sample diverse trajectories and optimize policies using terminal rewards that reflect overall task success or failure. To alleviate the sparsity of binary rewards, subsequent work has explored more fine-grained reward designs. For example, Qian et al. (2025) proposes decomposable scoring schemes that evaluate tool calls along multiple semantic dimensions, providing richer supervision signals. Other studies extend RL-based tool use to more complex settings, including hierarchical tool coordination (Dong et al., 2025) and multi-turn or conversational function calling (Zhao et al., 2025;

Tan et al., 2025; Wang et al., 2025).

Despite these advances, most RL-based tool-using agents remain primarily optimized for final task completion, with learning signals applied at the trajectory level or at individual tool calls in isolation. In contrast, TALE explicitly targets this gap by framing error recovery itself as a learnable sequential decision process, guided by progress-aware rewards over error trajectories.

2.2 Self-Correction in LLMs

Self-correction has emerged as a key mechanism for improving the reliability of LLM outputs, particularly in reasoning and tool-use settings. Early work focused on inference-time strategies, such as prompting models to reflect on or revise their previous answers (Yao et al., 2023; Madaan et al., 2023). Subsequent approaches introduce external verifiers or critics, i.e., either separate models or heuristic checkers, to identify errors and trigger revisions, reducing unnecessary repeated attempts and improving efficiency (Shinn et al., 2023).

However, these methods typically depend on explicit external feedback or well-defined correctness signals. When such signals are weak, ambiguous, or unavailable (as is often the case with real-world tools), models struggle to reliably assess their own failures. To address this limitation, recent work has explored internalizing self-correction through training. Some approaches optimize tool documentation (Qu et al., 2024; Yuan et al., 2025; Shi et al., 2025) or instruction-following behavior (Dong et al., 2024; Gou et al., 2024) using execution feedback, while others learn corrective patterns directly from error–fix trajectories (Zhang et al., 2025b). Although effective for recurring or well-characterized errors, these methods remain largely reactive, relying on observed fixes rather than discovering new recovery strategies.

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 explicit rewards (Huang et al., 2025). While such methods can capture general heuristics for robustness, they do not explicitly model multi-step error recovery or provide directional guidance toward improvement within a failed trajectory. TALE departs from prior self-correction paradigms by treating error recovery as a first-class learning objective. Instead of imitating fixes or relying on external critics, TALE leverages reinforcement learning with fine-grained, trajectory-aware rewards that explicitly encode progress across consecutive attempts. This formulation enables agents to learn systematic, multi-step recovery strategies driven by incremental improvement.

3 Methodology

We present TALE, which enables agents to learn effective self-correction through multi-step interactions. We begin with formalizing the multi-step interaction process (§3.1), then introduce progress-aware reward formulation (§3.2), and finally describe policy optimization (§3.3).

3.1 Rollout with Multi-step Interaction

Consistent with our goal of self-correction as a sequential decision-making problem, we formalize this process as a multi-step rollout in which each step corresponds to one recovery attempt, rather than treating tool feedback as isolated signals or relying on fixed error-fix templates. At each step $t \in \{1, \dots, T\}$, the policy π_θ first generates an action $a_t = (a_t^{\text{think}}, a_t^{\text{tool}})$ conditioned on the initial prompt q and the decision-making interaction history h_{t-1}. Each action consists of a reasoning trace a_t^{think} (wrapped in `<think> </think>` tags) and a set of tool calls $a_t^{\text{tool}} = \{c_1, \dots, c_n\}$ (wrapped in `<tool_call> </tool_call>` tags). Then, a defined environment $\mathcal{E}$ (see Appendix E) executes a_t, and returns an observation $o_t = \mathcal{E}(a_t)$ containing structured feedback such as successful outputs, error messages, or execution status (wrapped in `<tool_response> </tool_response>` tags). The history is updated as $h_t = h_{t-1} \oplus (a_t, o_t)$, where $\oplus$ denotes concatenation, and the process continues until the maximum horizon T is reached, or an early stopping criterion is met.

The resulting sequence of actions and observations constitutes a self-correction trajectory $\tau = (q, a_1, o_1, \dots, a_T, o_T)$. The policy defines a probability distribution over this trajectory as

$$\pi_\theta(\tau \mid q) = \prod_{t=1}^{T} \pi_\theta(a_t \mid q, h_{t-1}), \qquad (1)$$

where $\pi_\theta(a_t \mid q, h_{t-1})$ represents the probability of generating the tokens in action a_t.

3.2 Progress-aware Reward Formulation

To enable the agent to learn effective self-correction strategies from error feedback, we require a reward signal that is sensitive to partial progress. To this end, we design a progress-aware reward that (i) provides fine-grained step-wise scores beyond binary

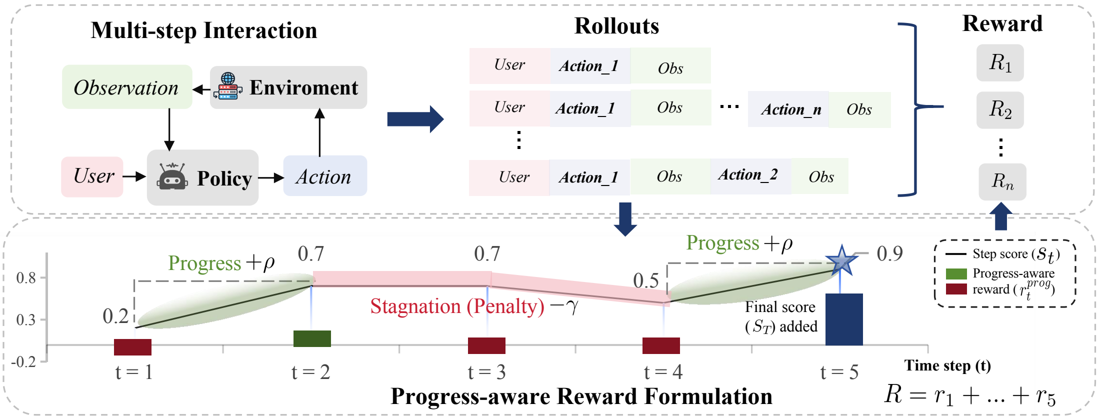


Figure 2: Overview of TALE: we generate multi-step rollouts and optimize the policy with a progress-aware reward that captures incremental improvement across consecutive recovery attempts.

success signals, and (ii) explicitly shapes trajectory-level progress to guide directional exploration.

3.2.1 Fine-grained Stepwise Scoring

Following the fine-grained tool-using stepwise scoring scheme (Qian et al., 2025), we score each tool call by decomposing it into semantically meaningful components. Let $c = (n, P)$ be a predicted tool call with name n and parameter map P, and $c^* = (n^*, P^*)$ be the ground-truth call. We define a matching score as

$$S(c, c^*) = \underbrace{\mathbf{1}[n = n^*]}_{\text{name}} + \underbrace{\frac{|K \cap K^*|}{|K \cup K^*|}}_{\text{key}} + \underbrace{\sum_{k \in K \cap K^*} \mathbf{1}[v_k = v_k^*]}_{\text{value}} \qquad (2)$$

where K is the set of parameter keys (with $K \subseteq P$), and v_k is the corresponding value for each $k \in K$. This design assigns credit for correct tool selection, partial credit for key overlap (Jaccard similarity), and additional credit for correctly matched values, enabling learning signals even when only parts of a call are correct. To handle actions with multiple tool calls, we match predicted and ground-truth calls using a greedy matching algorithm and aggregate the results with an F1-style normalization to obtain a step score $s_t \in [0, 1]$, balancing completeness and accuracy across attempts (details in Appendix D). Actions with syntax errors are assigned $s_t = 0$, strictly penalizing formatting mistakes.

3.2.2 Trajectory-level Progress-based Shaping

Stepwise scores quantify per-attempt quality but do not enforce improvement across attempts. We therefore convert $\{s_1, \ldots, s_T\}$ into progress-aware rewards $\{r_1, \ldots, r_T\}$ to promote monotonic progress and suppress redundant retries. First, a per-step cost $r_{\text{step}} = -\lambda$ ($\lambda > 0$) is introduced to promote recovery efficiency. To capture trajectory-level progress, the running maximum $\bar{s}_t = \max\{s_1, \ldots, s_t\}$ is tracked, initialized with $\bar{s}_0 = -1$, and the relative gain is defined as

$$g_t = \frac{\max(0, s_t - \bar{s}_{t-1})}{1 - \bar{s}_{t-1} + \epsilon}, \qquad (3)$$

where ϵ is a small constant for numerical stability. We then assign

$$r_t^{\text{prog}} = \begin{cases} +\rho \cdot g_t & \text{if } s_t > \bar{s}_{t-1} \text{ (progress)} \\ -\gamma & \text{otherwise (stagnation),} \end{cases} \qquad (4)$$

where $\rho, \gamma > 0$. This shaping signal rewards improvements proportional to their relative gains, encouraging persistent and directional exploration, while penalizing stagnation to discourage unproductive retry loops. The complete immediate reward at step t is

$$r_t = r_{\text{step}} + r_t^{\text{prog}} + \mathbf{1}[t = T] \cdot s_T, \qquad (5)$$

This formulation encourages the agent to reach correct solutions in fewer attempts while maintaining monotonic progress, by maximizing the cumulative return $R = \sum_{t=1}^{T} r_t$.

Early Stopping. For efficiency, we implement early stopping: interaction terminates if $s_t \leq \bar{s}_{t-1}$ for k consecutive steps, pruning trajectories where the agent fails to recover from stagnation.

3.3 Policy Optimization

Following (Qian et al., 2025; Zhang et al., 2025c), we optimize TALE using reinforcement learning by maximizing the expected trajectory return

$\mathbb{E}\tau \sim \pi\theta[R(\tau)]$ with Group Relative Policy Optimization (GRPO) (Shao et al., 2024). By normalizing advantages within a group of sampled trajectories, GRPO stabilizes training against the high variance inherent in multi-step interactions and effectively distinguishes between productive recovery strategies and unproductive trials. Formally, for each query q sampled from the dataset $\mathcal{D}$, we generate a group of G progressive trajectories $\{\tau_i\}_{i=1}^{G}$ by rolling out the policy $\pi_{\theta_{\text{old}}}$. Each trajectory τ_i contains T_i recovery steps before termination. We compute the trajectory return $R_i = \sum_{t=1}^{T_i} r_t^i$ and a group-normalized advantage A_i to measure its relative quality within the group

$$A_i = \frac{R_i - \text{mean}(\{R_j\}_{j=1}^{G})}{\text{std}(\{R_j\}_{j=1}^{G} + \epsilon)} \tag{6}$$

where ϵ is a small constant for stability. The policy is updated by maximizing the $\mathcal{J}_{\text{GRPO}}(\theta)$ objective over the tokens of the generated recovery trajectory

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{q\sim\mathcal{D},\, \{\tau_i\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(\cdot|q)} \Bigg[\frac{1}{G} \sum_{i=1}^{G} \min\Big(\rho_i(\theta)\, A_i, \\ \text{clip}\big(\rho_i(\theta),\, 1-\epsilon,\, 1+\epsilon\big)\, A_i\Big) - \beta\, \mathrm{D}_{\mathrm{KL}}\big(\pi_\theta \,\|\, \pi_{\text{ref}}\big) \Bigg], \tag{7}$$

where $\rho_i(\theta) = \pi_\theta(\tau_i \mid q)/\pi_{\theta_{\text{old}}}(\tau_i \mid q)$ is the trajectory-level likelihood ratio (computed via the token likelihood of the entire generated trajectory). The clipping term prevents overly large policy updates, while the KL regularization constrains drift from the initial supervised policy π_{ref}, yielding stable optimization under our progress-aware reward. We also adopt GSPO (Zheng et al., 2025) as an alternative policy objective.

Data Curation. Prior work on reinforcement learning for LLMs and tool-using agents has shown that careful data curation is critical for stable optimization and sample-efficient learning (Yu et al., 2025; Yue et al., 2025; Feng et al., 2025). Motivated by this, we target a training regime where self-correction is both necessary and learnable—neither trivially easy (often solved in one attempt) nor intractably hard (unlikely to be recover despite repeated attempts). The training data is curated from Qian et al. (2025) via difficulty-based filtering. For each query, difficulty is estimated by running G=20 rollouts with the reference policy π_{ref} and computing the empirical failure rate. We retain queries with $0.01 < f < 0.6$ (failure rate), yielding a compact training set of ∼1.4k examples spanning ∼7.6k unique tools. This curation focuses learning on tasks where self-correction matters most, filtering out instances that are solved in a single attempt as well as those that rarely succeed even under extensive exploration. By reducing data-level variance in trajectory returns, the curated set stabilizes policy gradients and improves sample efficiency.

4 Experiments

4.1 Experimental Setup

Evaluation Benchmarks. We evaluate TALE on four complementary benchmarks that collectively assess tool-using agents under interactive tool execution, controlled multi-turn dialogue, and compositional reasoning. BFCL-v3 is a comprehensive tool-use benchmark (4.4k+ cases) spanning single-step reasoning, irrelevant tool rejection, and multi-tool composition, with a focus on multi-turn interaction and feedback-driven self-correction in realistic settings (Patil et al., 2024). API-Bank is a multi-turn dialogue benchmark (∼600 cases) across three difficulty levels (Li et al., 2023). T-Eval includes 23k+ cases spanning instruction following, planning, reasoning, retrieval, understanding, and review (Chen et al., 2024). Nestful focuses on executable nested API sequences (1.8k+ cases), capturing dependency structures where outputs from earlier calls are consumed as inputs to subsequent ones (Basu et al., 2025).

Baselines. We compare representative baselines along two complementary lines: step-level methods and feedback-driven methods. (i) *Step-level methods:* These baselines optimize individual decisions within a single attempt, without explicitly learning a directed recovery process from execution errors. Supervised Fine-Tuning (SFT) fine-tunes the backbone on curated tool-use trajectories, serving as a straightforward imitation baseline. ToolRL (Qian et al., 2025) improves tool calling via fine-grained, stepwise scoring. We further include GRPO and GSPO as policy-optimization baselines trained with our step-wise reward variant under the same backbone, isolating optimizer effects under a controlled reward signal. (ii) *Feedback-driven methods:* These baselines explicitly leverage tool feedback, but primarily acquire correction behaviors by imitating error–fix templates rather than optimizing a learnable recovery policy. Specifically, ToolReflection (Qian et al., 2025) captures reflection patterns by mimicking static error-correction trajectories, whereas Early Experience (Zhang et al.,

Method	**Param.**	**Multi Turn**					**Single Turn**		**Hal.**	**Overall**
		Base	Miss Func	Miss Param	Long Context	Overall	Non-Live	Live		
Open Source Model										
GPT-5.2-2025-12-11	/	36.50	18.00	27.50	30.50	28.12	81.85	70.39	77.21	59.73
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
Step-level Methods										
Qwen3-4B (Backbone)	4B	39.50	32.00	19.00	22.00	28.10	87.40	79.79	78.21	64.67
+ SFT	4B	33.50	34.50	24.50	19.50	27.98	87.79	79.70	78.39	64.75
+ GRPO	4B	35.00	31.50	23.00	19.00	27.10	88.52	80.19	80.96	64.74
+ GSPO	4B	31.50	33.00	26.00	21.50	28.00	84.04	78.61	78.72	65.21
+ ToolRL	4B	26.50	17.00	17.50	22.50	20.89	88.48	79.52	70.28	62.50
Feedback-driven Methods										
+ ToolReflection	4B	36.50	37.00	26.50	22.00	30.50	86.73	83.33	79.19	65.33
+ Early Experience	4B	35.00	34.00	24.00	23.50	29.13	87.73	80.01	72.91	65.24
+ TALE (GRPO)	4B	41.00	30.50	27.50	25.00	30.89	**88.71**	80.88	75.94	66.02
+ TALE (GSPO)	4B	43.00	36.00	30.00	29.00	34.50	88.44	80.50	78.72	**67.39**

Table 1: Performance on BFCL-v3, where TALE achieves the highest overall score. Open-source results are from the official leaderboard (Dec. 16, 2025); TALE and our re-implemented step-level and feedback-driven baselines are initialized from Qwen3-4B for a fair comparison. “Hal.” denotes Hallucination. Best/second-best are bold/underlined.

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

Table 2: Performance on additional benchmarks. Overall is the mean across the three benchmarks.

2025b) approximates environment dynamics by predicting tool responses conditioned on the dialogue history. Details are provided in Appendix B.

Backbone. All baselines and TALE are initialized from Qwen3-4B (Yang et al., 2025) to ensure a controlled comparison. Qwen3-4B provides a strong yet lightweight foundation, with reliable instruction following and basic tool-use capability.

4.2 Main Results

As shown in Table 1, TALE achieves the best overall performance on BFCL-v3, despite open-source competitors being trained with larger backbones and substantially more supervised data. In particular, TALE (GSPO) reaches an overall score of 67.39, improving by +2.72 over the Qwen3-4B backbone and outperforming the strongest open-source baseline, ToolACE-MT (Zeng et al., 2025), by +1.98. In contrast, step-level methods yield only marginal changes in overall score (+0.07 to +0.54 over the backbone) and do not improve the multi-turn overall metric, suggesting that optimizing isolated tool-calling decisions is insufficient for long-horizon recovery. Feedback-driven baselines provide modest improvements on multi-turn, but remain well behind TALE, consistent with the limitation of relying on static error-fix imitation rather than learning a progress-aware recovery policy. Besides, TALE’s improvements are robust to the choice of optimizer: TALE (GRPO) and TALE (GSPO) consistently outperform their corresponding step-level counterparts by +1.28 and +2.18. These results suggest that the gains stem from leveraging error experiences to learn directionally improving self-correction rather than from a particular policy-optimization recipe.

We further evaluate TALE on three additional benchmarks: API-Bank, T-Eval, and Nestful (Table 2). TALE achieves superior performance on T-Eval and Nestful, which emphasize compositional tool reasoning and dependency-sensitive execution, while maintaining competitive results on API-Bank. These findings validate the advantage of formulating self-correction as a sequential decision-making problem, where each recovery attempt constitutes an exploration step and cumulative progress.

4.3 Progressive and Efficient Recovery

To validate that TALE performs effective self-correction rather than blind retries, we analyze agents’ recovery behavior on the BFCL-v3 multi-turn benchmark. In this setup, a agent is permitted up to 20 attempts per turn to iteratively invoke tools

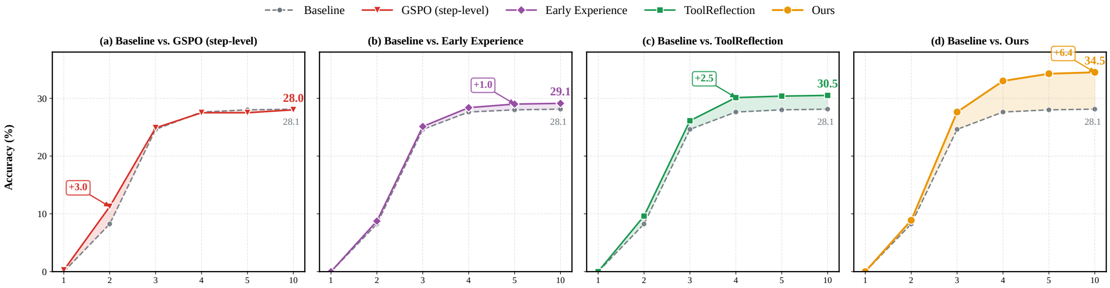


Figure 3: Promance on BFCL-v3 multi-turn subset. For each turn, we cap the evaluation-time attempt budget at k (early-terminating once a final answer is produced) and recompute SR@k.

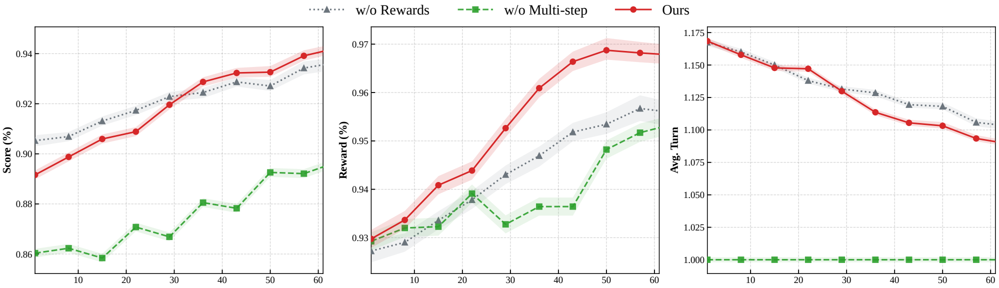


Figure 4: Training dynamics of TALE component ablations. Left: Score@1 (success under a single attempt). Middle: average episode return. Right: average interaction turns per episode. *w/o Multi-step* sets T=1, removing multi-attempt rollouts. *w/o Reward* keeps multi-step rollout but replaces the progress-aware reward with terminal binary success only.

and revise actions based on feedback before finalizing its response. This setting isolates a central question: whether the additional interaction budget yields incremental task progress and improved success rates, or merely amplifies repetitive failure.

Figure 3 reports SR@k, the fraction of turns solved when the evaluation-time attempt budget is capped at k. Small k measures early-step recoverability, whereas large k captures long-tail recovery enabled by accumulated feedback. The GSPO-optimized backbone (step-level) improves mainly in the first few attempts and then saturates, consistent with optimizing single-step success rather than targeted recovery. In contrast, ToolReflection and TALE continue to improve as k increases, indicating trajectory-level recovery that converts additional attempts into corrective actions using the growing error trajectories. TALE achieves the highest SR@20 and maintains an upward trend in the high-k regime, showing feedback-guided recovery behaviors rather than brute-force retries.

To summarize recovery efficiency under comparable interaction cost (full budget k=20 with early termination), Table 3 reports an attempt-normalized metric, SPA, which directly measures efficiency as the expected success yield per executed attempt (higher is better). Under this metric, TALE attains the best SPA, indicating that it converts each attempt into successful recovery more effectively than competing methods. Importantly, fewer executed attempts ($\bar{A}$) do not indicate higher efficiency: they may instead reflect premature abandonment rather than productive progress (Appendix F), where the agent terminates early without sustaining recovery over a longer horizon.

Method	**SR@20 (%)**	$\bar{A}$	**SPA ↑**
Qwen3-4B	28.1	2.5767	10.9
GSPO (step-level)	28.0	2.2488	12.5
ToolReflection	31.8	2.4722	12.9
Early Experience	29.1	2.5953	11.2
TALE	34.5	2.6029	**13.3**

Table 3: Recovery efficiency on the BFCL-v3 multi-turn subset. We report the terminal success rate (SR@20), the average number of executed attempts per turn ($\bar{A}$), and an attempt-normalized success metric SPA = SR(%)/$\bar{A}$, which measures success yield per attempt (higher is better).

4.4 Ablations and Controlled Variants

To explain why TALE can learn from its own error experiences, we disentangle which design choices are required to make feedback-guided recovery learnable. We first validate the necessity of TALE's core components via ablations. We then examine whether recovery strategies should be learned by optimizing individual recovery actions or by optimizing trajectory-level progress across attempts.

Component ablations. We begin by isolating the contributions of two core design choices in TALE: (i) multi-step rollouts with horizon $T >$

Module	Base	Miss Func	Miss Param	Long Context	Overall
Qwen-4B	39.5	32.0	19.0	22.0	28.1
w/o Multi-step	31.5	33.0	26.0	21.5	28.0
w/o Reward	35.0	31.0	29.5	24.0	29.9
TALE	**43.0**	**36.0**	**30.0**	**29.0**	**34.5**

Table 4: Component ablation results on BFCL-v3 multi-turn subsets. Overall is the mean across the four subsets.

1, which generate error trajectories for learning self-correction behaviors; and (ii) progress-aware rewards formulation, which assigns non-terminal credit for directional recovery. Thus, there are three variants compared in this ablation study: *w/o Multi-step* ($T{=}1$), *w/o Reward* (multi-step rollout with terminal binary success only), and the full TALE.

Figure 4 shows the training dynamics in terms of Score@1, average episode return, and the average number of interaction steps per episode. *w/o Multi-step* reduces episode return, indicating limited exposure to recoverable error trajectories. With the same multi-step rollout, removing progress-aware rewards (*w/o Reward*) also lowers return and increases interaction steps, suggesting less efficient exploration under sparse signals. In contrast, TALE simultaneously improves Score@1 and episode return while reducing interaction steps, reflecting a more directed recovery policy that converts feedback into incremental progress.

On BFCL-v3 multi-turn evaluation (Table 4), *w/o Multi-step* eliminates most gains and stays near the backbone, showing that multi-attempt exposure is necessary for turning tool feedback into effective improving recovery. Compared to *w/o Reward*, TALE improves the overall score by +4.6 points, with the largest gain on Base. Overall, multi-step interaction supplies learnable error trajectories, while progress-aware reward shapes how effectively the policy translates them into directionally improving recovery, yielding higher return with fewer attempts and stronger performance.

Trajectory-level vs. Action-level recovery learning. The ablations above demonstrate the importance of multi-step rollouts and progress-aware rewards, but they do not yet clarify whether trajectory-level progress modeling is necessary, or whether similar gains can be achieved with standard action-level reinforcement learning over a trajectory. In tool-using rollouts, agents must reason not only about individual actions, but also about whether successive attempts make directional progress toward a valid solution. To isolate this factor, we introduce StepGRPO, a controlled variant that keeps the rollout distribution and step-level scoring unchanged, while removing trajectory-level progress shaping. This comparison directly tests whether credit assignment across attempts is essential for learning effective recovery.

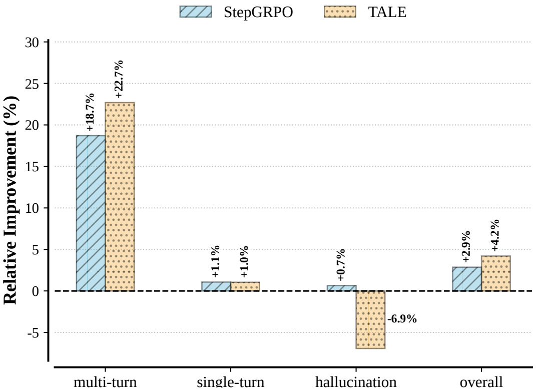


Figure 5: Relative improvement over the backbone.

Formally, consider a query q and a group of G sampled trajectories $\{\tau_i\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(\cdot \mid q)$, where trajectory τ_i contains T_i tool-use actions. Let $y_{i,t}$ denote the token span of the t-th action and $s_{i,t}$ its step score. StepGRPO treats each action independently: it pools all (i,t) actions for q, normalizes $\{s_{i,t}\}$ into per-action advantages, and applies policy updates to each $y_{i,t}$ separately (see Appendix A for details). In contrast, TALE assigns credit at the trajectory level via a progress-aware return (§3.2).

Figure 5 reports the relative improvement over the backbone on BFCL-v3 benchmark. TALE yields larger gains on the multi-turn subset, while both methods perform comparably on the single-turn setting. This suggests that action-level updates can effectively rank locally optimal recovery steps, but fail to capture the temporal dependencies required for modeling recovery as a multi-step process. In contrast, trajectory-level credit with progress shaping facilitates more consistent and progressive improvement across interaction steps.

5 Conclusion

TALE improves the reliability of tool-using LLM agents by learning feedback-guided recovery from their own errors. With a progress-aware reward that encourages directional recovery, TALE boosts both task success and recovery efficiency across four benchmarks. These results underscore learning from error experience as a key step toward more robust tool-using agents.

Limitations

We deliberately study self-correction in structured tool-calling environments, where each action yields explicit execution feedback and thus supports precise diagnosis of recovery dynamics. This controlled scope does not fully instantiate the complexities of general agentic settings (e.g., long-horizon planning and adaptive subgoal management). Future work will apply the proposed framework to training and evaluating agents on broader benchmarks under less constrained interaction dynamics.

References

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, Yifei Chen, Xiaoxi Li, Jiajie Jin, Hongjin Qian, Yutao Zhu, Hangyu Mao, Guorui Zhou, Zhicheng Dou, and Ji-Rong Wen. 2025. Tool-star: Empowering llm-brained multi-tool reasoner via reinforcement learning. *arXiv preprint arXiv:2505.16410*.

Guanting Dong, Keming Lu, Chengpeng Li, Tingyu Xia, Bowen Yu, Chang Zhou, and Jingren Zhou. 2024. Self-play with execution feedback: Improving instruction-following capabilities of large language models. *arXiv preprint arXiv:2406.13542*.

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.

Zihao Feng, Xiaoxue Wang, Bowen Wu, Hailong Cao, Tiejun Zhao, Qun Yu, and Baoxun Wang. 2025. Toolsample: Dual dynamic sampling methods with curriculum learning for rl-based tool learning. *arXiv preprint arXiv:2509.14718*.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2024. Critic: Large language models can self-correct with tool-interactive critiquing. *Preprint*, arXiv:2305.11738.

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. *arXiv preprint arXiv:2506.13977*.

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. 2025b. Teaching language models to reason with tools. *arXiv preprint arXiv:2510.20342*.

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

Xiaoxi Li, Guanting Dong, Jiajie Jin, Yuyao Zhang, Yujia Zhou, Yutao Zhu, Peitian Zhang, and Zhicheng Dou. 2025c. Search-o1: Agentic search-enhanced large reasoning models. *arXiv preprint arXiv:2501.05366*.

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

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, Rithesh RN, and 1 others. 2024b. 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 Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. 2024. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*.

Gregory Polyakov, Ilseyar Alimova, Dmitry Abulkhanov, Ivan Sedykh, Andrey Bout, Sergey Nikolenko, and Irina Piontkovskaya. 2025. Toolreflection: 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-Tür, 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. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2024. From exploration to mastery: Enabling llms to master tools via self-driven interactions. *arXiv preprint arXiv:2410.08197*.

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. *arXiv preprint arXiv:2401.07324*.

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. *arXiv preprint arXiv:2406.19228*.

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*.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, and 1 others. 2025. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*.

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. *arXiv preprint arXiv:2505.16421*.

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. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

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

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.

Yu Yue, Yufeng Yuan, Qiying Yu, Xiaochen Zuo, Ruofei Zhu, Wenyuan Xu, Jiaze Chen, Chengyi Wang, TianTian Fan, Zhengyin Du, and 1 others. 2025. Vapo: Efficient and reliable reinforcement learning for advanced reasoning tasks. *arXiv preprint arXiv:2504.05118*.

Xingshan Zeng, Weiwen Liu, Lingzhi Wang, Liangyou Li, Fei Mi, Yasheng Wang, Lifeng Shang, Xin Jiang, and Qun Liu. 2025. Toolace-mt: Non-autoregressive generation for agentic multi-turn interaction. *arXiv preprint arXiv:2508.12685*.

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*.

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, and 1 others. 2025. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*.

A StepGRPO

A.1 Instance-wise Action Grouping

To provide a controlled comparison against our progress-aware objective, we construct a step-wise baseline that assigns credit at the granularity of individual actions. The key design choice is to normalize advantages over all actions sampled for the same query, which yields a dense learning signal while intentionally removing trajectory-level progress shaping across recovery attempts.

For each query q, we sample a group of G trajectories $\{\tau_i\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(\cdot \mid q)$. Trajectory τ_i consists of T_i actions (recovery attempts), and we denote the token span of its t-th action by $y_{i,t}$. Each action is assigned a fine-grained step score $s_{i,t} \in [0,1]$ using the scoring function in §3.2.1.

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

$$\mathcal{A}(q) = \{(i,t) \mid 1 \le i \le G,\ 1 \le t \le T_i\}, \quad (8)$$

and define the total number of actions under q as

$$M_q = |\mathcal{A}(q)| = \sum_{i=1}^{G} T_i. \quad (9)$$

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

$$A_{i,t}^{\text{ACT}} = \frac{s_{i,t} - \operatorname{mean}(\{s_{j,u}\}_{(j,u)\in\mathcal{A}(q)})}{\operatorname{std}(\{s_{j,u}\}_{(j,u)\in\mathcal{A}(q)}) + \epsilon}. \quad (10)$$

This objective encourages the policy to increase the likelihood of high-quality actions relative to other actions sampled for the same query, providing dense step-level 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 step score $s_{i,t}$ and global grouping normalization, it removes the explicit trajectory-level progress coupling used in TALE, making it a targeted control for isolating the benefit of progress-aware learning.

A.2 StepGRPO Objective

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

$$\rho_{i,t}(\theta) = \frac{\pi_\theta(y_{i,t} \mid q, y_{i,<t})}{\pi_{\theta_{\text{old}}}(y_{i,t} \mid q, y_{i,<t})}, \quad (11)$$

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

The step-wise objective mirrors Eq. 7 but applies clipping and advantages at the action level:

$$\begin{aligned}\mathcal{J}_{\text{StepGRPO}}(\theta) = \mathbb{E}_{q\sim\mathcal{D},\,\{\tau_i\}_{i=1}^{G}\sim\pi_{\theta_{\text{old}}}(\cdot|q)}\Bigg[\frac{1}{M_q} \\ \sum_{(i,t)\in\mathcal{A}(q)} \min\Big(\rho_{i,t}(\theta)\, A^{\text{ACT}}_{i,t}, \\ \text{clip}\big(\rho_{i,t}(\theta),\,1-\epsilon,\,1+\epsilon\big)\, A^{\text{ACT}}_{i,t}\Big) \\ -\,\beta\, \mathrm{D}_{\text{KL}}\big(\pi_\theta \,\|\, \pi_{\text{ref}}\big)\Bigg]. \qquad (12)\end{aligned}$$

Step-wise Credit Assignment. In practice, $A^{\text{ACT}}_{i,t}$ is broadcast to all tokens within the span $y_{i,t}$ so that the policy gradient assigns credit to the entire action realization. This is consistent with treating each repair attempt as an atomic decision: all tokens within $y_{i,t}$ jointly determine the action's syntactic validity (e.g., well-formed tool calls) and semantic correctness (e.g., correct function and arguments). Broadcasting, therefore, provides a stable step-wise learning signal while preserving the token-level likelihood factorization required by GRPO optimization.

B Training Details

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 5 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 6 reports the SFT hyperparameters.

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	$\{\text{GRPO, GSPO}\}$

Table 5: 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 6: Hyperparameter configuration for SFT training.

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 constructed environment (see Appendix E) 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 reconstruct the training data on our internal split using a multi-step rollout protocol. Specifically, we run the current policy in the environment to collect multi-step interaction trajectories. To align with the core principle of ToolReflection–learning from self-generated failures and subsequent corrections—we apply a filtering rule and retain trajectories that exhibit recovery behavior. Concretely, a trajectory is kept if the model encounters an API error or clearly suboptimal execution feedback at an intermediate step, but later revises its tool-use decisions and ultimately completes the task successfully. We format the retained trajectories by conditioning the model on the full interaction history, including prior actions. The training objective is to predict the final corrective action given this context.

Hyperparameter Sensitivity. We study the sensitivity to the rollout interaction budget by varying the maximum number of attempts in multi-step training, $T \in \{1, 2, 3, 4\}$, while keeping the backbone and the remaining training recipe fixed. Figure 6 reports the relative improvement over the baseline on the BFCL-v3 benchmark. Performance improves as T increases from 1 to 3, peaks at $T = 3$, and degrades at $T = 4$. Accordingly, we set $T = 3$ for all subsequent experiments.

C Robustness to Underspecified Feedback

TALE is motivated by the observation that tool feedback in deployment is heterogeneous and often underspecified: some tools expose structured diagnostics, while others return sparse, ambiguous, or even misleading signals. An open question is whether TALE's repair learning depends on rich error attribution, or whether progress-aware training can still induce effective self-correction when feedback is reduced to a minimal retry signal. To examine this robustness dimension, we ablate the feedback interface while keeping the backbone and the GSPO training recipe fixed.

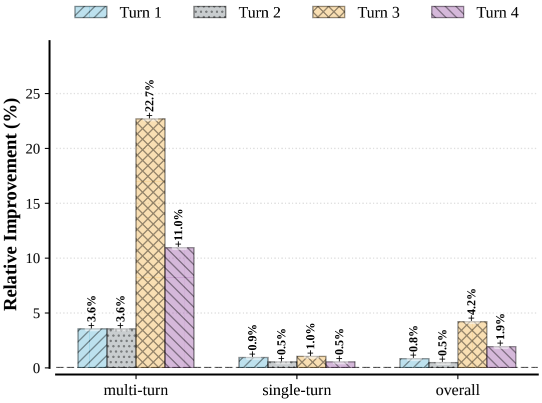


Figure 6: Rollout budget sensitivity. Relative improvement over the baseline on BFCL-v3 as a function of the maximum rollout attempts $T \in \{1, 2, 3, 4\}$.

Module	Base	Miss Func	Miss Param	Long Context	Overall
Error-fix	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 7: Impact of feedback design on BFCL-v3 subsets under the same training recipe.

Feedback variants. We compare three regimes: (i) *Static error-fix imitation* (ERROR-FIX), trained from static error–fix trajectories without on-policy interaction; (ii) *TALE (detailed feedback)*, which provides attributed natural-language diagnostics (e.g., wrong function selection, missing arguments, key mismatch); and (iii) *TALE (coarse feedback)*, which maps all failures to a single generic message ("Response error, please retry"), removing attribution and retaining only a retry affordance. All settings use the same rollout horizon, reward definition, and optimization hyperparameters; only the feedback content provided to the policy is varied.

Results and training dynamics. As shown in Table 7, on-policy interaction with execution outcomes is critical: both interactive TALE variants outperform ERROR-FIX in overall score (34.1/34.5 vs. 31.8). This aligns with our central premise that robust self-correction is not well captured by static error–fix imitation alone, but benefits from learning from the agent's own error experiences through iterative repair attempts. Notably, collapsing feedback to a generic retry signal does not eliminate TALE's gains: *TALE (coarse feedback)* matches or exceeds *TALE (detailed feedback)* overall, while detailed

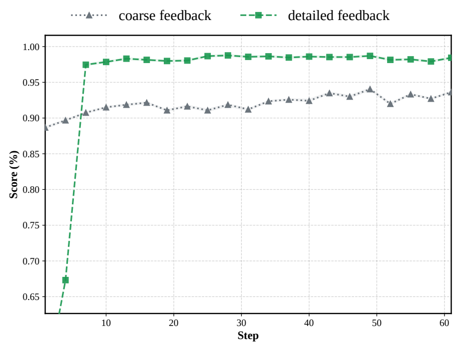


Figure 7: Training curves of TALE under different feedback interfaces, measured by Score@1.

diagnostics remain most beneficial for parameter-level recovery (MISS PARAM: 31.5 vs. 30.0).

Figure 7 further compares optimization dynamics under the two 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 §4.4, 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.

D Fine-grained Stepwise Scoring

When evaluating tool call generation, an action can produce multiple predicted calls $\mathcal{C} = \{c_1, \ldots, c_n\}$, while the ground truth requires $\mathcal{C}^* = \{c_1^*, \ldots, c_m^*\}$. To fairly assess the quality of these predictions, we employ a matching-based scoring approach that accounts for partial correctness at both the call and parameter levels. We first establish an optimal matching $\mathcal{M}$ between predicted and ground-truth calls by greedily maximizing total similarity. The algorithm iteratively selects the highest-scoring pair (c_i, c_j^*) according to the score function defined in §3.2.1, adds this pair to M, removes both calls from consideration, and repeats until no calls remain in either set. This produces the aggregate actual score:

$$s_{\text{actual}} = \sum_{(c_i, c_j^*) \in \mathcal{M}} S(c_i, c_j^*) \tag{13}$$

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 with parameter set P of size $|P|$, the maximum possible score is: $s_{\max}(c) = 1 + 1 + |P| = 2 + |P|$. This decomposition reflects our 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:

$$s_{\max}^{\text{pred}} = \sum_{j=1}^{n} (2 + |P_j|), \quad s_{\max}^{\text{gt}} = \sum_{i=1}^{m} (2 + |P_i^*|) \tag{14}$$

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

$$\mathcal{P} = \frac{s_{\text{actual}}}{s_{\max}^{\text{pred}}} \tag{15}$$

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

$$\mathcal{R} = \frac{s_{\text{actual}}}{s_{\max}^{\text{gt}}} \tag{16}$$

The final step score combines both metrics via their harmonic mean (F1 score):

$$s_t = \frac{2\mathcal{P}\mathcal{R}}{\mathcal{P} + \mathcal{R}} \tag{17}$$

This balanced metric penalizes both over-generation (low precision) and under-generation (low recall), providing a comprehensive assessment of tool call quality.

E Environment Construction

We implement a deterministic interaction environment $\mathcal{E}$ to provide structured tool feedback at each recovery attempt. To ensure reproducibility and cost-efficiency, we adopt an offline protocol: instead of executing real API calls, $\mathcal{E}$ evaluates the model-predicted tool calls (Pred) by comparing them against the benchmark-provided ground truth (GT). When a mismatch is detected, the environment returns a diagnosis and a natural-language feedback message (wrapped in `<tool_response>` `</tool_response>`) that specifies the failure type, which is then appended to the interaction history for subsequent attempts. We categorize tool-use failures into four common error types, each paired with a fixed feedback template, following the error taxonomy in Huang et al. (2025).

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.

F 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 environment 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:

```
<tools>
....
</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: From repetitive error loops to directed recovery. In this case, both the backbone and TALE correctly identify the target file using `find` (`./project/analysis_report.csv`), but they diverge in how they act under truncated reasoning and execution feedback. After the initial success, the backbone treats subsequent attempts largely as independent retries: it issues `mv` without first aligning the execution context to the file's directory, and when the environment repeatedly returns `No such file or directory`, it makes superficial adjustments to the destination path (e.g., `./archive` vs. `../archive`) while leaving the underlying state and key assumptions unverified. As a result, the agent fails to convert feedback into a directed repair decision and enters a persistent error loop with little incremental progress.

In contrast, TALE explicitly conditions on interaction history to construct a progress-aware repair sequence: it first enforces the user constraint "`archive/` in the same directory" by switching into the file's directory (`cd project`), treats `mkdir: File exists` as a recoverable signal, and then applies minimal, feedback-driven argument edits to satisfy tool-specific constraints, ultimately completing the move and terminating promptly. This result shows that TALE can efficiently leverage execution feedback to perform targeted repairs.

Case Study: Premature abandonment vs. adaptive retrieval under weak feedback. In this file-retrieval task, both the backbone and TALE follow the user instruction to enter the `project` directory via `cd`, and both initially attempt to locate the report using `find`. However, the tool feedback is uninformative: `find` returns an empty match list, which does not distinguish between a missing file and a name mismatch (e.g., paraphrased user reference vs. actual filename). The backbone exhibits rigid, literal matching and prematurely terminates after a single failed lookup, treating the first negative result as a final answer rather than a cue for iterative repair. Concretely, it makes no further progress-oriented actions (e.g., inspecting directory contents or testing plausible filename variants), and thus fails to recover from an under-specified query.

In contrast, TALE interprets the same weak signal as requiring additional evidence and performs targeted exploration by listing directory contents with `ls`. It then maps the user intent "Kelly's test report" to a plausible candidate (`test_report.docx`) and verifies this hypothesis with a minimal follow-up read (`cat`), successfully retrieving the relevant content. This contrast illustrates that robust tool use under heterogeneous feedback requires progress-aware recovery strategies that expand the search space and validate candidates, rather than prematurely abandoning after the first non-diagnostic failure.

Case Study (Backbone): Repeated Failures with Truncated Thinking

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 and error repetition
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 failure 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 (TALE): 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): Task Failure due to Rigid Reasoning and Premature

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 tur n</tool_response>`

Case Study (TALE): 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!